Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints

Mark H. Liffiton and Karem A. Sakallah
Department of Electrical Engineering and Computer Science,
University of Michigan, Ann Arbor 48109-2121
({liffiton,karem}@eecs.umich.edu*)*

Abstract. Much research in the area of constraint processing has recently been focused on extracting small unsatisfiable "cores" from unsatisfiable constraint systems with the goal of finding *minimal* unsatisfiable subsets (MUSes). While most techniques have provided ways to find an approximation of an MUS (not necessarily minimal), we have developed a sound and complete algorithm for producing *all* MUSes of an unsatisfiable constraint system. In this paper, we describe a useful relationship between satisfiable and unsatisfiable subsets of constraints that we subsequently use as the foundation for MUS extraction algorithms, implemented for Boolean satisfiability constraints. The algorithms provide a framework with which many related subproblems can be solved, including relaxations of completeness to handle intractable instances, and we develop several variations of the basic algorithms to illustrate this. Experimental results demonstrate the performance of our algorithms, showing how the base algorithms run quickly on many instances, while the variations are valuable for producing results on instances whose complete results are intractably large. Furthermore, our algorithms are shown to perform better than the existing algorithms for solving either of the two distinct phases of our approach.

1. Introduction

While a great deal of research has been done on solving constraint satisfaction problems (CSPs), far less attention has been paid to understanding an instance when it is unsatisfiable. A large portion of artificial intelligence research has been devoted to techniques for deciding whether a given CSP is satisfiable, almost all presenting a short proof in the form of a satisfying assignment when it is. But because deciding a general CSP is in the complexity class NP, no such short proof or other explanation is known for *un*satisfiable instances. On unsatisfiable instances, most available solvers provide nothing more than a one word response: "Unsatisfiable."

In the past few years, there has been an upswing of interest and research in one particular mechanism for providing information beyond that opaque response: extraction of Minimal Unsatisfiable Subsets of constraints (MUSes), also called "unsatisfiable cores." Given an unsatisfiable system of constraints C, an MUS of C is a subset of those constraints that is (1) unsatisfiable and (2) minimal, in the sense that

© 2007 *Kluwer Academic Publishers. Printed in the Netherlands.*

removing any one of its elements makes the remaining set of constraints satisfiable. MUSes can be seen as compact, irreducible explanations of a CSP's infeasibility.

Most experimental research on MUSes has focused on algorithms for extracting a single unsatisfiable subset or "core," often not even minimal, from infeasible constraint systems (see Section 9 for details). But a minimal core (a single MUS) may not provide complete information about a constraint system's infeasibility. Unsatisfiable systems of constraints often contain many MUSes, and the presence of any one makes the system unsatisfiable. Finding a single MUS is like pointing to a single hole to explain why a sieve doesn't hold water, and "plugging" a single MUS (e.g., by relaxing constraints in the MUS to make it satisfiable) will not necessarily affect other MUSes, leaving the instance infeasible. Finding all MUSes has found practical application for various constraint types in formal verification systems, including both hardware [2, 1] and software [4] verification. In the former, MUSes are used to refine an abstraction of a hardware design, and finding all MUSes is necessary to produce the best refinement. In the latter, the MUSes are explanations of type errors, and again all MUSes are required to generate the best explanation of the errors.

We have developed sound and complete algorithms for computing *all* MUSes of an unsatisfiable constraint system, dubbed CAMUS (**C**ompute **A**ll **M**inimal **U**nsatisfiable **S**ubsets - pronounced "ka-**moo**" after the French writer), as well as an implementation for Boolean satisfiability instances. In this paper, we present and explain the theoretical basis of our work, algorithms for finding all MUSes of an infeasible constraint system, and empirical results showing the performance of CAMUS on a variety of Boolean satisfiability instances.

Our algorithms provide a general framework on which different systems for finding MUSes can be built with small modifications. For example, because the problem of finding all MUSes is generally intractable, due to the potentially exponential size of the result, we have developed a modification of CAMUS that relaxes the completeness criterion to return several, but not all, MUSes. We discuss this and another modification to illustrate the flexibility of our algorithms and of the underlying theory.

The remainder of this paper is organized as follows. Section 2 lays out some background with formal definitions and introduces an example formula. The fundamental concept underlying our work is developed in Section 3. We introduce our approach to finding all MUSes in Section 4, followed by details of the algorithms that make up CAMUS and several variations in Sections 5, 6, and 7. Experimental results are

presented in Section 8. Finally, we discuss related work in Section 9 and conclude in Section 10.

2. Background

2.1. Boolean Satisfiability and CNF

While the ideas in this work are applicable to any type of constraint system, in this paper we use Boolean satisfiability constraints to describe and explain our ideas and algorithms (we have also implemented related systems for finding all MUSes of Disjunctive Temporal Problems (DTPs) [26] and Satisfiability Modulo Theories (SMT) instances [2]). Boolean satisfiability instances are generally expressed as a conjunction of disjunctions called Conjunctive Normal Form (CNF): Formally, a CNF formula φ is defined as follows:

$$\varphi = \bigwedge_{i=1...n} C_i$$
$$C_i = \bigvee_{j=1...k_i} a_{ij}$$

where each *literal* a_{ij} is either a positive or negative instance of some Boolean variable (e.g., x_3 or $\neg x_3$), k_i is the number of literals in the *clause* C_i (a disjunction of literals), and n is the number of clauses in the formula. In more general terms, each clause is a *constraint* of the constraint system φ. A CNF instance is said to be *satisfiable* if there exists some assignment to its variables that makes the formula evaluate to TRUE, otherwise it is *unsatisfiable*. The problem of deciding whether a given CNF instance is satisfiable is the canonical NP-Complete problem to which many other constraint satisfaction problems can be polynomially reduced.

The following unsatisfiable CNF instance will be used as an example throughout this paper. We will refer to individual clauses as C_i, where i refers to the position of the clause in the formula (e.g., $C_3 = (\neg x_1 \vee x_2)$).

	C_1		C_2		C_3		C_4		C_5		C_6
$\varphi =$	(x_1)	$\wedge$	$(\neg x_1)$	$\wedge$	$(\neg x_1 \vee x_2)$	$\wedge$	$(\neg x_2)$	$\wedge$	$(\neg x_1 \vee x_3)$	$\wedge$	$(\neg x_3)$

2.2. Clause-Selector Variables

Some of our algorithms use *clause-selector variables*, which can be used to augment a CNF formula in such a way that standard Boolean satisfiability (SAT) solvers can manipulate and, in effect, reason about the formula's clauses without any modification to the solver itself. This type of augmentation has also been used in other algorithms such as those in [28] and [30].

Every clause C_i in a CNF formula φ is augmented with a negated clause-selector variable y_i to give $C_i' = (\neg y_i \vee C_i)$ in a new formula φ'. Notice that each C_i' is an implication, $C_i' = (y_i \rightarrow C_i)$. Assigning a particular y_i the value TRUE implies the original clause, essentially enabling it. Conversely, assigning y_i FALSE has the effect of disabling or removing C_i from the set of constraints, as the augmented clause C_i' is satisfied by the assignment to y_i. This change gives a standard, unmodified SAT solver the ability to enable and disable constraints as part of its normal search, checking the satisfiability of the enabled subsets of constraints within a single backtracking search tree.

For our example formula, the augmented formula φ' created by adding these clause-selector variables is:

$$\begin{aligned}\varphi' =&(\neg y_1 \vee x_1) \wedge (\neg y_2 \vee \neg x_1) \wedge (\neg y_3 \vee \neg x_1 \vee x_2) \wedge \\ &(\neg y_4 \vee \neg x_2) \wedge (\neg y_5 \vee \neg x_1 \vee x_3) \wedge (\neg y_6 \vee \neg x_3)\end{aligned}$$

2.3. AtMost Constraints

One of our algorithms employs *AtMost* constraints, a type of counting constraint that can be constructed from many types of constraints or added to a constraint solver with few modifications. Given a set of n literals $\{l_1, l_2, \ldots, l_n\}$ and a positive integer k, an AtMost constraint is defined as

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

where $\text{val}(l_i)$ is 1 if l_i is assigned TRUE and 0 otherwise. This constraint places an upper bound on the number of literals in the set assigned TRUE.

Though this constraint can be encoded into Boolean CNF using encodings such as in [33], the encodings are unnecessarily complex. In practice (and in the solver we use specifically), an efficient implementation of the AtMost constraint "watches" the assignments to the included variables and immediately propagates the negation of each

remaining literal once k of them have been 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. The encodings presented in [33] require the use of additional variables and either $O(n \cdot k)$ or $7n - 3\lfloor \log n \rfloor - 6$ additional clauses. In modern SAT solvers that use watched literals, each additional clause adds two "watches," and the trigger action of each is more complex than a simple counter increment. A CNF encoding would of course be needed if CAMUS were implemented on a closed SAT solver that did not allow implementing the AtMost constraint internally.

