Computing Minimal Unsatisfiable Subsets of Constraints

Seminar

Author: Shahriar Robbani
Supervision: Erika Ábrahám
RWTH Aachen University, LuFG Informatik 2

WS 16/17

Abstract

We present a procedure as published in [1] which computes all Minimal Unsatisfiable Subsets (MUSes) of unsatisfiable constraint system. Authors have developed a complete algorithm to compute all MUSes and introduced to us two phases, one is generating all Minimal Correction Subsets (MCSes) and another one is generating all MUSes. Removing an MCS makes the unsatisfiable constraint system satisfiable. Also removing any clause from an MUS implies that the remaining set of clauses is not an unsatisfiable subset anymore.

1 Introduction

Many real-world problems have arised in formal verification of hardware and software which can be formulated as constraint satisfaction problems and translated it into Boolean formulas in Conjunctive Normal Form (CNF) (see Section 2.1 for details). Boolean satisfiability (SAT) solvers determine whether a formula is satisfiable or unsatisfiable. When a formula is unsatisfiable, it is required to find unsatisfiable and minimal subsets where each subset is a set of clauses. Minimal Unsatisfiable subsets (MUSes), also called Minimal Unsatisfiable Cores (MUCs), are essential to determine the reasons for the failure of the system. In the past few years, the interest and research on the extraction of MUSes of an unsatisfiable constraint system has been increasing, but most of the research has concentrated on the algorithms to extract a single MUS [1]. This single MUS may not provide complete information about the failure of constraint system, because the system might have multiple MUSes and the appearance of any one makes the system unsatisfiable.
In this paper, we are explaining some algorithms which are necessary to compute all MUSes of a system as in [1]. The paper is organized as follows. Section 2 presents some preliminaries with formal definitions. We introduce the algorithms in details in Section 3 and conclude in Section 4.

2 Preliminaries

2.1 Conjunctive Normal Form

A formula is said to be in Conjunctive Normal Form (CNF) if and only if it is a conjunction of clauses and a clause is a disjunction of literals. A literal could be a positive or negative instance of Boolean variable. A CNF formula φ is defined as follows:

$$\varphi = \bigwedge_{i=1...n} C_i$$

$$C_i = \bigvee_{i=1...k_i} a_{ij}$$

here, C_i is the i^{th} clause and n is the number of total clauses in φ, a_{ij} is a literal and m is the total number of literals in C_i.

Example 1. *Assume the following CNF formula,*

$$\varphi = \underbrace{(x)}_{C_1} \wedge \underbrace{(\neg x)}_{C_2} \wedge \underbrace{(\neg x \vee y)}_{C_3} \wedge \underbrace{(\neg x \vee \neg y)}_{C_4}$$

where, $x, \neg x, y$ and $\neg y$ are the literals.

The aforementioned CNF formula will ne used as an example throughout this paper which is unsatisfiable.

2.2 Satisfiability Checking

Satisfiability checking determines whether the existentially quantified first-order-logic formulas are satisfiable by generating solutions automatically. The formulas are Boolean combinations of theory constraints and the form of theory constraints varies depending on with which theory we want to represent first-order-logic. Some example theory constraints from different theories are given in [2]. There are two types of solvers for satisfiability checking: 1) Boolean satisfiability (SAT) solver and 2) SAT-modulo-theories (SMT) solver.

A SAT solver solves the Boolean satisfiability problem (the problem of determining if there exists any satisfying soultion for a given Boolean formula) by implementing decision procedure. The Davis–Putnam–Logemann–Loveland (DPLL) algorithm is the basis for most modern SAT solvers. The input formula of it is expected to be in CNF.

Example: Let us consider a CNF, $(x) \wedge (\neg x \vee \neg y) \wedge (z)$. Boolean constraint propagation (BCP) (determines the assignment of next variable implied by the last decision) starts with x as it is a single literal and x is set to *true*. Then, BCP indicates that y of the second clause must be $false$ to satisfy the CNF. There is still a unassigned variables. A new decision will be made and z is set to *true*. As all variables are assigned and there is no conflict, the satisfying soluiton is $x = 1, y = 0$ and $z = 1$.

