# Efficient Combination of Decision Procedure for MUS Computation

Cédric Piette, Youssef Hamadi, Lakhdar Saïs

# Efficient Combination of Decision Procedures for MUS Computation

Cédric Piette[1], Youssef Hamadi[2], and Lakhdar Saïs[1]

[1] Université Lille-Nord de France, Artois
CRIL-CNRS UMR 8188
F-62307 Lens Cedex, France
{piette,sais}@cril.fr

[2] Microsoft Research
7 J J Thomson Avenue
Cambridge, United Kingdom
{youssefh}@microsoft.com

**Abstract.** In recent years, the problem of extracting a MUS (Minimal Unsatisfiable Subformula) from an unsatisfiable CNF has received much attention. Indeed, when a Boolean formula is proved unsatisfiable, it does not necessarily mean that all its clauses take part to the inconsistency; a small subset of them can be conflicting and make the whole set without any solution. Localizing a MUS can thus be extremely valuable, since it enables to circumscribe a minimal set of constraints that represents a cause for the infeasibility of the CNF. In this paper, we introduce a novel, original framework for computing a MUS. Whereas most of the existing approaches are based on complete algorithms, we propose an approach that makes use of both local and complete searches. Our combination is empirically evaluated against the current best techniques on a large set of benchmarks.

## 1 Introduction

The concept of Minimally Unsatisfiable Subformula (MUS for short) proves more and more valuable in many applications of SAT, such as bounded model checking (see e.g. [1, 2]) or knowledge-base validation. Indeed, when a Boolean formula is proved unsatisfiable, it does not necessarily mean that all its clauses take part to the inconsistency; a small subset of them can be conflicting and make the whole set without any solution. Localizing a MUS can thus be extremely valuable, since it enables to circumscribe a minimal set of constraints that represents a cause for the infeasibility of a CNF formula. For instance, in [3], an abstraction-based approach for deciding the satisfiability of finite-precision bit-vector is presented. Roughly, this method consists in generating an under-approximation such that if this latter one is satisfiable, so is the original problem. In the opposite case, an approximation of MUS (also called core) of this CNF formula is computed for generating an over-approximation whose unsatisfiability proves that the original

problem has no solution. Experimental results clearly show that this way is more efficient than running a complete DPLL-like procedure on the whole original formula.

SAT solving has made a wonderful progress in the recent years, and some problems considered out-of-reach a decade ago can now be solved in a few seconds. A natural consequence of those algorithmic improvements is the modeling of larger problems whose reasons for unsatisfiability can be very difficult to catch. One way to explain inconsistency is to extract a Minimally Unsatisfiable Subformula, which represents an irreducible (w.r.t. the involved clauses) cause of the unsatisfiability. With the recent increase in formulae sizes, the problem of extracting one or several MUS has received much attention these last years. However, the current approaches which are based on complete search algorithms (DPLLs) often deliver an unsatisfiable subformula, which is not guaranteed to be minimal. Among these techniques, let us mention `AMUSE` [4] which consists in marking clauses during the exploration of a DPLL-related search tree. Another technique called `zCore` [5], the core extractor related to the well-known solver `zChaff`, is able to localize an unsatisfiable subformula by analyzing the resolution graph of a complete search procedure. Indeed, the source clauses from which is derived the empty one by the resolution process, simulated by modern solvers, forms a non-minimal core of the CNF formula.

More recently, a new technique called `AOMUS` [6] has also been introduced. It makes an original use of local search and relies on a so-called critical clause concept, to progressively focus on the difficult parts of a problem. This allows `AOMUS` to compute the upper-approximation of a MUS. Finally, an approach based on a serie of redundancy checks is presented in [7]. Let us also note that the problem of finding a Minimal Unsatisfiable Core (MUC) in the CSP framework is also the subject of many publications [8, 9].

Although these approaches are based on different algorithmic principles, they are *destructive* by nature, in the sense that they consider the whole formula to *remove* parts which have been shown useless in a particular proof of unsatisfiability. In this paper, we are concerned with a new *constructive* framework. Indeed, this approach starts without any clause and builds a core by adding some particular clauses regularly, until an unsatisfiable set of clauses (which represents an approximated MUS) has been obtained. This framework makes an incomplete search – in charge of delivering sets of models – and an exhaustive technique act together. This can be opposed to destructive approaches which often imply the ability to prove a CNF unsat before starting a core extraction process. For applications like MaxSat solving through detection of cores [10], this could be an issue, though. Our constructive method does not suffer from this drawback, as it only requires to check the unsatisfiability of the discovered core instead of repetitively proving the whole CNF and some of its suformulae inconsistent.

To reduce the number of calls to a complete DPLL solver, our combination exploits the power of stochastic local search (SLS) for both finding models (proving the consistency of the current subformula) and for selecting the set of relevant clauses that might be added to the current subformula. A call to

a complete solver is only done when the SLS method is not able to prove the consistency of the formula.

The paper is organized as follows. After some preliminary definitions and notations about SAT and MUS, we describe in section 3 the constructive framework. In section 4, an implementation of the framework is presented and experimented in section 5. Finally, we conclude by providing some interesting future lines of research.