2.4. Hitting Sets

Given a collection Ω of sets from some finite domain D, a *hitting set* of Ω is a set of elements from D that "hits" every set in Ω by having at least one element in common with it. Formally:

Definition 1. A hitting set H of Ω is $H \subseteq D$ such that $\forall S \in \Omega,\ H \cap S \neq \emptyset$.

In this paper, we refer to *minimal* or *irreducible* hitting sets, which are hitting sets from which no element can be removed without losing the property of being a hitting set. For example, $\{1, 2, 3\}$ and $\{1, 2\}$ are both hitting sets of $\{\{1, 4\}, \{2, 3, 4\}\}$, but only the latter is irreducible. Note that we are not necessarily referring to the smallest hitting set, which for this example is $\{4\}$.

The problem of deciding whether a given collection of sets has a hitting set of size k or smaller is NP-Complete [23]. This makes the problem of finding the smallest hitting set of a collection NP-Hard, but it does not place any such complexity bounds on the problem of finding a minimal hitting set.

The *hypergraph transversal* problem [16] is equivalent to the hitting set problem. Given a hypergraph $G = (V, E)$, a transversal of G is a set of vertices in V that touches every edge in E. (The equivalence to hitting sets equates the domain with V and the collection of sets with E.) In this paper, we will mostly use the hitting set terminology because sets give a more natural representation for our problems than hypergraphs do. The main exception to this is in the related work section; most work on the problem has been done in the context of hypergraph transversals.

2.5. Useful Subsets of Constraint Systems

The definition of a *Minimal Unsatisfiable Subset* of constraints (MUS) is fundamental to this work, as is the closely related concept of a *Minimal Correction Subset* (MCS). As mentioned earlier, an MUS is a subset of the constraints of an infeasible constraint system which is both unsatisfiable and cannot be made smaller without becoming satisfiable. An MCS is a subset of the constraints of an infeasible constraint system whose removal from that system results in a satisfiable set of constraints ("correcting" the infeasibility) and which is minimal in the same sense that any proper subset does not have that defining property. Any infeasible constraint system can have multiple MUSes and MCSes, potentially exponential in the number of constraints. Formally, given an unsatisfiable constraint system C, its MUSes and MCSes are defined as follows:

Definition 2. A subset $U \subseteq C$ is an MUS if U is unsatisfiable and $\forall C_i \in U,\ U \backslash \{C_i\}$ is satisfiable.

Definition 3. A subset $M \subseteq C$ is an MCS if $C \backslash M$ is satisfiable and $\forall C_i \in M,\ C \backslash (M \backslash \{C_i\})$ is unsatisfiable.

To aid understanding, we also introduce the *Maximal Satisfiable Subset* (MSS), defined similarly:

Definition 4. A subset $S \subseteq C$ is an MSS if S is satisfiable and $\forall C_i \in (C \backslash S),\ S \cup \{C_i\}$ is unsatisfiable.

The MSS is a generalization of a solution to the well-known MaxSAT (or MaxCSP) problem, which is concerned with maximum cardinality satisfiable subsets. MSSes are defined in terms of *inaugmentable* subsets of constraints; the largest satisfiable subset is a special case of this. Thus, any MaxSAT/MaxCSP solution is an MSS, but the converse is not necessarily true.

The MUSes, MCSes, and MSSes of our example formula are shown here (in general, we will use MUSes(C), MCSes(C), etc. to refer to the complete collections of each subset type for any given constraint system C):

MUSes(φ)	Clauses (for reference)
$\{C_1, C_2\}$	$\{(x_1), (\neg x_1)\}$
$\{C_1, C_3, C_4\}$	$\{(x_1), (\neg x_1 \vee x_2), (\neg x_2)\}$
$\{C_1, C_5, C_6\}$	$\{(x_1), (\neg x_1 \vee x_3), (\neg x_3)\}$

MCSes(φ)	MSSes(φ)
$\{C_1\}$	$\{C_2, C_3, C_4, C_5, C_6\}$
$\{C_2, C_3, C_5\}$	$\{C_1, C_4, C_6\}$
$\{C_2, C_3, C_6\}$	$\{C_1, C_4, C_5\}$
$\{C_2, C_4, C_5\}$	$\{C_1, C_3, C_6\}$
$\{C_2, C_4, C_6\}$	$\{C_1, C_3, C_5\}$

Note that any MCS is the complement of some MSS, and vice versa[1]. Removing an MCS from an infeasible constraint system "corrects" the system by turning it into an MSS.

3. MUS/MCS Duality

Recently, an important connection between MUSes and MCSes has been noted independently by Bailey and Stuckey [4], Birnbaum and Lozinskii [6], and Liffiton, et. al. [26]. This relationship is the foundation of the work we present in this paper. We describe here the relationship and a general approach for finding all MUSes of a constraint system that follows from it.

This relationship can be stated simply: The set of MUSes of a constraint system C and the set of MCSes of C are "hitting set duals" of one another. The set of MUSes is equivalent to the set of all irreducible hitting sets of the MCSes, and the MCSes are likewise all the irreducible hitting sets of the MUSes. This is stated formally in the following theorem, whose proof appears in [6] as Theorem 4.5 (c) and (d). We provide a more intuitive explanation and an example in this section.

Theorem 1. *Given an unsatisfiable constraint system C:*

1. *A subset M of C is an MCS of C iff M is an irreducible hitting set of MUSes(C);*
2. *A subset U of C is an MUS of C iff U is an irreducible hitting set of MCSes(C).*

Recall that the presence of any MUS in a constraint system C makes C infeasible. By nature of its minimality, an MUS can be made satisfiable by removing any one constraint from it. Therefore, one way to make C feasible is to "neutralize" its MUSes by removing at least one constraint from each. An MCS of C provides a set of constraints

[1] In [26, 27], we referred to MCSes as "CoMSSes," the complements of MSSes; we use the name MCS now because it is simpler and more descriptive.

$$\begin{array}{rccccccccccc} & C_1 & & C_2 & & C_3 & & C_4 & & C_5 & & C_6 \\ \varphi = & (x_1) & \wedge & (\neg x_1) & \wedge & (\neg x_1 \vee x_2) & \wedge & (\neg x_2) & \wedge & (\neg x_1 \vee x_3) & \wedge & (\neg x_3) \end{array}$$

MCSes(φ)	C_1	C_2	C_3	C_4	C_5	C_6
$\{C_1\}$	X					
$\{C_2, C_3, C_5\}$		X	X		X	
$\{C_2, C_3, C_6\}$		X	X			X
$\{C_2, C_4, C_5\}$		X		X	X	
$\{C_2, C_4, C_6\}$		X		X		X

$$\begin{aligned} \text{MUSes}(\varphi) &= (C_1)(C_2 \vee C_3 \vee C_5)(C_2 \vee C_3 \vee C_6)(C_2 \vee C_4 \vee C_5)(C_2 \vee C_4 \vee C_6) \\ &= C_1C_2 \vee C_1C_3C_4 \vee C_1C_5C_6 \\ &= \{\{C_1, C_2\}, \{C_1, C_3, C_4\}, \{C_1, C_5, C_6\}\} \end{aligned}$$

MUSes(φ)	C_1	C_2	C_3	C_4	C_5	C_6
$\{C_1, C_2\}$	X	X				
$\{C_1, C_3, C_4\}$	X		X	X		
$\{C_1, C_5, C_6\}$	X				X	X

$$\begin{aligned} \text{MCSes}(\varphi) &= (C_1 \vee C_2)(C_1 \vee C_3 \vee C_4)(C_1 \vee C_5 \vee C_6) \\ &= C_1 \vee C_2C_3C_5 \vee C_2C_3C_6 \vee C_2C_4C_5 \vee C_2C_4C_6 \\ &= \{\{C_1\}, \{C_2, C_3, C_5\}, \{C_2, C_3, C_6\}, \{C_2, C_4, C_5\}, \{C_2, C_4, C_6\}\} \end{aligned}$$

Figure 1. Covering Problems Linking MCSes(φ) and MUSes(φ)

whose removal will accomplish this: an MCS M is an irreducible set of constraints whose removal makes C satisfiable. Thus, every MCS contains at least one constraint from every MUS of C. So almost directly from the definition of MCS we can see that MCSes and minimal hitting sets of the MUSes are equivalent: both are minimal sets of constraints whose removal makes C satisfiable. A similar argument goes the other way to show that MUSes are irreducible hitting sets of the MCSes, but it is not as intuitive.