A SMT solver can be applied on quantifier-free first-order-logic formulas with an underlying theory to check the satisfiability. There are two type of SMT solvers, Eager SMT solver and Lazy SMT solver. To know about SMT solver in details you can read [3].

2.3 Clause-Selector Variables

Authors of the paper [1] has used clause-selector variable by augmenting a CNF formula. Clause-selector variable is a variable, w_i which is negated to augment each clause of C_i of a CNF such that $C'_i = (\neg w_i) \vee C_i$ in a new formula φ'. Notice that each C'_i is an implication, $C'_i = (\neg w_i \rightarrow C_i)$. It means if w_i is set to *true*, the original clause C_i is being enabled. Conversely, assigning w_i $false$ means C_i is being disabled or removed from the

Table 2.1 All MUSes and MCSes

$MUSes(\varphi)$	$\{C_1, C_2\}, \{C_1, C_3, C_4\}$
$MCSes(\varphi)$	$\{C_1\}, \{C_2, C_3, C_4\}, \{C_2, C_4\}$

set of clauses, because C_i' is satisfied by the assignment of w_i. So, adding clause-selector variables provides the SAT solver the ability to enable and disable clauses as a part of its normal search.

For our example formula $\varphi = (x) \wedge (\neg x) \wedge (\neg x \vee y) \wedge (\neg x \vee \neg y)$, after adding clause-selector variables we get the following augmented formula φ',

$$\varphi' = (\neg w_1 \vee x) \wedge (\neg w_2 \vee \neg x) \wedge (\neg w_3 \vee \neg x \vee y) \wedge (\neg w_4 \vee \neg x \vee \neg y)$$

2.4 Minimal Unsatisfiable Subset and Minimal Correction Subset

An Unsatisfiable Subset (US) ia a subset of the clauses of an unsatisfiable CNF formula φ where the conjunction of clauses is unsatisfiable. US is said to be Minimal Unsatisfiable Subset (MUS) if and only if removing any clause from UC makes it satisfiable. A Minimal Correction Subset (MCS) is a subset of the clauses of φ whose removal from φ make it satisfiable and which is minimal. A formula can have multiple MUSes and MCSes. For any formula φ, MUSes and MCSes of φ can be represented as MUSes(φ) and MCSes(φ), respectively. The formal definitions of the set of MUSes and MCSes of φ are given in the following: Defintion: A subset U be a MUS which is a subset of φ if U is unsatisfiable and for each clause C_i containing in U, the removal of C_i from U is satisfiable.

$$MUSes(\varphi) \quad = \quad \begin{cases} U \subseteq \varphi & \text{, U is unsatisfiable} \\ \\ \forall C_i \varepsilon U & \text{, U } \backslash \{C_i\} \text{ is satisfiable} \end{cases}$$

Defintion: A subset M be a MCS which is a subset of φ if the removal of M from φ is satisfiable and for each clause C_i containing in M the removal of M from φ is unsatisfiable where M does not have C_i.

$$MUSes(\varphi) \quad = \quad \begin{cases} M \subseteq \varphi & , \varphi \backslash M \text{ is satisfiable} \\ \\ \forall C_i \varepsilon M & , C \backslash (M \backslash \{C_i\}) \text{ is unsatisfiable} \end{cases}$$

For our example formula $\varphi = (x) \wedge (\neg x) \wedge (\neg x \vee y) \wedge (\neg x \vee \neg y)$, the MUSes and MCSes are shown in the Table 2.1.

2.5 Hitting Sets

Assume, Ω is a collection of sets of a finite set D. A hitting set of Ω is a subset of D such that it contains at least one element from each subset in Ω. Formally, the definition is given below:

Definition: A hitting set H of Ω is $H \subseteq D$ such that $\forall S \in \Omega$, $H \cap S \neq 0$ where S represents each subset of Ω.