## 2 Technical Beackground

Let $L$ be the propositional language of formulas defined in the usual inductive way from a set $P$ of propositional symbols (represented using plain letters like $a, b, c$, etc.), the Boolean constants $\top$ and $\bot$, and the standard connectives $\neg$, $\wedge$, $\vee$, $\Rightarrow$ and $\Leftrightarrow$. A SAT instance is a propositional formula in conjunctive normal form (CNF for short), i.e. a conjunction of clauses, where a clause is a disjunction of literals, a literal being a possibly negated propositional variable.

An interpretation is a function that assigns values from $\{true, false\}$ to every Boolean variable. An interpretation is called *model* w.r.t. a particular CNF if it satisfies this formula, namely if this interpretation makes it *true*. SAT is the well-known NP-complete problem that consists in deciding whether a propositional CNF is satisfiable or not, i.e. whether the formula admits at least one model. In this paper, CNF formulae are represented using a set (interpreted as a conjunction) of clauses, where a clause is a set (interpreted as a disjunction) of literals.

When a CNF is satisfiable, most of the current approaches are able to provide a model, which is a certificate of its satisfiability. By opposite, in case of inconsistency, these approaches only ensure that no satisfying interpretation exists. However, this is not very informing. The user might prefer to localize, if there exists, a small part of the CNF which triggers the inconsistency of the whole problem. In this respect, the concept of MUS (Minimally Unsatisfiable Subformula) is extremely valuable.

**Definition 1.** *Let $\Sigma$ be a CNF formula. $\Gamma$ is a MUS (Minimally Unsatisfiable Subformula) of $\Sigma$ iff:*

1. *$\Gamma \subseteq \Sigma$;*
2. *$\Gamma$ is unsatisfiable;*
3. *for each $\Gamma' \subset \Gamma$, $\Gamma'$ is satisfiable.*

Extracting of MUS from an unsatisfiable CNF is a highly complex problem. For instance, checking if a CNF is a MUS is DP-complete [11] (where DP is the class composed of the union of a language in NP and another one in CoNP), while checking whether a set of clauses belongs to at least one MUS of an unsatisfiable CNF is $\Sigma_2^p$-hard [12].

Despite these not encouraging complexity results, many approaches for extracting a *core* (i.e. an approximated MUS) have already been proposed. In the

next section, we propose a new constructive framework to extract cores from unsatisfiable formulae.

## 3 A Constructive Framework

### 3.1 Presentation

Most of the current approaches are destructive, in the sense that they consider the whole formula to progressively reduce it toward a core. In order to present the ideas of this new framework in comparison to the destructive one, we introduce here some notations that will be used in the remaining of this paper. First of all, let us denote $\Sigma$ the CNF from which a MUS (or an approximation) is to be extracted. We also denote $\Gamma$ another set of clauses which will contain the computed core. This set contains at any step of the procedure a subset of original formula $\Sigma$. Generally, the algorithms for computing a MUS are based on the iteration of the same function. So, let $f$ be a function that takes $\Sigma$ and $\Gamma$ as input to produce a subformula $\Gamma_N$.

With destructive methods, $\Gamma$ is initially set to $\Sigma$. In addition, $f$ is a function that takes $\Sigma$ and $\Gamma$ as input to produce a subformula $\Gamma_N$ of $\Gamma$, the most often by analyzing a proof of unsatisfiability. The resulting CNF $\Gamma_N$ is then a subset of $\Gamma$. This new CNF is then used as $\Gamma$ with respect to $f$, to obtain a potentially even smaller CNF. This iteration is performed until the delivered formula is $\Gamma$ itself, namely until a fix point is reached.

For the constructive method, we propose to start the computation considering $\Gamma$ as the empty set, and to progressively add clauses from $\Sigma$ in $\Gamma$. So, a core is *built* thanks to $f$ which returns a superset $\Gamma_N$ of $\Gamma$ which contains new clauses from $\Sigma \backslash \Gamma$. This operation is repeated until $\Gamma$ becomes unsatisfiable and represents a core for $\Sigma$. The following table summarizes the main differences between those general frameworks.

| | Constructive | Destructive |
|---|---|---|
| Initial $\Gamma$ | $\emptyset$ | $\Sigma$ |
| $f(\Sigma,\Gamma)=\Gamma_N$ | $\Gamma \subseteq \Gamma_N$ and $\Gamma_N \subseteq \Sigma$ | $\Gamma_N \subseteq \Gamma$ |
| Stopping Condition | $\Gamma$ is UNSAT | $f(\Sigma,\Gamma)=\Gamma$ |

Actually, constructive procedures have already been studied for the problem of minimizing an unsatisfiable system toward a minimal one, in order to extract an exact cause of its infeasibility. For example, this approach is known for a long time in the mathematical programming [13] and CSP [14] communities. However, it is not used in practice any more, because of its complexity in the worst case. Indeed, by considering the number of calls to a complete procedure and denoting $n$ the number of constraints of the CNF, and $c_n$ the number of constraints of the biggest MUS (in number of constraints), the constructive approach exhibits

a worst case complexity in $\mathcal{O}(n \times c_n)$, whereas the worst case complexities of destructive and dichotomic approaches are only $\mathcal{O}(n)$ and $\mathcal{O}(log(n) \times c_n)$ [8], respectively.

Fortunately, this is only true for systematic complete procedures that aim at returning a minimal set of clauses; in this paper, we focus on approaches that aim at approximating a MUS (delivering an upper set of clauses). Consequently, they do not suffer from this bad complexity result. In the next section, we discuss about particular clauses that can be safely added to $\Gamma$ without discarding any MUS of the formula.