This relationship is depicted in Figure 1 for our example formula. The first table corresponds to the problem of finding hitting sets of the MCSes to generate MUSes(φ), while the second table corresponds to the dual problem of finding hitting sets of the MUSes to generate

MCSes(φ). In the first table, each column corresponds to a clause from the formula, and each row represents a single MCS. We say that a clause "covers" an MCS (marked with an 'X' in that row) if it is contained in the MCS. Each MUS is then an irreducible subset of the columns that covers all of the rows. The table below represents the MUSes in the same fashion, and every MCS is an irreducible subset of the columns that covers all of *its* rows. Underneath each table, we show how the MUSes can be found from the table of MCSes (and the MCSes from the table of MUSes) in a straightforward, though computationally inefficient, manner: each row becomes a disjunction of the columns that cover that row, and the disjunctions are multiplied out and simplified by removing subsumed terms to produce the minimal hitting sets.

4. Our Approach

In practice, it is easier to find satisfiable subsets of constraints than unsatisfiable subsets; thus, finding MCSes (equivalent to finding their complementary MSSes) is easier than finding MUSes directly. This follows from the relative simplicity of problems in NP (e.g., SAT) as compared to those in Co-NP (e.g., UNSAT). Therefore, our approach for generating all MUSes of a constraint system C is to first find MCSes(C) and then to compute the irreducible hitting sets of MCSes(C), which are all MUSes of C.

Our implementation of this approach for Boolean satisfiability finds MUSes of unsatisfiable CNF instances in two distinct phases, using an independent algorithm for each. The first phase, computing MCSes, is built on top of a constraint solver and requires few changes, if any, to the underlying solver. This makes our approach easily generalizable and simple to build on top of other solvers, for example to immediately exploit advances in constraint solver technology or to provide the functionality of finding MUSes for new types of constraints. The second phase, computing MUSes from the MCSes, uses a recursive branching algorithm we have developed to efficiently compute irreducible hitting sets, and it operates independently of the source of the MCSes.

Recall that both computing MUSes and computing MCSes are cases of computing irreducible hitting sets of some collection of sets. Why then do we use such different methods for the two phases of CAMUS? The methods contrast because in the first phase we are finding hitting sets of the MUSes, but the collection of MUSes is hidden from us. It is "encoded" within the constraints. We use a constraint solver to work with the information given and provide hitting sets (MCSes) without ever revealing the underlying MUSes themselves. In the second phase,

MCSes(φ)

1. $\varphi' \leftarrow$ **AddYVars**(φ)
2. MCSes $\leftarrow \emptyset$
3. k $\leftarrow 1$
4. **while** (**SAT**(φ'))
5. $\quad \varphi'_k \leftarrow \varphi' \wedge \text{AtMost}(\{\neg y_1, \neg y_2, \ldots, \neg y_n\}, \texttt{k})$
6. $\quad$ **while** (newMCS $\leftarrow$ **IncrementalSAT**(φ'_k))
7. $\quad\quad$ MCSes $\leftarrow$ MCSes $\cup$ {newMCS}
8. $\quad\quad \varphi'_k \leftarrow \varphi'_k \wedge$ **BlockingClause**(newMCS)
9. $\quad\quad \varphi' \leftarrow \varphi' \wedge$ **BlockingClause**(newMCS)
10. $\quad$ **end while**
11. $\quad$ k $\leftarrow$ k + 1
12. **end while**
13. **return** MCSes

Figure 2. Algorithm for finding all MCSes of a formula φ

all of the information we need is given explicitly in the set of MCSes, and so we can use a more direct, efficient method to compute irreducible hitting sets.

In this light, the method employed by CAMUS for computing MUSes of a constraint system may seem roundabout and unnecessary; it seems a more direct algorithm, which extracts the "hidden" information of the MUSes without going through the intermediate stage of MCSes, should exist. At this time, we are unaware of any technique that utilizes the hitting set duality efficiently without generating MCSes or their equivalent. The question of whether any such technique exists remains open for further research.

5. Computing MCSes

The first phase of CAMUS finds MCSes by successively solving an optimization problem similar to the MaxSAT problem. The goal is to find minimal sets of clauses whose removal renders the given formula satisfiable. As noted above, this is equivalent to finding maximal satisfiable subsets (MSSes) because the complement of any MCS (resp. MSS) is an MSS (resp. MCS). CAMUS finds MSSes by iteratively finding the largest satisfiable subset *that has not been found in a previous iteration*. Essentially, it solves a set of consecutive MaxSAT problems, each with the added restriction of excluding previously found results, until no satisfiable sets of clauses (modulo the restriction) remain.

Solving independent sequential optimization problems of that sort is very expensive, however; we avoid some of this expense by utilizing an incremental solver and retaining some information, such as learned clauses, between solutions. The pseudocode for the algorithm CAMUS uses to find every MCS of a formula φ is shown in Figure 2.

Our implementation for Boolean satisfiability is integrated directly with a modern SAT solver (specifically MiniSAT [15] version 1.12b in the current implementation), exploiting its efficient pruning and variable ordering heuristics. Satisfiable subsets are found in a standard SAT backtracking search after augmenting the input CNF instance φ with clause-selector variables to create φ' (line 1 of the pseudocode) as described in Section 2.2.

Using these clause selector variables does significantly increase the number of variables in the instance, and the search space grows correspondingly to the set of assignments to the original variables variables for any subset of the original clauses. This is the exactly the space we wish to search, however, and the increased instance complexity is unavoidable in this domain[2]. Furthermore, the clause-selector variables are added in a structurally very simple way. Along with the fact that learned clauses can now record interactions between original variables and clause activation, this leads to a tractable increase in complexity.

MCSes are obtained by finding assignments that satisfy φ' with a minimal set of y_i variables assigned FALSE, which ensures that as few constraints as possible are disabled. The set of y_i variables assigned FALSE indicates the clauses in an MCS. Solving multiple optimization problems of this sort separately would involve a great deal of duplicate work, so CAMUS utilizes a sliding objective approach that enables a more efficient incremental search, avoiding much redundancy. CAMUS finds all MCSes of a particular size within a single search tree, efficiently reusing information such as learned clauses and variable ordering. Specifically, each iteration of the outer **while** loop (lines 4–12) finds all MCSes of size k, which is incremented by 1 after each iteration.

Line 5 places an AtMost bound on the number of clause-selector variables that may be assigned FALSE by adding a constraint of the form $\text{AtMost}(\{\neg y_1, \neg y_2, \ldots, \neg y_n\}, k)$ to φ', creating φ'_k. Then, the **while** loop on lines 6–10 exhaustively searches for all satisfiable assignments to the augmented formula φ'_k, thus finding all MCSes of size k. The call to **IncrementalSAT** on line 6 uses MiniSAT's incremental solving ability to find a solution to the formula augmented with selector variables and the AtMost bound (φ'_k). Each satisfying assignment produces an MCS

[2] As opposed to, for example, linear constraints over the reals such as in [20], where the slack variables already used by the SIMPLEX algorithm can be used to similarly deactivate constraints.

from the set of y_i variables assigned FALSE. We made one small change to MiniSAT's ordering heuristics to better suit this problem: The default variable ordering was changed to always try the positive polarity of a variable first (the original code always tries the negative value first). This matches the general variable-ordering heuristic of aiming for solutions, as our solutions will have most clause-selector variables set TRUE, and those make up the majority of the variables. This change also performed better empirically than the original ordering. Future work can investigate more complex variable and value orderings crafted for this particular problem.

Each new MCS is recorded (line 7), and a blocking clause is added to both φ' and φ'_k to block that solution (lines 8 and 9). The blocking clause asserts that at least one of the clauses in the MCS must be enabled in any future solution. For example, if the MCS consists of clauses C_2, C_3, and C_6 (i.e., y_2, y_3, and y_6 are all FALSE in the satisfying assignment), the blocking clause will be $(y_2 \vee y_3 \vee y_6)$. This forces at least one of the y_i variables to be true, excluding the MCS and any of its supersets from any future solutions.

Finding MCSes in order of increasing size (i.e., MSSes in order of decreasing size) and excluding supersets from future solutions ensures that all MCSes found are irreducible. Incrementing by 1 after exhausting all solutions with a bound of k forces any solutions then found with $k+1$ disabled clauses to be irreducible, because any potential subsets would have been found earlier and blocked. Within a search with a given bound, the incremental SAT solver can utilize learned clauses and dynamic variable ordering heuristics to full effect.

An incremental search only works if changes to the constraint system do not create new solutions in previously explored portions of the search tree; as long as CAMUS *adds* constraints (the blocking clauses), it can use an incremental search. Incrementing the bound, however, relaxes a constraint, potentially creating new solutions where there were none before and invalidating much of the learned clause database. When that occurs, the search starts over for solutions of φ' augmented with all blocking clauses created thus far and the new AtMost bound.

The condition of the outer **while** loop on line 4) checks that φ' augmented with all collected blocking clauses is still satisfiable *without* any bound on the y_i variables. If there is no satisfying assignment, even with no AtMost constraint on the y_i variables, then this indicates that we have found *and blocked* all possible ways of removing clauses to yield a satisfiable set. Thus, the entire set of MCSes has been found, and the algorithm terminates.