In the paper [1], authors refer to minimal hitting sets which are the hitting sets from which no elements cannot be removed and if at least one element is removed, it will not be a hitting set anymore.

Example 2. *Let us consider,* $\Omega = \{\{a, b\}, \{b, c, d\}\}$. *Then,* $\{a, b\}, \{b, c, d\}, \{a, c, d\}, \{b\}, \cdots$ *are the hitting sets where,* $\{b\}$ *is the only minimal hitting set.*

2.6 MUS \ MCS Duality

There is a relationship between MUSes and MCSes which is the foundation of paper [1]. The relationship states that the set of MUSes of formula φ is equal to the set of minimal and irreducible hitting sets of the set of MCSes and vice-versa. This is the duality of MUS and MCS. Formally, it can said that:

1. A subset U of φ is an MUS if and only if U is an irreducible hitting set of MCSes(φ).

2. A subset M of φ is an MUS if and only if M is an irreducible hitting set of MUSes(φ).

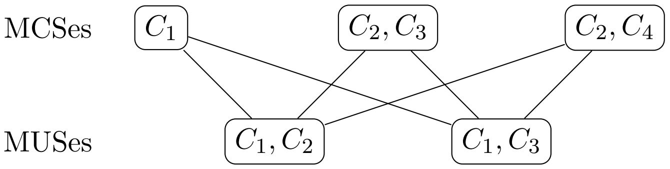

Figure 1: Duality of MUSes and MCSes.

To show duality we use our example formula $\varphi = (x) \wedge (\neg x) \wedge (\neg x \vee y) \wedge (\neg x \vee \neg y)$. First, take the set MCSes of φ and it is $\{\{C_1\}, \{C_2, C_3\}, \{C_2, C_4\}\}$. For this, the set of minimal and irreducible hitting sets is $\{\{C_1, C_2\}, \{C_1, C_{3,}C_4\}\}$ which is exactly like the set of MUSes (given in Table 1). Similarly, we can show that the set of MCSes is the set of irreducible hitting sets of the set of MUSes. So, every MCS has at least one clause from every MUS and vice-versa. The duality is illustrated in Figure(1).

2.7 AtMost Constraints

AtMost Constraint is a type of counting clause that can be generated from many types of clauses. If $\{l_1, l_2, l_3, \cdots, l_\mathrm{n}\}$ is a set of n literals and k is a positive integer, an AtMost constraint is defines as,

$$AtMost(\{l_1, l_2, \ldots, l_n\}, k) \equiv \sum_{i=1}^{n} val(l_i) \leqslant k$$

where, $val(l_\mathrm{i})$ is 1 if l_i is assigned $true$ and otherwise 0. k is a bound which tells that maximum how many literals can be assigned $true$.
Authors used an implementation of the AtMost constraints where the variables are watched and propagates the negation of each literal when k of them are assigned $true$. This is implemented with exactly n watched literals and a counter that is incremented or decremented whenever one of them is assigned or unassigned.
Example:Let us consider we have three literals, l_1, l_2,l_3 and $k = 2$. If we call AtMost, we get a clause, $C_{AB} = (l_1 \vee l_2 \vee l_3)$. Here, to make C_{AB} satisfiable exactly two variables can be assigned $true$ as bound is 2. After that the third variable will be assigned $false$. So, the satisfying soluiton could be ($l_1 = true, l_2 = true$ and $l_3 = false$) or ($l_1 = true, l_3 = true$ and $l_2 = false$) or ($l_2 = true, l_3 = true$ and $l_1 = false$).