### 3.2 The role of necessary clauses

In a recent study which aims at categorizing the clauses of a formula w.r.t. their role in its unsatisfiability [15], different classes of clauses have been established. Roughly, a clause is called *necessary* if it belongs to all MUS of a formula. Clearly, those clauses participate to any proof of inconsistency, and removing any one of them makes the CNF recover satisfiability. A clause is *potentially necessary* if it is not a necessary clause but the removal of some particular clause(s) can make it necessary. Those clauses appear in at least one MUS of the formula. Finally, a clause which is neither necessary nor potentially necessary is *never necessary* and does not belong to any MUS. They can be used in a proof of unsatisfiability, but all of them can be removed from the CNF keeping it unsatisfiable.

Necessary clauses are computationally hard to be extracted: indeed, for an unsatisfiable CNF $\Sigma$ and a clause $c \in \Sigma$, if $\Sigma \backslash \{c\}$ is satisfiable, then $c$ is necessary with respect to $\Sigma$. A linear number of calls of a complete algorithm is thus needed to exhaustively extract all necessary clauses of a given CNF. Fortunately, an inexpensive technique [6] can find them in an incomplete way. During the exploration of a local search for the satisfiability of $\Sigma$, if all its clauses are satisfied except one, the falsified clause is clearly necessary and is marked as *protected*. As necessary clauses represent global minima for the objective function of the local search, this notion proves really efficient for most of benchmarks, in an empirical point of view. Actually, most of the necessary clauses can be computed through the protected clause concept on many instances, but the approach cannot ensure they have all been delivered.

Nevertheless, an unsatisfiable CNF can exhibit no necessary clause. Indeed, containing two independent causes of unsatisfiability, namely two MUS with an empty intersection, is a sufficient condition for not exhibiting such clauses. However, most of realistic CNF which are expected to be satisfiable are not of this form. As said previously, a necessary clause of a formula belongs to any of its MUS and can therefore be safely added to $\Gamma$, in our constructive framework.

The main difference between destructive procedures is the strategy used to select the subset of clauses of $\Gamma$ (represented previously by the function $f$). With respect to a constructive policy, a lot of possible techniques can also be imagined for augmenting $\Gamma$. However, when $f$ only considers particular clauses to be added, interesting properties can be established. The interest of such particular functions are described in the next section.

### 3.3 The non-redundancy property

Our technique is based on the following proposition, which is a straight consequence of the definition of MUS itself:

**Proposition 1.** *Let $\Sigma$ be a CNF formula, and $\omega$ an interpretation of $\Sigma$. $\forall M \in \Sigma$ s.t. $M$ is a MUS, $\exists c \in M$ s.t. $\omega \not\models c$.*

Clearly enough, as a MUS is an irreductible unsatisfiable set of clauses, any interpretation of a formula falsifies at least one clause from each of its MUSes. Accordingly, when considering a large number of interpretations, the most often falsified clauses are heuristically a good approximation of clauses belonging to MUSes. The highest score clauses are then added to the growing core. Thus, we associate a score (initialized to 0) to each clause, and while a large number of interpretations are considered, the score of each falsified clause is increased by 1.

A lot of different constructive algorithms can be created, choosing various strategies to enumerate interpretations. Those ones can be randomly generated, but it appears reasonable to attempt to reduce the number of clauses falsified by the explored interpretations. Indeed, some falsified clauses may not appear in any MUS and are just "noise" for the heuristic. Attempting to reduce the number of falsified clauses limits this phenomenon. Hence, meta-heuristics designed for optimisation problems (simulated annealing, genetic algorithms, etc.) can be used in this context.

Moreover, if the scoring function only considers falsified clauses of $\Sigma$ with respect to models of $\Gamma$, then the following proposition ensures that no clause redundant with $\Gamma$ can be added to this set of clauses. Those redundant clauses are clearly superfluous when considering the growing core.

**Proposition 2.** *Let us consider the CNF formulae $\Sigma$, $\Gamma$ and the function $f_{S_\omega}$ defined previously in the constructive context. $f_{S_\omega}$ depends on a set of explored interpretations $S_\omega$. The following proposition holds:*
*if $f_{S_\omega}(\Sigma, \Gamma) \subseteq \Gamma \cup \{c \in \Sigma \backslash \Gamma \mid \exists \omega \in S_\omega$ s.t. $\omega \models \Gamma$ and $\omega \not\models c\}$*
*then $\nexists c \in (f(\Sigma, \Gamma) \backslash \Gamma)$ such that $\Gamma \models c$.*

Thus, within this condition, it is not possible for our constructive policy to add to the built core, the clauses which are redundant with $\Gamma$, namely clauses which "cover" the same part of the search space. This feature is obviously not shared by destructive methods. However, if $f$ enables to add several clauses at the same time, a subset of them can clearly be redundant with another one in conjunction of $\Gamma$. Nevertheless, this is only possible for clauses added at the same time. The number of added clauses after each iteration can be tuned through the definition of $f$ itself. If it is small, a bigger number of iterations would be necessary to construct a core, but this one would exhibit a good quality. In the opposite, with a big number of clauses added, only a few iterations are sufficient to obtain a core, but it will probably not be fine-grained. This property makes

the constructive approaches very flexible, since it provides a parameter that represents a trade off between the quality of the obtained core and the time spent for its computation. Destructive approaches cannot propose to the user such a parameter. Let us illustrate the non-redundancy property through an example.