Consider the execution of **MCSes** on our example formula. In its first iteration, it will add an AtMost bound with $k = 1$, and it will find

the only one-clause MCS $\{C_1\}$, corresponding to the MaxSAT solution. After adding the blocking clause, (y_1), the incremental solver will be unable to find any further solutions with the same AtMost bound in place. There is no other way to remove one clause to satisfy φ, no other single clause *covers* all of its MUSes. After exiting the inner **while** loop, the bound is increased to 2, and the search continues, because the augmented formula φ' (without the AtMost bound) is still satisfiable. This iteration will not find any new MCSes, however, because without being able to remove the first clause, there is no set of two clauses that covers all of the MUSes (i.e., there are no two-clause MCSes). After incrementing the bound once more to 3, the remaining MCSes will all be found within the next iteration. When all of the 3-clause MCSes are found, the main **while** loop will exit, because φ' with all of the blocking clauses added is no longer satisfiable; there is no way to choose one y_i (enabling one original clause) from each blocking clause without enabling an entire MUS.

6. Computing MUSes

Once the entire collection of MCSes has been computed, the second phase of CAMUS produces all MUSes of the given instance by finding all irreducible hitting sets of the MCSes. This problem is equivalent to computing all minimal transversals of a hypergraph, for which many algorithms have been developed. We developed a new algorithm from first principles, as described below, that performs better experimentally in the specific application of CAMUS, i.e., on collections of MCSes, than any other algorithm of which we are aware (an experimental comparison is presented in Section 9.3).

The following subsections describe our algorithms for finding all irreducible hitting sets of *any* collection of sets. They are independent of the first phase of CAMUS in that they do not depend on the semantics of the inputs as MCSes, and they can be applied to any minimal hitting set or hypergraph transversal problem. We present the algorithms in terms of their inputs being MCSes and their outputs MUSes, however, to maintain a stronger connection with the other concepts in this paper.

6.1. Computing a Single MUS

Consider the problem of computing a single MUS. Given a collection of sets of clauses, the MCSes, the goal is a set of clauses that "hits" every set in that collection *and* is irreducible. The first criterion, that of hitting each set, could be met by iteratively choosing arbitrary clauses

PropagateChoice(`MCSes`, `thisClause`, `thisMCS`)

```
1. for each clause ∈ thisMCS
2.     for each testMCS ∈ MCSes
3.         if (clause ∈ testMCS)
4.             testMCS ← testMCS - {clause}
5.         end if
6.     end for
7. end for
8. for each testMCS ∈ MCSes
9.     if (thisClause ∈ testMCS)
10.        MCSes ← MCSes - {testMCS}
11.    end if
12. end for
13. MaintainNoSupersets(MCSes)
```

Figure 3. Algorithm for altering `MCSes` to make the choice of `thisClause` irredundant as the only element hitting `thisMCS`

from MCSes that have not yet been hit until we have hit each MCS at least once. This alone does not guarantee an irreducible solution, however.

Notice that for a solution to be irreducible, each element must be *irredundant*. In the case of generating MUSes, this means that every clause in the solution must be the sole "representative" of at least one MCS. For example, given a collection $\{\{C_1, C_2, C_3\}, \{C_2, C_4\}\}$, one could generate a hitting set by the simple algorithm described above: $\{C_1, C_2\}$. But the element C_1 is redundant, because there is no set for which it is the sole representative; the trivial algorithm will not produce *irreducible* hitting sets. One potential solution to this problem is to filter redundant clauses out of every candidate MUS, but this will not scale.

The approach that we have taken is to *force* every selected clause to be irredundant by altering the remaining problem after each selection. Given some clause C_i and a particular MCS in which it appears, removing the other clauses in that MCS from the remaining problem ensures that C_i will not be redundant in the solution. For example, given a set of MCSes $\{\{C_1, C_2, C_3\}, \{C_2, C_4\}, \{C_2, C_5\}\}$, we can select C_3 to be contained in a growing MUS. It appears only in the first MCS of the set, so we will alter the remaining MCSes to enforce that C_3 is irredundant by removing C_1 and C_2 entirely. This leaves $\{\{C_4\}, \{C_5\}\}$ as the remaining subproblem.

SingleMUS(`MCSes`)

```
1. MUS ← ∅
2. while (MCSes ≠ ∅)
3.     selClause ← SelectRemainingClause(MCSes)
4.     selMCS ← SelectMCSContaining(MCSes,selClause)
5.     MUS ← MUS ∪ {selClause}
6.     PropagateChoice(MCSes, selClause, selMCS)
7. end while
8. return MUS
```

Figure 4. Algorithm for computing a single MUS from a set of MCSes

Figure 3 contains pseudocode for a subroutine that propagates a choice of clause and MCS containing it in this way. Lines 1–7 make the choice of `thisClause` irredundant as described, preventing any of the other clauses in `thisMCS` from being added in later iterations. Lines 8–12 remove any other MCSes hit by choosing `thisClause`, because they have now been "satisfied" by the partial solution. Line 13 calls a subroutine that removes any set in MCSes that is now a superset of some other. This last step is needed because our algorithm requires that no MCS is a superset of any other (which is by definition the case for the initial set of "real" MCSes, but must be maintained manually in the induced subproblems).

Computing a single MUS from the collection of MCSes is shown in pseudocode in Figure 4. It follows the simple method outlined above, using the **PropagateChoice** subroutine to modify the remaining MCSes after selecting a clause for inclusion in the MUS and some MCS containing that clause. The choice of clause and MCS can be arbitrary. When `MCSes` is empty, the set `MUS` contains an irreducible hitting set of `MCSes`; every MCS has been hit by some selection, and each selection was forced to be irredundant.

6.2. Computing All MUSes

We developed our algorithm for computing all MUSes from the algorithm for finding a single MUS above. Notice that the selections of a clause and an MCS in which it appears (on lines 3 and 4 in Figure 4) are arbitrary. Different MUSes can be computed by making different choices at those two points. Therefore, we generate the complete set of MUSes with a recursive algorithm that branches at those two points and tries all possible choices for each. The pseudocode for this algorithm is shown in Figure 5.

AllMUSes(MCSes, currentMUS)

```
1. if (MCSes = ∅)
2.     print(currentMUS)
3.     return
4. end if
5. for each selClause ∈ RemainingClauses(MCSes)
6.     newMUS ← currentMUS ∪ selClause
7.     for each selMCS ∈ MCSes such that selClause ∈ selMCS
8.         newMCSes ← MCSes
9.         PropagateChoice(newMCSes, selClause, selMCS)
10.        AllMUSes(newMCSes, newMUS)
11.    end for
12. end for
13. return
```

Figure 5. Algorithm for computing the complete set of MUSes from a set of MCSes

AllMUSes takes as input (1) the remaining set of MCSes and (2) the MUS currently being constructed in each branch of the recursion (initialized at the root of the recursion to the complete set of MCSes and the empty set, respectively). The recursion terminates in the conditional on lines 1–4 when no MCSes remain, at which point it outputs the MUS constructed in the current recursion branch and returns to explore other branches. Lines 5–12 iterate through all possible choices of a clause (selected on line 5) that is added to the growing MUS and an MCS (line 7) in which it appears. For every such choice, **PropagateChoice** is called to modify a copy of the current MCSes. The recursion then descends into another call to **AllMUSes** with the new MCSes and the current MUS. In terms of the matrix representation of a set of MCSes depicted in Figure 1, the nested **for** loops can be thought of as iterating over every single X in the matrix of the current MCSes. The selection order does not affect correctness, and what we show here is just one possible ordering that works well in our implementation. Other orderings can be explored in future work, mainly with regards to their interplay with the optimizations discussed below and their effect on runtime.

We have presented the algorithm in its most basic form to illustrate the fundamental concepts behind its operation. We made a number of additions and optimizations to increase the performance well beyond that of the basic algorithm, though the overall operation remains the same.

The first optimization addresses the fact that this algorithm can produce duplicate outputs. While the selections made on lines 5 and 7 determine which MUS is produced, the result for a given set of selections is not unique. For example, given a partial set of MCSes, $\{\{C_1, C_2\}, \{C_1, C_3\}\}$, the simple algorithm will return $\{C_1\}$ as a solution twice, because there are two MCSes from which to chose C_1, both leading to that solution. Reporting duplicate results can be eliminated by recording visited states and pruning portions of the recursion tree that match any stored state. The saved state could be as simple as the final MUSes output (in which case nothing is pruned, but duplicate outputs are avoided) or as complex as the complete input to the recursive **AllMUSes** procedure. In our implementation, we use a hash table to store an intermediate state (based on the `currentMUS` input and the set of removed clauses) at each call to **AllMUSes**, returning immediately from **AllMUSes** if the current state matches an entry already in the table. This prunes a large portion of the recursion tree, yielding considerable speedups in our experience: up to an order of magnitude in the automotive benchmarks reported in Section 8.