3 Algorithms for Computing Minimal Unsatisfiable Subset

The approach of computing all MUSes of φ is to first find all MCSes(φ) and then to compute all irreducible hitting sets of MCSes(φ), which are all MUSes(φ). So, there are two phases to generate all MUSes of a unsatisfiable CNF formula. The first phase is to compute MCSes by using an algorithm and the second phase is to compute MUSes from the MCSes by using a recursive algorithm which authors have developed to compute irreducible hitting sets.

In the first phase, authors use a SAT solver (produces an satisfying assignment, if exists) to work with the input given and generate hitting sets of MUSes (MCSes) without revealing the underlying MUSes. For the second phase, they get all the information with MCSes and use an recursive algorithm to compute minimal and irreducible hitting sets of MCSes (MUSes).

Algorithm 1 Algorithm for finding all MCSes of a formula φ

```
MCSes ( $\varphi$ )
 1  $\varphi' \leftarrow AddYVars(\varphi)$
 2  $MCSes \leftarrow \emptyset$
 3  $k \leftarrow 1$
 4  while  SAT($\varphi$)
 5  do $\varphi'_k \leftarrow \varphi' \wedge AtMost(\{\neg w_1, \neg w_2, \ldots, \neg w_n\}, k)$
 6      while $newMCS \leftarrow IncrementalSAT(\varphi'_k)$
 7      do $MCSes \leftarrow MCSes \cup newMCS$
 8          $\varphi'_k \leftarrow \varphi'_k \wedge BlockingClause(newMCS)$
 9          $\varphi' \leftarrow \varphi' \wedge BlockingClause(newMCS)$
10
11      $k \leftarrow k + 1$
12
13  return $MCSes$
```

3.1 Computing MCSes

In Algorithm 1, there is a method named **MCSes** which takes a CNF formula φ as input. The goal of this is to find minimal set of clauses whose removal makes φ satisfiable. In Line 1, φ is augmented with clause-selector variables by creating φ'. The augmentation increases the number of variables. Also the search space grows for finding a satisfying assignment. Initially, the variable, ***MCSes*** is empty and at the end we will get all MCSes in this variable. There is a counter variable, K which is initialized by 1.

Each iteration of the outer while loop (Lines 4-12) finds an MCS of size K, which is incremented by 1 after each iteration. In Line 5, a clause is added of the form $AtMost(\{l_1, l_2, \ldots, l_n\}, k)$ to φ' and a new formula, φ'_{k} is created. K number of variables, w_i are assigned to $false$ so that K number of clauses could be disabled. The set of variables, w_i are assigned to $false$ by indicating the clauses as an MCS.

The inner while loop (Lines 6-10) searches all the satisfiable assignments of φ'_{k} and also finds all MCSes of size k. In Line 6, **IncrementalSAT** method is called which uses MiniSAT (a modern open-source SAT solver which has incremental solving ability) for finding a solution of φ'_{k}. In this method the solving function is invoked multiple times, but each time with a different set of assumption literals and then the solver checks the satisfiability of all the clauses provided with the current assumptions only [4]. Each satisfying assignment produces an MCS stored in a variable ***newMCS*** which contains the clauses having w_{i} assigned $false$. Each ***newMCS*** is recorded in ***MCSes*** (Line 7). In Line 8 and 9, a blocking clause is added to φ'_k and φ', respectively to block that solution. Blocking clause is a clause containing the clause-selector variables which consist in the clauses of that MCS. It means at least one of the clauses in the MCS will be enabled for future solution. For example, an MCS has the clauses C_2 and C_3 which means w_2 and w_3 are assigned false in satisfying assignment. Then the blocking clause will be $(w_2 \lor w_3)$. $True$ is assigned to at least one w_i excluding the MCS and any of its supersets from any future solution.

All founded MCSes are irreducible as the MCSes were found in increasing size and all supersets are excluded form the future solutions. After evaluating all the solutions with bound K enforces to find solutions with bound $K+1$. Also, $K+1$ disabled clause to be irreducible as potential subsets would have found and blocked.