*Example 1.* Let $\Sigma$ be a CNF formula made of 13 clauses, involving 4 variables such that:

$$\Sigma = \left\{ \begin{array}{lcllcl} c_1 & : & (\neg a \vee c \vee d) & c_2 & : & (a \vee b \vee c) \\ c_3 & : & (\neg a \vee \neg b \vee c) & c_4 & : & (c \vee \neg d) \\ c_5 & : & (\neg a \vee \neg c \vee d) & c_6 & : & (\neg b \vee d) \\ c_7 & : & (a \vee \neg c \vee d) & c_8 & : & (a \vee \neg b \vee \neg c) \\ c_9 & : & (b \vee \neg c) & c_{10} & : & (\neg b \vee \neg c \vee \neg d) \\ c_{11} & : & (a \vee \neg c \vee \neg d) & c_{12} & : & (a \vee c \vee d) \\ c_{13} & : & (\neg a \vee b \vee \neg d) & & & \end{array} \right.$$

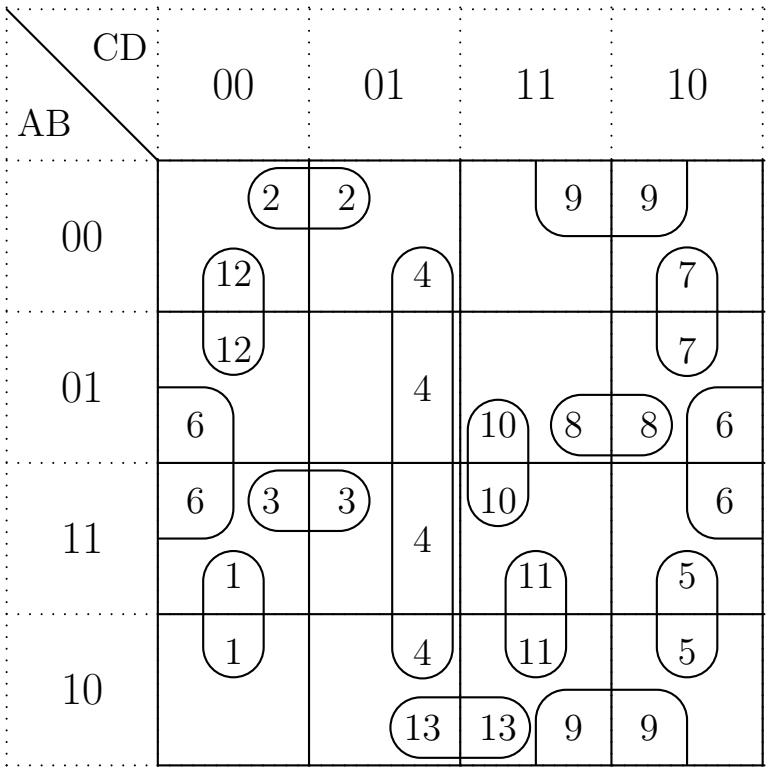
The search space induced by those 4 variables is represented through a Karnaugh map in Figure 1a, together with each clause of $\Sigma$ falsified by the different possible interpretations. Let us note that $\Sigma$ is clearly unsatisfiable, since every interpretation is falsified by at least one clause of the CNF. As an example, $\omega_1 = \{\neg a, b, c, d\}$ falsifies the clauses $c_8$ and $c_{10}$.

Let us note that some interpretations are only falsified by a unique clause. For instance, $\omega_2 = \{\neg a, b, \neg c, d\}$ is only violated by $c_4$. Accordingly, this clause is necessary [15], since without it, $\omega_2$ would become a model of $\Sigma$. $c_4$ thus participates to all sources of inconsistency of $\Sigma$ and belongs to all its MUS. $\Sigma$ exhibits two other necessary clauses, which are $c_1$ and $c_9$. Indeed, we can observe in Figure 1a that there exists at least one interpretation which is only "covered" (i.e. falsified) by one of these clauses.

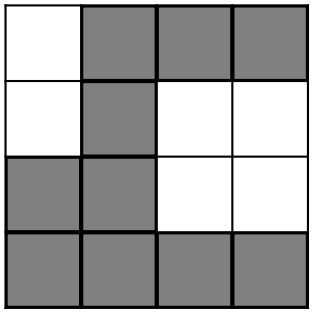
Assume that those necessary clauses are first computed, as presented previously. $\Gamma$ is then set to $\Gamma = \{c_1, c_4, c_9\}$. Consequently, only the models of this latter CNF are now considered for selecting the other clauses of $\Sigma$. The set of models of $\Gamma = \{c_1, c_4, c_9\}$ is represented in Figure 1b using white cells, the interpretations that falsify this CNF being in gray. For instance, the clauses $c_3$ and $c_{13}$ cannot be added to $\Gamma$ any more, since they are falsified iff at least one necessary clause is also falsified. Hence, they do not belongs to any MUS of $\Sigma$. With destructive methods, proofs of inconsistency can use these clauses which can then be present in the delivered core.

Thus, if $f$ only considers the falsified clauses of $\Sigma$ with respect to models of $\Gamma$, then $c_{12}$ would be candidate to the built core and in this case would also prevent $c_2$ and $c_6$ from being added in the future, since these latter clauses cannot be falsified w.r.t. a model of $\Gamma \cup \{c_{12}\}$.