An ordering heuristic provides the second major optimization. Though we are using a complete search, which will have the same number of solution leaf nodes regardless of order, the ordering does affect the number of redundant nodes and interacts with the pruning from the first optimization to change the size of the recursion tree. We impose a static ordering of the clauses that is used by the **for each** loop on line 5 to select the next remaining clause in each iteration. We order the clauses by their frequency, i.e., the number of MCSes in which they appear. Selecting clauses in order of increasing frequency experimentally yields the best performance overall, though the opposite ordering performs better in some instances.

Other important optimizations are a subroutine that immediately includes the clauses in any single-element MCSes (similar to unit-clause propagation in Boolean satisfiability solvers [13, 12]) when they appear due to modifications made by **PropagateChoice**; explicitly removing a clause from the remaining MCSes after it has been tried in an iteration of the **for each** loop starting on line 5; and carefully optimizing the **MaintainNoSupersets** subroutine, as well as how it is called, to avoid redundant work.

7. Variations

Several variations of these algorithms can be used to suit particular classes of instances or to achieve different goals. In this section, we

first present modifications that combat intractability by relaxing the completeness criteria without sacrificing correctness. Another variation involves grouping constraints for greatly improved performance and more meaningful results when dealing with constraints encoded from a higher-level language.

7.1. Relaxing Completeness to Combat Intractability

In addition to the fact that the set of MUSes can be exponentially large, the complete set of MCSes is potentially exponential in the size of the original instance as well. For example, an instance with n pairwise disjoint MUSes each having k clauses (e.g., $\{\{C_1, C_2, C_3\}, \{C_4, C_5, C_6\}, \ldots\}$) will have k^n MCSes with n clauses each. The second phase of CAMUS can be stopped at any time to deal with massive sets of MUSes, but for those cases with intractably large sets of MCSes, the completeness criterion of the first phase of CAMUS must be relaxed. While the **MCSes** algorithm in Figure 2 is technically an anytime algorithm in that it returns results as they are found during search, one cannot generate MUSes by halting **MCSes** early and passing a subset of the MCSes to the **AllMUSes** algorithm. Hitting sets of any proper subset of the collection of MCSes may not be unsatisfiable.

Therefore, we have developed a modification of the **MCSes** algorithm that produces an output smaller than the complete set of MCSes while still guaranteeing that irreducible hitting sets of its output will be MUSes. It is not possible to generate *all* MUSes from this smaller first stage result, but that is a direct consequence of relaxing the completeness criterion of the first phase.

As presented, the first phase of CAMUS computes all of the MCSes and the second builds MUSes by branching on which clauses will be included in each resulting MUS. Clearly, by pruning some of the branches in the second phase (i.e., eliminating some choices from each branching point), we can greatly reduce the number of MUSes returned by the algorithm. And in fact that pruning can be done earlier, within the first phase, to reduce the number of MCSes computed as well. Just as the **AllMUSes** algorithm removes clauses from the problem when descending into a branch, so too can we remove clauses from the remaining problem at any point during the search for MCSes. By doing this, we can reduce the size of the results of both phases, reducing the complexity and effectively overcoming intractability by returning a portion of the complete results. Note that this does not relax correctness at all; all of the outputs of the second phase will still be minimal.

PCSes(φ)

```
 1. φ' ← AddYVars(φ)
 2. k ← 1
 3. PCSes ← ∅
 4. while (SAT(φ'))
 5.     φ'_k ← φ' ∧ AtMost({¬y_1, ¬y_2, ..., ¬y_n}, k)
 6.     while (newMCS ← IncrementalSAT(φ'_k))
⋆7.         (keptClauses,removedClauses) ← Truncate(newMCS)
⋆8.         newPCS ← keptClauses
⋆9.         φ'_k ← RemoveClauses(φ'_k, removedClauses)
⋆10.        φ' ← RemoveClauses(φ', removedClauses)
 11.        PCSes ← PCSes ∪ {newPCS}
 12.        φ'_k ← φ'_k ∧ BlockingClause(newPCS)
 13.        φ' ← φ' ∧ BlockingClause(newPCS)
 14.    end while
 15.    k ← k + 1
 16. end while
⋆17. PCSes ← RemoveSubsumed(PCSes)
 18. return PCSes
```

Figure 6. A generalization of **MCSes**(), capable of finding PCSes (Partial Correction Subsets) of a formula φ [A $\star$ indicates a line not in **MCSes**()]

Figure 6 contains pseudocode for a modified **MCSes** algorithm, which we call **PCSes** (Partial Correction Subsets), that accomplishes the described relaxation. Lines 7–10 and 17, marked with $\star$ symbols, have been added, while the remaining lines are not significantly changed from **MCSes**. The major change in this algorithm is that we are now interested in subsets of MCSes, found by *truncating* MCSes, which are still computed in the same way as in **MCSes**. We refer to a truncated MCS as a *Partial Correction Subset* (PCS):

Definition 5. A subset $P \subseteq C$ is a PCS if there exists some MCS M such that $P \subseteq M$.

Lines 7–10 accomplish this truncation in three main steps: First, each computed MCS is split into two subsets via a **Truncate** subroutine, `keptClauses` and `removedClauses`; second, a PCS is created that contains only `keptClauses`; and third, the clauses in the set `removedClauses` are removed entirely from the instance. Overall, this is equivalent to pruning any branches of the **AllMUSes** algorithm in which any clause from `removedClauses` is selected; it can reduce the

number of MUSes computed in the second phase, and it has an added benefit of reducing the size of the first phase's output as well (each PCS "represents" all of the MCSes that are supersets of it, so fewer are needed to form a complete set).

The internals of **Truncate** are left unspecified because the subroutine can be implemented in different ways to achieve numerous different goals. The only requirements are that 1) it splits the given MCS into two subsets, `keptClauses` and `removedClauses`, such that `keptClauses` is non-empty, and 2) any clauses that were included in `keptClauses` in a previous call to **Truncate** are again in `keptClauses`. The latter requirement, crucial for correct operation, arises from the idea that any earlier split into `keptClauses` and `removedClauses` was making a decision about which clauses to consider, and removing a previously kept clause would conflict with that decision[3].

Line 17 adds a call to a subroutine that removes subsumed PCSes, that is, PCSes that are supersets of others in the collection. We did not need this in the earlier **MCSes** algorithm because the technique of computing (and blocking) MCSes in increasing order of size precluded finding spurious supersets. In **PCSes**, the **Truncate** subroutine may invalidate the condition that results are found in this order. To have any real control over the size of the output, **Truncate** must be able to limit the size of `keptClauses` as much as possible. Yet the requirement that it include any clauses kept in previous calls could force it to return more than the desired limit. For example, while in the main loop of **PCSes** with an AtMost bound of $k = 3$, it could be forced to include four clauses in one PCS because all four have been included in previous PCSes. When it returns to the prescribed limit of three clauses per PCS in later iterations, it could produce PCSes that are subsets of the one that was forced to be larger. This would cause the larger PCS to be subsumed and redundant. In the pseudocode, we have placed the call to **RemoveSubsumed** at the end of the entire process, though it could be somewhat more efficiently implemented within the main loop.

Whenever **Truncate** returns a non-empty `removedClauses` set, those clauses are removed from the problem entirely. The final result is equivalent to pruning any branches of **AllMUSes** in which one of those clauses is chosen. Removing the clauses from the problem in the first phase has the added benefit of reducing the size of the first phase's output as well. The following examples should aid in understanding how the addition of MCS truncation affects the performance and the results of the **PCSes** algorithm. Many different useful behaviors can be

[3] Another way to handle this conflict is to remove any clauses in the `removedClauses` set from any previously computed PCSes as well. We have implemented the approach that prevents previously kept clauses from being removed.

built from these examples, and the variety of possible implementations is quite large.

Example 1. The behavior of the original **MCSes** algorithm is contained within **PCSes** in the case where **Truncate** always returns the entire MCS in `keptClauses` and the empty set for `removedClauses`.

Example 2. Consider the variant of **PCSes** in which **Truncate** returns only a single clause in `keptClauses` every time it is called. This variant (the most extreme possible in terms of number of clauses removed) finds a single MUS of the original instance. The final set of PCSes will be a collection of single-element sets, whose only irreducible hitting set is the union of those sets. Compare this result to a single path to a leaf node in the **AllMUSes** algorithm. For every MCS under consideration, we chose one clause to represent it and removed the others from the problem; we essentially moved the decisions made along one path of **AllMUSes** into the first phase of CAMUS. This method can generate any of the MUSes of a given instance, depending only on the clause chosen by **Truncate** to be kept in each iteration.