The outer while loop checks if φ' is still satisfiable without any bound on w_{i}. Note that, φ' is augmented with all generated blocking clauses. Finding no satisfying assignments indicates that we have found all MCSes and there is no other way to remove clauses for satisfying formula. Finally, the algorithm terminates and returns all MCSes to caller.

Example 3. *Let us consider, our example formula* $\varphi = (x) \land (\neg x) \land (\neg x \lor y) \land (\neg x \lor \neg y)$. *So,* $\varphi' = (\neg w_1 \lor x) \land (\neg w_2 \lor \neg x) \land (\neg w_3 \lor \neg x \lor y) \land (\neg w_4 \lor \neg x \lor \neg y)$. *Initially,* $MCSes = \emptyset$ *and* $k = 1$.

In the first iteration, it will add an AtMost bound with $K = 1$ *by creating* $\varphi'_1 = \varphi' \land AtMost(\{l_1, l_2, l_3, l_4\}, 1)$. *At first* $false$ *is assigned to* w_1 *and the incremental solver find a solution for* φ'_k. *Now, we get the followings:*

$$MCSes = \{C_1\}$$

$$\varphi'_1 = \varphi'_1 \land w_1$$
$$\varphi' = \varphi' \land w_1$$

After adding the blocking clause, (w_1) *the incremental solver will be unable to find any further solution with* $K = 1$. *There is no other way to remove one clause to satisfy* φ. *It means no other single clause covers all of its MUSes. The inner while loop is exited.*

As φ' *is still satisfiable, the bound is incremented to 2 and the search for soluitons of* $\varphi'_2 = \varphi' \land AtMost(\{l_1, l_2, l_3, l_4\}, 2)$ *is continued.For* $K = 2$, *we get the followings, respectively:*

- $w_2 = false$ *and* $w_3 = false$,

$$MCSes = \{\{C_1\}, \{C_2, C_3\}\}$$

$$\varphi'_2 = \varphi'_2 \land (w_2 \lor w_3)$$
$$\varphi' = \varphi' \land (w_2 \lor w_3)$$

- $w_2 = false$ *and* $w_4 = false$,

$$MCSes = \{\{C_1\}, \{C_2, C_3\}, \{C_2, C_4\}\}$$

$$\varphi'_2 = \varphi'_2 \land (w_2 \lor w_4)$$
$$\varphi' = \varphi' \land (w_2 \lor w_4)$$

We will not get any other MCSes for $K = 2$. *Also, the outer while loop will exit as* φ' *with all the added blocking clauses is no longer satisfiable. So, final MCSes are,* $\{C_1\}, \{C_2, C_3\}$ *and* $\{C_2, C_4\}$.

Table 3.1 My caption

Step No.		k	φ'	MCSes
1	Initialization	1	A	E
2	Run1		B	F
3	Run2	2	C	G
4	Run3		D	H

3.2 Compuitng MUSes

Once the entire collection of MCSes has been computed, the second phase produces all MUSes of the given instances and this phase is independent of first phase as it does not depend on the semantics of the input of Algorithm 1. The second phase can be applied to any minimal hitting set. Authors presented a recursive algorithm which takes two inputs, one is set of MCSes and another the MUS currently being constructed in each branch of the recursion. The outputs are all MUSes.

Algorithm 2 is the recursive algorithm where the inputs are ***MCSes*** and ***currentMUS***.

Algorithm 2 Algorithm for computing the complete set of MUSes from a set of MCSes

```
ALLMUSES(MCSES, CURRENTMUS)($\varphi$)
 1  $' \leftarrow AddYVars(\varphi)$
 2  if a condition being True or False
 3    then point 1
 4
 5    else if another condition
 6           then point 2
 7
 8    else point 3
 9         $MCSes \leftarrow \emptyset$
10  $k \leftarrow 1$
11  while SAT($\varphi$)
12  do $\varphi'_k \leftarrow \varphi' \wedge AtMost(\{\neg w_1, \neg w_2, \ldots, \neg w_n\}, k)$
13     while $newMCS \leftarrow IncrementalSAT(\varphi'_k)$
14     do $MCSes \leftarrow MCSes \cup newMCS$
15        $\varphi'_k \leftarrow \varphi'_k \wedge BlockingClause(newMCS)$
16        $\varphi' \leftarrow \varphi' \wedge BlockingClause(newMCS)$
17
18     $k \leftarrow k + 1$
19
20  return $MCSes$
```