However, if $f$ enables to add simultaneously several clauses, $c_2$ and $c_{12}$ could be both added to $\Gamma$, whereas $c_2$ is a logical consequence of $\Gamma \cup \{c_{12}\}$ ($c_2$ covers an underset of interpretations covered by $\Gamma \cup \{c_{12}\}$ in Figure 1a).




(a) Karnaugh map of Example 1



(b) Models of the CNF composed of necessary clauses of $\Sigma$

Fig. 1: Graphical representation of example 1

Moreover, the traditional approaches aiming at extracting a MUS, or an approximation, from an unsatisfiable CNF delivers an arbitrary one, based on the refutation proof obtained. However, a propositional formula can contain several MUS (actually an exponential number of MUS in the worst case), and the user

might prefer to extract a core that contains specific information, or clauses. Such a computation can be very useful: as an example, in FPGA routing task, in order to check if a particular part of the circuit is involved in the inroutability of the layout. Classical approaches cannot perform such a computation, and the only way to do this until now was to use a complete procedure, namely an approach that computes the exhaustive set of MUS, like [16] does, for instance.

The constructive policy which respects the non-redundancy property offers such a choice to the user by guiding a search toward a specific core. To this purpose, instead of starting the computation with the empty formula, it suffices to begin with the chosen clauses. Let us denote $\Delta$ the set of clauses selected by the user to build a core. As shown previously, some clauses can be redundant w.r.t. the rest of the formula, and the constructive approach ensures that none of them can be added by the algorithm. Hence, by setting the first set of clauses to $\Delta$, no other constraint redundant with it can be considered by the procedure. The following example shows the interest and the limitation of this kind of approach.

*Example 2.* Let us consider the CNF presented in the Example 1. Assume the user wants to compute a core involving the set of clauses $\{c_1, c_3, c_8, c_{12}\}$.

First, $c_1$ is a necessary clause, and thus belongs to all MUS/cores of $\Sigma$ anyway. On the opposite, $c_3$ is a *never necessary* clause (w.r.t. the terminology of [15]) and does not belongs to any MUS. Consequently, the user's choice implies a non-optimal construction of core. $c_8$ and $c_{12}$ are *potentially necessary* clauses, and belong to at least one MUS of the CNF. By adding those clauses before the first iteration, assuming that the 3 necessary clauses are first detected, $\Gamma$ contains the clauses $\{c_1, c_3, c_4, c_8, c_9, c_{12}\}$. Using this "partial core", the only clauses that can be falsified with respect to these models are $c_5$, $c_6$, $c_{10}$ and $c_{11}$, and some of them will then be added to $\Gamma$ to construct a core. If $c_5$ and $c_{11}$ are selected, the resulting core can only be minimized by removing $c_3$ and all the clauses chosen by the user (except this last one) do participate to the computed MUS. Unfortunately, the potential choice of $c_6$ and $c_{10}$ to start the core construction would make $c_8$ redundant, and not belong to all its MUS.

This example shows the constructive approach's flexibility. It enables the end-user to guide the computation toward a specific core, and might be driven by the user's expertise on the problem. Clearly, the proposed process is not limited to the initial core, and different user-interactions during the construction of the core can be imagined. However, this is done without ensuring that the chosen clauses will actually be a part of all MUS of this core. We believe that this incomplete computation is a good tradeoff in front of the very high complexity of such a problem. Indeed, assuming $P \neq NP$, we cannot expect the development of an efficient complete procedure, since checking whether a formula belongs to the set of MUS of an unsatisfiable SAT instance is in $\Sigma_2^p$ (a consequence of theorem 8.2 of [12]). In the next section, we present a first implementation of this constructive approach which uses local search for adding new clauses to $\Gamma$.

## 4 A First Implementation: `constructMUS`

In this section, we present a particular implementation of the constructive framework that respects the non-redundancy property. The function $f$ is based on local search for choosing clauses to be added. Initially, this stochastic technique is used for extracting protected clauses, and initialize $\Gamma$ [6].

We then perform multiple runs of a local search on $\Gamma$ and consider the discovered models. We then consider the falsified clauses of $\Sigma$ and increment a related score, initially set to 0 for each clause. As at least one clause of each set that could complement $\Gamma$ to form a MUS is falsified, it suffices to consider the clauses that have obtained the highest score and add them to the approximation of the MUS. Accordingly, at the end of the incomplete procedure, $\Gamma$ is augmented with these "high-score clauses" and these operations are repeated, searching models to the new set of clauses.

This process is iterated until no model can be found to the growing formula $\Gamma$. However, due to the incomplete nature of the local search, this computed set of clauses is not guaranteed to be unsatisfiable, even if in practice this case occurs quite often. To ensure the completeness of this algorithm, when the local search fails to find a model of $\Gamma$, we run a DPLL complete method on this formula. If $\Gamma$ is proved unsatisfiable, then $\Gamma$ is returned as the computed MUS approximation. Otherwise, a model $\omega$ which represents a certificate of the satisfiability of $\Gamma$, is discovered by the complete approach; in this case, the model is used by the local search as its first explored interpretation, ensuring at least one model will be discovered. Moreover, as local search has not found a model previously, this means that this part of the search space has probably not been explored by the incomplete method. Forcing the model as the seed enables to diversify the search, and to find other "close" models.