Example 3. PCSes can be used to heuristically obtain a diverse sampling of the space of MUSes by attempting to find dissimilar MUSes. This is useful in systems with goals of correcting or acting on knowledge of all causes of the infeasibility. Just as eliminating one MUS may not be enough to eliminate infeasibility, it is also unlikely that removing a cluster of similar MUSes would. Furthermore, interactive systems presenting MUSes to users, for example to explain the infeasibility of a scheduling problem, are more useful if they present a diverse set of MUSes, as this will provide a more comprehensive set of explanations than one MUS or several similar MUSes.

This variant operates in a greedy fashion by iteratively finding a single MUS with the approach in Example 2 while *biasing* the clause selection within **Truncate** each time towards keeping clauses that were *not* included in a previous iteration's result. That is, a counter is kept for every clause, and a clause's counter is incremented when that clause is included in one of the MUSes returned. **Truncate** takes the clauses in `newMCS` and sorts them in increasing order of that count, taking the clause with the smallest count for `keptClauses` each time it is called. This will produce a sampling of the MUSes biased towards including "underrepresented" clauses in each new result. It can be extended to more complex biases, such as looking at the actual structure of the MUSes found thus far and biasing the search away from those structures, as opposed to just the contents of previous MUSes.

Figure 7 shows the execution of this variant of **PCSes** on our example formula. The algorithm is run three times with a truncation limit

$$\begin{array}{cccccccccccc} & C_1 & & C_2 & & C_3 & & C_4 & & C_5 & & C_6 \\ \varphi = & (x_1) & \wedge & (\neg x_1) & \wedge & (\neg x_1 \vee x_2) & \wedge & (\neg x_2) & \wedge & (\neg x_1 \vee x_3) & \wedge & (\neg x_3) \end{array}$$

	Execution	**Clause Counts**
Initialization		[0,0,0,0,0,0]
1st Run	Find MCS $\{C_1\}$	[**0**,0,0,0,0,0]
	Keep PCS $\{C_1\}$	
	Find MCS $\{C_2, C_3, C_5\}$	[0,**0**,**0**,0,**0**,0]
	Keep PCS $\{C_2\}$	
	Final MUS $\{C_1, C_2\}$	[1,1,0,0,0,0]
2nd Run	Find MCS $\{C_1\}$	[**1**,1,0,0,0,0]
	Keep PCS $\{C_1\}$	
	Find MCS $\{C_2, C_3, C_5\}$	[1,**1**,**0**,0,**0**,0]
	Keep PCS $\{C_3\}$	
	Find MCS $\{C_4\}$	[1,1,0,**0**,0,0]
	Keep PCS $\{C_4\}$	
	Final MUS $\{C_1, C_3, C_4\}$	[2,1,1,1,0,0]
3rd Run	Find MCS $\{C_1\}$	[**2**,1,1,1,0,0]
	Keep PCS $\{C_1\}$	
	Find MCS $\{C_2, C_3, C_5\}$	[2,**1**,**1**,1,**0**,0]
	Keep PCS $\{C_5\}$	
	Find MCS $\{C_6\}$	[2,1,1,1,0,**0**]
	Keep PCS $\{C_6\}$	
	Final MUS $\{C_1, C_5, C_6\}$	

Figure 7. Running **PCSes** on the example formula φ – three separate runs with the truncation limit set to 1 kept clause, biasing clause selection by previous selection frequency

of 1 clause, keeping a count for each clause of how many times it has been in some resulting MUS. The clause counts guide the truncation in each run.

An adaptive implementation of **Truncate** can provide more complex behaviors, such as enabling rough limits on runtime or output size without sacrificing correctness. Its implementation is left for future work, but we present important considerations here. Each clause removed by **Truncate** directly impacts the runtime and output size of the first phase of CAMUS, so both can be controlled to some degree by controlling the frequency of removing clauses. For example, one could set a rough runtime limit and gradually (or sharply) increase the

frequency of removing clauses as the limit is approached. This cannot immediately halt execution – recall that previously kept clauses must be kept in each new PCS, so several more PCSes may be generated even after setting a truncation limit to remove as many clauses as possible – but it can drastically reduce the remaining runtime; hence it can provide a *rough* limit on runtime.

A rough limit can be placed on the size of the generated MUS set in the same way. However, knowing when to increase the clause removal frequency requires a means of estimating the number of MUSes that will be produced by the final set of PCSes at any point as they are generated. One such estimation function approximates a maximal independent set (MIS) of the current PCSes; multiplying the cardinalities of the PCSes included in the MIS estimate gives an estimate of how many MUSes would be produced from the PCSes. Unfortunately, this is neither a strict upper nor lower bound on the actual size, and in practice it can be off by several orders of magnitude. Other inaccurate yet simple estimates could be produced from the number of PCSes of each size, using the idea that each PCS of size k will generally increase the number of MUSes by a factor proportional to k – such factors could be determined experimentally for any given class of problems. These estimates will function if the goal is to differentiate between, say, 100 and 100,000 MUSes, but not for fine-grained estimation.

7.2. Constraint Grouping

In many applications of constraint solvers, including Boolean satisfiability solvers, instances are created by encoding constraints from some higher level language. For example, several model checking systems take problems specified in expressive first-order logics such as Alloy [22] and the CLU logic [9] and encode them as Boolean CNF instances which are passed on to standard SAT solvers. In these cases, knowledge about which low-level constraints are generated from which high-level statements can be used to greatly increase performance and produce MUSes of the high-level statements directly. Instead of assigning a single clause-selector variable y_i per low-level constraint, one y_i variable is created per high-level statement, and it is added to every constraint generated from that statement.

With these selector variables, the search for satisfiable subsets (in **MCSes**) can now enable or disable entire statements from the original problem at once; the MCSes and MUSes generated are subsets of those high-level statements. In addition to providing directly meaningful results (not requiring a mapping back from low-level constraints), this greatly improves performance by reducing the size of the search

space exponentially (because the size of the search space is exponential in the number of selector variables). Additionally, a single MUS of the high-level statements may lead to several MUSes in its low-level encoding (potentially exponential in the size of the high-level MUS), and the grouping eliminates this added complexity as well. Grouping constraints in this way proved to be valuable when applying CAMUS in [1] (which uses the CLU logic), because the running time of CAMUS was unusably high without this optimization.

8. Experimental Results

Here, we report empirical results demonstrating the performance of CAMUS on a variety of benchmarks and with a variety of configurations. We aim to show both the general performance of our algorithms as well as the effectiveness of certain implementation choices we made. As mentioned earlier, our implementation of the first phase of CAMUS is based on MiniSAT [15] version 1.12b. The entire implementation is written in C++, compiled for the x86-64 instruction set by the g++ compiler with the -O3 optimization flag. All experiments were run in Linux on a 2.2GHz AMD Opteron processor with 8GB of RAM.

8.1. General Performance

Table I contains experimental data produced from a set of CNF benchmarks from an automotive product configuration domain [32, 34]. Each instance encodes a set of available configurations for a product, along with constraints enforcing a specific property to be checked. We observed that the encodings contained numerous duplicate clauses, which can yield a combinatorial explosion of MUSes; we removed the duplicate clauses from each instance before gathering data. There are a total of 84 benchmarks in the set, each with around 1500–1800 variables and 4000–8000 clauses (after removing duplicate clauses). We set a 600 second timeout on each phase of CAMUS. It was able to complete the stage of finding all MCSes within this timeout for 49 of the 84 instances. Table I reports on 39 of these 49 instances[4]. On the 35 instances for which CAMUS did *not* find all MCSes in 600 seconds, it did output an average of 26,000 MCSes per instance within that time, indicating

[4] The 10 instances excluded from this table were left out for space reasons, and all matched very closely in terms of runtimes and output to at least one instance that is included in the table. The results for the excluded instances, as well as additional data such as MUS sizes and results for other benchmarks, are available online at: `http://www.eecs.umich.edu/%7Eliffiton/camus/results.php`