The goal of this is to find a irreducible set of clauses that hits each set of ***MCSes***. The terminating condition of the recursion is on Line 1 that is if ***MCSes*** is empty. If it is,prints the MUS generated in current recursion branch and returns to look for other MUSes in another branches on Line 2 and in Line 3,respectively. Lines $5 - 12$ is iterated through all possible choices of clause selected on Line 5 and recorded in a variable, ***selClause***. In Line 6, ***selClause*** is stored in a growing MUS, ***newMUS***. For all possible choices of MCSes (selected on line 7) recorded in ***selMCS*** such that ***selClause*** consists in ***selMCS***,

iteration occurs from Lines $7-11$. Then, ***MCSes*** is recorded in a variable, ***newMCSes***. For every ***selMCS*** a method, **PropagateChoice** (Algorithm 3) is called with ***newMCSes***, ***selClause*** and ***selMCS***. Algorithm 3 has the goal to alter a copy of the current MCSes (***newMCSes***). It alters by removing an MCS from ***MCSes*** hitted by ***thisClause***. In the last line, **MaintainNoSupersets** is invoked which removes any MCS from ***MCSes*** which is a superset of others. This is needed as we are dealing with the minimal set of MCSes and no MCS cannot be a superset of others. For example, let we have $MCSes = \{\{C_2\}\{C_1, C_3\}\{C_2, C_4\}\}$. Here, $\{C_2, C_4\}$ is a superset of $\{C_2\}$. If we pass this MCSes through **MaintainNoSupersets**, $\{C_2, C_4\}$ will be removed.
The recursion of Algorithm 2 continues by calling **AllMUSes** with ***newMCSes*** and ***newMUS***.

Example 4. *Let us consider, our example formula φ. We already have found $MCSes(\varphi) = \{\{C_1\}, \{C_2, C_3\}, \{C_2, C_4\}\}$. Let, C_1 is chosen as* ***selClause*** *in the first iteration. So, C_1 is added in the growing MUS. Now, for each set of MCSes another iteration will occur which calls* ***PropagateChoice*** *method by creating a modified copy of current MCSes and makes a recursive call to itself. Modified MCSes contain $\{C_2, C_3\}\{C_2, C_4\}$. The recursive call continues until empty MCSes is not found. Empty MCSes is found by producing a MUS, $\{C_2, C_3\}$. Now, it will look for other MUSes in other branches. For our example the whole procedure to generate all MUSes is illustrated in the Figure 1. Finally, we get all MUSes, $\{C_1, C_2\}$ and $\{C_1, C_3, C_4\}$.*

4 Conclusion

We have discussed a set of algorithms which generate MUSes and also authors made a framework consisting of the algorithms. The generation of MUSes is based on the duality of MCSes and MUSes. The algorithms can be used to work with different types of constraint solvers. Authors have implemented the algorithms on a solver for Disjunctive Temporal Problems (DTPs) [26] and using YICES [14], an efficient Satisfiability Modulo Theories (SMT) solver which covers a wide range of constraint types [1]. That's why, the algorithms (may be with small modification) can be used on top of new constraint solvers to provide the functionality of finding MUSes for new types of constraints.

References

[1] V. Weispfenning, *Quantifier elimination for real algebra - the quadratic case and beyond.* Appl. Algebra Eng. Commun. Comput, 1997.

[2] R. Loss, V. Weispfenning, *Applying linear quantifier elimination.* The computer Journal 36 (1993), pp. 450-462.