With these features, the proposed constructive algorithm is sound for the problem of extracting an unsatisfiable subformula from any CNF. This approach is synthesized in the Algorithm 1. As explained previously, the considered "core" $\Gamma$ is initially set of the empty formula, and only clauses found necessary (in a very cheap way) through the protected clause concept (line 4) are added. A flag $call\_DPLL$ is set initially to $false$, which implies the computation of a sample of models of $\Gamma$ by a local search call (line 14). The $p$ most falsified clauses of the remaining part of the formula are added to $\Gamma$ (lines 18-19). Let us note that this parameter plays a role in the quality of the core and in the runtime of the procedure, since it provides the number of added clauses at each iteration. These operations are iterated until no model can be found by LS to $\Gamma$.

If no model is found by LS, then the flag $call\_DPLL$ is set to $true$ (line 16) and a complete approach is run on $\Gamma$ (line 8). If $\Gamma$ is unsatisfiable, it is returned as the computed core; in the opposite case, a model is found by the procedure and is used as a "seed" in the next call of the local search (line 11). In the next section, this first implementation of the constructive framework is empirically evaluated.

**Algorithm 1**: `construct_MUS`

```
   Input: an unsatisfiable CNF: Σ
   Output: a core of Σ: Γ
 1 begin
 2     call_DPLL ← false ;
 3     Γ ← ∅ ;
 4     Γ ← search_protected_clauses() ;
 5     Σ ← Σ\Γ ;
 6     while true do
 7         if call_DPLL then
 8             if DPLL(Γ) = UNSAT then
 9                 return Γ ;
10             else
11                 seed ← found_model ;
12                 call_DPLL ← false ;
13         else
14             seed ← random_interpretation() ;
15         S_ω ← LS_finds_models(Γ,seed) ;
16         if S_ω = ∅ then
17             call_DPLL ← true ;
18         else
19             Φ ← select_most_falsified_clauses(Σ, S_ω, p) ;
20             Γ ← Γ ∪ Φ ;
21             Σ ← Σ\Φ ;
22 end
```

## 5 Experiments

In order to assess the ability of the proposed constructive framework to efficiently extract cores, we have implemented the ideas described in the previous sections, and compared them to three state-of-the-art approaches on various instances from SATLIB and the SAT competitions. As a case study, we have used the `RSAPS` [17] local search. For the complete approach, `Minisat` [18] has been chosen, since it is recognized by the community as one of the best complete solver. Based on empirical tuning, the parameter $p$ of `constructMUS` procedure has been set to $(1 + \#cla_{\Sigma} \times 3\%)$ and the flips limit devoted to the local search runs at each iteration of the algorithm has been set to $((1 + \#cla_{\Gamma}) \times 100)$, where $\#cla_{\Sigma}$ is the number of clauses of $\Sigma$ not added to $\Gamma$, and $\#cla_{\Gamma}$ the number of clauses of the current sub-formula $\Gamma$.

We compare `constructMUS` to 3 state-of-the-art approaches: `zCore` [5], `AMUSE` [4] and `AOMUS` [6]. The Table 1 reports these experiments. For each result, the size (in terms of number of clauses) of the delivered approximation of MUS is reported together with the time needed for the computation. Moreover, for those two values, the best obtained one is given in bold. All experiments have been