Table I. Experimental results for automotive product configuration benchmarks

	Runtime (sec)		MCS sizes			
Name	MCSes	MUSes	#MCSes	Min	Max	#MUSes
C168_FW_UT_851	0.301	0.001	30	1	8	102
C170_FR_RZ_32	0.269	0.486	242	1	2	32768
C170_FR_SZ_58	0.341	7.18	177	1	8	218692
C170_FR_SZ_92	0.141	0	131	1	1	1
C170_FR_SZ_95	0.218	–	175	1	3	$> 2 \cdot 10^7$
C170_FR_SZ_96	4.19	–	2605	1	22	$> 1 \cdot 10^7$
C202_FS_RZ_44	5.93	–	2658	1	48	$> 7 \cdot 10^6$
C202_FS_SZ_121	0.101	0.001	24	1	2	4
C202_FS_SZ_122	0.109	0	33	1	1	1
C202_FS_SZ_95	448	–	59307	1	51	$> 6 \cdot 10^6$
C202_FS_SZ_97	20.6	–	7823	1	46	$> 5 \cdot 10^6$
C202_FW_RZ_57	0.434	0.001	213	1	1	1
C202_FW_SZ_118	0.5	–	257	1	2	$> 1 \cdot 10^7$
C202_FW_SZ_123	0.174	0	38	1	2	4
C208_FA_RZ_43	6.66	–	4317	1	20	$> 8 \cdot 10^4$
C208_FA_RZ_64	0.215	0.001	212	1	1	1
C208_FA_SZ_120	0.076	0	34	1	2	2
C208_FA_SZ_87	0.309	0.545	139	1	12	12884
C208_FA_UT_3254	0.387	0.349	155	1	4	17408
C208_FC_RZ_70	0.229	0.001	212	1	1	1
C208_FC_SZ_127	0.067	0	34	1	1	1
C210_FS_RZ_38	113	–	12715	1	141	$> 5 \cdot 10^6$
C210_FS_RZ_40	0.275	0.002	212	1	2	15
C210_FS_SZ_107	163	–	16511	1	141	$> 2 \cdot 10^6$
C210_FS_SZ_123	0.526	–	363	1	3	$> 1 \cdot 10^7$
C210_FS_SZ_129	0.088	0	33	1	1	1
C210_FW_RZ_57	337	–	20007	1	213	$> 4 \cdot 10^6$
C210_FW_RZ_59	0.374	0.001	212	1	2	15
C210_FW_SZ_111	374	–	23625	1	179	$> 6 \cdot 10^6$
C210_FW_SZ_129	1.13	–	584	1	5	$> 7 \cdot 10^6$
C210_FW_SZ_135	0.138	0.001	33	1	1	1
C220_FV_RZ_12	0.253	1.4	150	1	6	80272
C220_FV_RZ_13	0.199	0.118	76	1	6	6772
C220_FV_RZ_14	0.085	0.001	20	1	3	80
C220_FV_SZ_114	10.8	–	5654	1	55	$> 1 \cdot 10^6$
C220_FV_SZ_121	0.163	0.001	102	1	3	9
C220_FV_SZ_46	4.44	–	1533	1	52	$> 1 \cdot 10^7$
C220_FV_SZ_55	18.1	–	3974	1	22	$> 2 \cdot 10^6$
C220_FV_SZ_65	0.524	2.66	198	1	26	103442

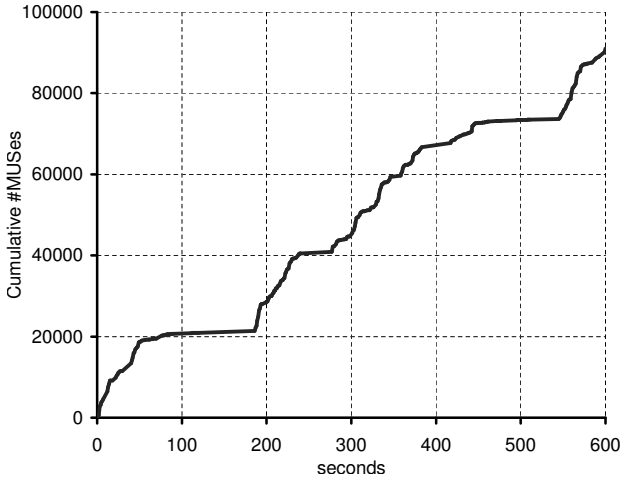

Figure 8. Anytime graph of computing MUSes from MCSes for instance C208_FA_RZ_43

that the output size was the primary factor in the hardness of those instances.

The first column of Table I gives the instance name. The next two columns contain the CPU time (in seconds) used by each phase of CAMUS; an entry of "-" indicates that the timeout for that phase was reached. The next group of columns lists the number of MCSes in each instance as well as the size of the smallest and largest MCS, and the final column reports the number of MUSes found. A number of MUSes preceded by a ">" indicates the number output before reaching the timeout.

Though all of the instances were generated in the same manner and have the same general size, the number and size of MCSes and MUSes in each instance vary widely. Some have a single MUS, while others have millions; runtimes can range from less than a millisecond to days or longer. This is to be expected, because either the number of MCSes or MUSes can potentially be exponential in the size of the original instance. One set can also provide an exponential "compression" of the other. C202_FW_SZ_118, for instance, has structured MCSes that can be analyzed to find that the instance has 2^{127} (approximately 1.7×10^{38}) MUSes.

Because of these potential output sizes, the best complexity one can hope to achieve for finding all MUSes is polynomial in the size of the output. In these benchmarks, both phases of CAMUS certainly do scale with the size of their outputs, though neither has theoretical guarantees that their runtimes will be sub-exponential in the size of the output.

When faced with exponential output, an *anytime* algorithm is essential. While we have not formulated the entirety of CAMUS as such, the second phase does provide good anytime performance. It guarantees that the initial output will come in polynomial time, and the remaining results are returned at a high rate. Comparing the number of MUSes found to the runtime of the second phase of CAMUS (including the partial results for instances that timed out) in Table I shows that, for these instances, the second phase of CAMUS generated close to 26,000 MUSes per second on average. Figure 8 shows an anytime curve (cumulative number of MUSes returned over time) for C208_FA_RZ_43, the *slowest* of the instances that timed out in the second phase. In this benchmark, there are some periods during which little is produced, but the overall rate of output remains relatively constant. Given some target number of MUSes, one could predict with reasonable accuracy how long the algorithm would take to generate it.

One interesting result is that the smallest MCS in every one of these instances contains a single clause. This indicates that all of the MUSes in each instance share at least one common clause, often more, because the only way to "hit" such a singleton MCS is to include its sole clause. Along with the fact that the largest MCS is usually a small percent of the clauses in each instance, this supports our technique of searching for MCSes by size incrementally. The MCSes are generally found within a small number of sizes, and the time spent searching empty "size blocks" is very small on average. We have observed the same characteristics in many other benchmark suites as well, though we can generate pathological cases for which they do not hold (such as the class of instances mentioned earlier with n pairwise disjoint MUSes of k clauses each and k^n MCSes with n clauses each).

8.2. Variations

8.2.1. *PCSes*

To demonstrate the value of the **PCSes** algorithm, we used it to find MUSes of the automotive benchmark instances for which **MCSes** timed out in our earlier experiments. We ran **PCSes** with a simple implementation of **Truncate** that takes a bound on the number of clauses to keep in each PCS and attempts to match it (it will at times be forced to keep more clauses if they were all kept previously). Table II lists results on the 35 automotive benchmark instances that timed out in **MCSes** during the earlier experiment. We report the runtime of **PCSes** in seconds and the number of MUSes constructed from the PCSes found for two different truncation bounds. We have not reported the runtime of the **AllMUSes** algorithm in this case; we have already established

Table II. Using **PCSes** to compute MUSes of the more difficult product configuration benchmarks

	Size limit = 2		Size limit = 3	
Name	PCSes (sec)	#MUSes	PCSes (sec)	#MUSes
C168_FW_SZ_107	29.6	2136	124	$> 1.4 \cdot 10^7$
C168_FW_SZ_128	3.02	6268144	7.32	$> 2.2 \cdot 10^7$
C168_FW_SZ_41	2.9	118	10.1	4500
C168_FW_SZ_66	4.16	248	13.1	434035
C168_FW_SZ_75	2.39	824	6.69	418463
C168_FW_UT_2463	6.14	1152	38.1	$> 8.3 \cdot 10^5$
C168_FW_UT_2468	5.66	13184	9.28	409509
C168_FW_UT_2469	5.43	1792	30.9	403392
C168_FW_UT_714	0.397	2	0.38	3
C202_FS_SZ_74	0.408	16	0.388	60
C202_FS_SZ_84	32.6	$> 1.3 \cdot 10^6$	138	$> 4.0 \cdot 10^6$
C202_FW_SZ_100	3.78	267	13.9	1105768
C202_FW_SZ_103	115	$> 1.2 \cdot 10^5$	273	$> 4.1 \cdot 10^6$
C202_FW_SZ_61	12.3	314	31.7	43238
C202_FW_SZ_77	0.826	64	0.727	144
C202_FW_SZ_87	101	$> 8.5 \cdot 10^4$	–	
C202_FW_SZ_96	35.2	$> 4.3 \cdot 10^4$	339	$> 5.6 \cdot 10^6$
C202_FW_SZ_98	1.66	123	9.38	37718
C202_FW_UT_2814	43.1	1198	267	4869852
C202_FW_UT_2815	43	1198	262	4869852
C208_FC_RZ_65	0.335	48	1.12	2494
C208_FC_SZ_107	1.74	400	4.5	32718
C210_FS_RZ_23	3.19	15406	3.59	474404
C210_FS_SZ_103	1.9	$> 1.4 \cdot 10^7$	6.42	$> 1.5 \cdot 10^7$
C210_FS_SZ_55	2.43	42608	4.93	14589828
C210_FS_SZ_78	1.17	48	1.35	432
C210_FW_RZ_30	8.03	58842	10.8	$> 1.6 \cdot 10^7$
C210_FW_SZ_106	11.2	$> 4.2 \cdot 10^6$	29.8	$> 1.8 \cdot 10^7$
C210_FW_SZ_128	1.03	28672	2.66	493568
C210_FW_SZ_80	2.53	16	2.74	440
C210_FW_SZ_90	35.1	$> 3.7 \cdot 10^4$	101	$> 1.6 \cdot 10^6$
C210_FW_SZ_91	34.5	$> 2.3 \cdot 10^6$	112	$> 2.4 \cdot 10^6$
C210_FW_UT_8630	15.8	16016	65.7	4192496
C210_FW_UT_8634	6.08	20480	68.9	5207976
C220_FV_SZ_39	12.6	$> 1.3 \cdot 10^4$	30	$> 2.7 \cdot 10^6$

the strong correlation between its runtime and the number of MUSes produced. Cases in which this algorithm timed out (again, with a 600 second timeout) are noted by a "$> n$" number of MUSes, indicating roughly how many MUSes were generated before the timeout.