| Instances | | | zCore | | constructMUS | | AMUSE | | AOMUS | |
|---|---|---|---|---|---|---|---|---|---|---|
| Name | #var | #cla | #cla | time | #cla | time | #cla | time | #cla | time |
| 23.shuffled | 198 | 474 | **221** | 0.05 | **221** | 4.83 | 230 | **0.04** | **221** | 2.05 |
| 42.shuffled | 378 | 904 | **421** | **0.08** | **421** | 5.00 | 434 | 0.09 | **421** | 3.80 |
| 3col20_5_5 | 40 | 176 | 46 | **0.04** | **42** | 19.6 | 60 | 0.06 | 46 | 13.8 |
| 3col20_5_6 | 40 | 176 | 43 | 0.07 | **40** | 18.3 | 46 | **0.06** | 66 | 7.15 |
| 3col20_5_7 | 40 | 176 | **40** | **0.05** | **40** | 18.5 | 60 | 0.06 | **40** | 9.39 |
| 3col20_5_8 | 40 | 176 | 52 | **0.05** | **40** | 18.6 | 46 | 0.06 | 55 | 8.96 |
| 4col100_9_5 | 200 | 1806 | *time out* | | **1462** | 335 | 1566 | **56.4** | 1512 | 1550 |
| 4col100_9_6 | 200 | 1806 | 1458 | 7030 | **1451** | 351 | 1505 | **79.9** | 1506 | 5222 |
| 4col100_9_7 | 200 | 1806 | 1596 | 3165 | **1461** | 352 | 1472 | **38.7** | *time out* | |
| 4col100_9_8 | 200 | 1806 | 1618 | 6694 | **1455** | 356 | 1510 | **71.5** | 1502 | 2638 |
| 4col100_9_9 | 200 | 1806 | 1556 | 5202 | **1458** | 377 | 1527 | **45.1** | 1484 | 2973 |
| 5cnf...30f4 | 30 | 419 | 316 | 0.65 | **237** | 127 | 369 | **0.13** | 340 | 26.7 |
| 5cnf...40f1 | 40 | 608 | 601 | 0.86 | **397** | 189 | 593 | **0.34** | 418 | 39.5 |
| am_4_4 | 433 | 1458 | 929 | 6.3 | 944 | 25.6 | **902** | **3.95** | 929 | 29.2 |
| am_5_5 | 1076 | 3677 | **2046** | 7614 | 2244 | **62.5** | **2046** | 765 | 2140 | 76.4 |
| ca008 | 130 | 370 | 276 | 0.17 | **255** | 4.83 | 283 | **0.08** | **255** | 1.93 |
| ca016 | 272 | 780 | 584 | 0.72 | **559** | 8.73 | 646 | **0.30** | **559** | 5.33 |
| ca032 | 558 | 1606 | **1176** | **2.66** | 1230 | 14.4 | 1316 | 3.46 | 1281 | 52.9 |
| ca064 | 1132 | 3264 | **2421** | **4.29** | 2793 | 20.2 | 2687 | 22.9 | 2613 | 141 |
| ezfact16_1 | 193 | 1113 | **41** | 0.19 | 169 | 20.4 | 100 | **0.10** | **41** | 383 |
| ezfact16_2 | 193 | 1113 | **47** | **0.07** | 169 | 20.8 | 55 | 0.09 | 41 | 382 |
| gt-012 | 144 | 1398 | 1356 | 7.4 | **1124** | 171 | 1219 | **1.40** | 1205 | 162 |
| gt-014 | 196 | 2289 | 2113 | 53.4 | 1940 | 301 | **1713** | **5.27** | 1989 | 581 |
| hanoi4u | 1312 | 16856 | *time out* | | 7605 | **300** | **6434** | 9670 | *time out* | |
| homer06 | 180 | 830 | **415** | 14.2 | 461 | 50.2 | **415** | 38.1 | **415** | **9.66** |
| homer07 | 198 | 1012 | 506 | 19.2 | 531 | 62.3 | 506 | 38.6 | **415** | **13.9** |
| homer08 | 216 | 1212 | 606 | 39.5 | 650 | 79.0 | 506 | 54.4 | **415** | **20.3** |
| hwb-n20-01 | 134 | 630 | **624** | 1248 | **624** | 93.6 | 627 | **58.8** | **624** | 351 |
| hwb-n20-02 | 134 | 630 | 625 | 1270 | **624** | 142 | 628 | **50.6** | 625 | 508 |
| hwb-n20-03 | 134 | 630 | 626 | 272 | **622** | **58.6** | 626 | 180 | 625 | 191 |
| linvrinv2 | 24 | 61 | 51 | **0.05** | **50** | 6.03 | 51 | **0.05** | 53 | 0.10 |
| linvrinv3 | 90 | 262 | 250 | 0.15 | **239** | 16.9 | 253 | **0.09** | 244 | 0.30 |
| linvrinv4 | 224 | 689 | 689 | 16.4 | **653** | 42.3 | 689 | **4.46** | 657 | 14.2 |
| mm-2x2-5-5-s.1 | 324 | 2064 | **1290** | 136 | 1857 | 152 | 1791 | **40.4** | 2064 | 259 |

Table 1: Experimental evaluation of a constructive implementation

conducted on Pentium IV, 3 Ghz with 1 GB of memory, and for each tested benchmark, a 10,000 seconds timeout has been respected.

First of all, only `AMUSE` cannot succeed to precisely localize in a few seconds the single MUS of the `{23,42}.shuffled` benchmarks, which come from bounded model checking. These ones actually encode formal verification of the open-source Sun PicoJava II microprocessor, and as emphasized in [19], the discovered MUS enables to identify the relevant components of the system for the checked property. We also have performed some experimental tests on various

benchmarks from the graph coloring problem (`{3,4}col-*`). On those CNFs, the finding of a core enables to localize a sub-graph that cannot be colored with the given number of colors, and make the whole problem infeasible. The constructive approach appears to be the best when we consider the size of the returned sub-formula. In addition, this result is obtained in a reasonable time, even if generally `AMUSE` provides a rougher core in a shorter time. As an example, for `4col100_9_8`, `constructMUS` extracts a 1455-clause core in less than 6 minutes whereas `zCore` only localizes a core made of 1618 clauses in about one hour and a half. AMUSE and AOMUS provide sets of 1510 and 1502 clauses in 1 and 23 minutes, respectively. Let us note that on this family of benchmarks, our constructive method is systematically more accurate than the destructive ones.

More generally, our results show that this first implementation of the algorithm `constructMUS` delivers very satisfying results on various CNF, compared to state-of-the-art approaches (see e.g. `hwb-*, gt-*`). However, `AOMUS` appears to be a very good approach for some industrial problems (`homer`), while DPLL-based procedures are more adapted on various families, such that the `am_*_*` one. Consulting the presented results, the developed approach is validated, but let us note that the four tested methods deliver orthogonal results with respect to the considered CNF. Each one of them appears the best appropriate technique for some families of benchmarks, but no one clearly outerperforms the others. Obviously, the fact that the result is here multi-criterion (size of the approximation and run time) is not helpful for the comparison.

In [20], it is suggested to use the notion of *velocity*, which is defined as the ratio between the size of the delivered core and the time needed for the computation. This element enables to know how many clauses are "eliminated" per second, then to know the *best* approach. However, this choice is not completely satisfying. Indeed, let us assume that a CNF formula exhibits 2 MUS that contain a different number of clauses. Is a method which extracts exactly the bigger MUS less efficient that another one which delivers roughly the smaller one (and then, a smaller set of clauses) ?

Deciding the best approach is still intrinsically a problem, that appears to be difficult to deal with objectively without considering any particular application domain. Nevertheless, the behaviour of our implementation appears to be very satisfying for various benchmarks, in comparison to the "best" current approaches. This added to its flexibility (parameter to control the precision, possibility to start with a guess) makes it very interesting as a new general framework for MUS extraction.

## 6 Conclusion

In this paper, an original framework called *constructive* has been presented for the problem of computing an approximated MUS, or *core*. The presented technique is based on a combination of a local search procedure and an exhaustive DPLL-like algorithm. It exhibits interesting features. For instance, it does not need to be able to solve the whole CNF for extracting a core; moreover, the pro-

cedure gives rooms to user's expertise and allows him to guide the search toward a specific core. This is not possible with classical approaches that always compute arbitrary cores. Preliminary experiments show that in many cases, an instance of this framework proves competitive, and can outperform previous approaches. Particularly, our implementation delivers very satisfying results in terms of the size of core.

This first implementation opens many interesting perspectives. The concept of a critical clause, presented in [6], has been proved useful for another local-search-based extractor. This concept cannot be used as such, due to the nature of the approach, but a possible adaptation could improve the accuracy of the delivered core. Other hybridizations of this constructive framework, based for instance on a genetic algorithm for providing sets of models, are also planned as further works. Finally, a combination of a constructive and a destructive approaches could also be explored to combine the best of both worlds.

## References


1. McMillan, K.L., Amla, N.: Automatic abstraction without counterexamples. In: Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03). (2003) 2–17
2. Gupta, A., Ganai, M., Yang, Z., Ashar, P.: Iterative abstraction using SAT-based BMC with proof analysis. In: Proceedings of the IEEE/ACM international conference on Computer-aided design (ICCAD'03), Washington, DC, USA (2003) 416–423
3. Bryant, R.E., Kroening, D., Ouaknine, J., Seshia, S.A., Strichman, O., Brady, B.: Deciding bit-vector arithmetic with abstraction. In: Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07). Volume 4424 of Lecture Notes in Computer Science., Springer (2007) 358–372
4. Oh, Y., Mneimneh, M.N., Andraus, Z.S., Sakallah, K.A., Markov, I.L.: AMUSE: a minimally-unsatisfiable subformula extractor. In: DAC'04: Proceedings of the 41st annual conference on Design automation, New York (USA), ACM Press (2004) 518–523
5. Zhang, L., Malik, S.: Extracting small unsatisfiable cores from unsatisfiable boolean formula. In: Sixth international conference on theory and applications of satisfiability testing (SAT'03), Portofino (Italy) (2003)
6. Gregoire, E., Mazure, B., Piette, C.: Extracting MUSes. In: Proceedings of the 17th European Conference on Artificial Intelligence (ECAI'06), Trento (Italy) (2006) 387–391
7. van Maaren, H., Wieringa, S.: Finding guaranteed MUSes fast. In: International Conference on Theory and Applications of Satisfiability Testing (SAT'08). Volume 4996 of Lecture Notes in Computer Science., Guangzhou (China) (2008) 291–304
8. Hemery, F., Lecoutre, C., Saïs, L., Boussemart, F.: Extracting MUCs from constraint networks. In: Proceedings of the 17th European Conference on Artificial Intelligence (ECAI'06), Trento (Italy) (2006) 113–117
9. Junker, U.: QuickXplain: Preferred explanations and relaxations for over-constrained problems. In: Proceedings of the 19th National Conference on Artificial Intelligence (AAAI'04). (2004) 167–172

10. Marques-Silva, J., Planes, J.: Algorithms for maximum satisfiability using unsatisfiable cores. In: Design, Automation and Test in Europe (DATE'08). (2008) 408–413
11. Papadimitriou, C.H.: Computational Complexity. Addison-Wesley, New York (USA) (1994)
12. Eiter, T., Gottlob, G.: On the complexity of propositional knowledge base revision, updates and counterfactual. Artificial Intelligence **57** (1992) 227–270
13. Chinneck, J.W.: Feasibility and viability. In: Advances in Sensitivity Analysis and Parametric Programming. Volume 6., Kluwer Academic Publishers, International Series in Operations Research and Management Science (1997)
14. de Siqueira, J.L., Puget, J.F.: Explanation-based generalisation of failures. In: Proceedings of the 8th European Conference on Artificial Intelligence. (1988) 339–344
15. Kullmann, O., Lynce, I., Marques Silva, J.P.: Categorisation of clauses in conjunctive normal forms: Minimally unsatisfiable sub-clause-sets and the lean kernel. In: International Conference on Theory and Applications of Satisfiability Testing (SAT'06), Seatle (USA) (2006) 22–35
16. Gregoire, E., Mazure, B., Piette, C.: Boosting a complete technique to find mss and mus thanks to a local search oracle. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07). Volume 2., Hyderabad (India) (January 2007) 2300–2305
17. Hutter, F., Tompkins, D.A.D., Hoos, H.H.: Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In: Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP'02). (2002) 233–248
18. Eén, N., Sorensson, N.: Minisat home page http://www.cs.chalmers.se/cs/research/formalmethods/minisat
19. McMillan, K.L.: Applications of Craig interpolants in model checking. In: Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05). (2005) 1–12
20. Gershman, R., Koifman, M., Strichman, O.: Deriving small unsatisfiable cores with dominators. In: Proceedings of Computer-Aided Verification, Seattle (USA) (2006) 109–122