The results show that the **PCSes** algorithm allows us to overcome the intractability of instances with massive numbers of MCSes by relaxing the requirement that we find all of them. Take the results of truncating every MCS found to a PCS of size 2 (or larger only in cases where it is forced as explained earlier), for example. With this setting, we can find a complete set of PCSes, allowing us to generate correct MUSes, in under two minutes – most under ten seconds – for all of the instances which timed out at 600 seconds in the complete **MCSes** algorithm. Even with this rather strict limitation on the size of the PCSes, we still compute very many MUSes for most instances, timing out after computing millions of MUSes in some.

At a PCS size "limit" of 3, we see that we have substantially higher runtimes. Related to this, we also generate much larger sets of PCSes; the median size of the set of PCSes is 185 for a size limit of 2, and this increases to 369 for a size limit of 3. These larger sets of PCSes produce many more MUSes, however, and thus there is a correlation between the runtime of **PCSes** and the number of MUSes the PCSes produce. This motivates the adaptive implementation of **Truncate** discussed in Section 7.1, which could provide a way to roughly aim for a certain number of MUSes within the execution **PCSes**. Along with the anytime nature of **AllMUSes**, this gives us a quasi-anytime algorithm for generating multiple exact MUSes. The runtime of the first phase, employing **PCSes**, can manipulated by controlling the frequency of removing clauses from the remaining problem. Furthermore, this can be adjusted based on an estimate of how many MUSes will be produced in the second phase. The second phase, as mentioned earlier, produces MUSes rapidly and can be stopped at any point, based on either reaching output goals or hitting limits on time or other resources.

Notice that with a PCS size limit of 1 clause, CAMUS will produce a single, exact MUS, as described in Example 2 in Section 7.1. CAMUS is not intended to compete with algorithms for finding a single MUS, and indeed it does not. The runtimes for a size limit of 1 over the instances in Table II range from 0.077 to 31.1 seconds, with a median runtime of 1.06 seconds, which is not competitive with existing algorithms for finding single unsatisfiable cores of CNF instances (e.g., [30] and [35]). However, with the generality of the algorithms in CAMUS, such that they can be easily built on top of any existing constraint solver, this provides a simple way to produce single MUSes in cases where no single-

MUS algorithm exists for a particular type of constraint (with the bonus of guaranteeing minimality).

8.2.2. *Constraint Grouping*
We demonstrate the effectiveness of constraint grouping (as described in Section 7.2) using a set of benchmarks taken from a hardware design verification task. The Reveal flow [1] performs equivalence checking of hardware designs including, but not limited to, microprocessors. The flow uses counterexample-guided abstraction refinement, in which abstractions of the input designs are checked for equivalence, and if a counterexample (indicating a difference) is found to be spurious (due to the abstraction over-approximating the designs' behaviors), then MUSes are used to refine the abstractions.

Specifically, abstract counterexamples are written as constraints in a first-order logic. These high-level constraints are encoded into CNF to find corresponding concrete, bit-level counterexamples. If the CNF instance is UNSAT, then no such concretization exists and the abstract counterexample is spurious. MUSes of this instance represent generalizations of the infeasibility, each essentially saying "This counterexample is spurious because $[x, y,$ and $z]$ can never occur together," where x, y, and z are some subset of the complete counterexample. Using this information, new facts can be added to the abstractions to avoid this counterexample, and due to the generalizations provided by the MUSes, a large class of related spurious counterexamples will be removed as well. Using all MUSes provides the best refinement of the abstraction, eliminating the largest set of spurious counterexamples.

The desired generalizations are actually MUSes of the high-level constraints. These can be obtained by mapping an MUS of the individual CNF clauses back to their corresponding high-level constraints *or* by using the constraint grouping described in Section 7.2 and computing MUSes of the high-level constraints directly. Table III contains results for using both approaches on benchmarks taken from the abstraction refinement phases of Reveal running on three different microprocessor designs.

The first four columns list the instance name and its size in terms of CNF variables, CNF clauses, and clause groups (equal to the number of high-level constraints). The following pairs of columns list the runtime in seconds of the first phase of CAMUS, the number of MCSes produced, and the number of MUSes (the runtime of the MUS phase is negligible in all of these instances). Each metric is reported both for the case of ignoring the constraint grouping information ("noG") and for the case of using the groups and finding MCSes and MUSes in terms of those groups ("G"). A 600 second timeout was used for these

Table III. Computing MCSes for a hardware verification task using constraint groups ("G") and without ("noG")

Name	Vars	Clauses	Groups	Runtime (sec)		#MCSes		#MUSes	
				noG	G	noG	G	noG	G
dlx_1	6804	78364	25	-	0.720	>795	3	-	5
dlx_2	6268	98290	23	-	0.664	>15196	2	-	2
dlx_3	5976	139141	18	-	1.160	>8172	2	-	4
dlx_4	12428	161242	16	15.7	1.120	327	2	3	2
dlx_5	17951	54212	21	-	0.428	>145	3	-	3
dlx_6	30852	92213	54	-	1.500	>0	9	-	9
dlx_7	36315	138197	15	2.27	0.716	39	2	1	1
int_1	1756	4634	7	-	0.232	>882	7	-	2
int_2	3512	4634	7	-	0.148	>1413	6	-	2
int_3	1704	4222	7	0.076	0.020	39	2	1	1
int_4	3886	5402	9	0.084	0.024	39	2	1	1
int_5	6291	5976	10	-	0.028	>62056	2	-	1
int_6	9481	8174	13	-	0.268	>1143	7	-	2
int_7	12671	8174	13	-	0.208	>978	6	-	2
oc_1	6129	14717	25	-	0.104	>12268	4	-	2
oc_2	12124	17500	24	-	0.100	>5420	3	-	1
oc_3	18436	18162	25	-	0.112	>5559	3	-	1
oc_4	23959	13405	23	-	0.124	>10882	3	-	2
oc_5	30271	18162	25	-	0.120	>4970	3	-	1
oc_6	5093	18129	19	-	0.124	>7563	2	-	2
oc_7	11277	17696	24	-	0.168	>7	3	-	4
oc_8	17374	14685	25	-	0.128	>10830	4	-	2
oc_9	23898	18762	26	-	0.164	>2130	2	-	2
oc_10	29421	13405	23	-	0.140	>10727	3	-	2
oc_11	32089	6197	10	0.288	0.052	38	2	1	1
oc_12	38401	18162	25	-	0.124	>5530	3	-	1
oc_13	44396	17500	24	-	0.132	>5458	3	-	1
oc_14	50708	18162	25	-	0.136	>5002	3	-	1
oc_15	54039	10834	12	-	0.092	>1760	2	-	1
oc_16	58995	14916	19	-	0.116	>13646	2	-	1
oc_17	63153	12853	17	-	0.108	>639	2	-	1

experiments. For those instances that timed out, we report the runtime as "-" and the #MCSes column contains the number of MCSes found before the timeout was reached.

These instances are all much larger than the others used in this paper, some reaching above 100,000 clauses. For this reason, running the first phase of CAMUS on them almost always times out. However, using the clause groups imposed by the higher level constraints results in greatly reduced runtime; all instances finished in under 2 seconds, most in a few hundred milliseconds. The number of MCSes found in both cases illustrates the source of the difference. The bare CNF instances tend to have several thousand MCSes (and quite likely several orders of magnitude more in many cases), and the size of the result set is simply too large. But when mapped to the high-level constraints, nearly all of these MCSes are redundant, in that they all map to just a few MCSes of the original constraints from which they were generated. The only instances on which the algorithm can complete *without* using the grouping information have very few MUSes (even in the raw CNF) and a structurally simple set of MCSes. Any application in which CNF clauses are generated from higher-level constraints will see the same benefits from this simple modification of the algorithm: markedly decreased runtime and direct applicability of the results. Another option is to use an implementation of CAMUS for the high-level constraints using a suitable solver. This is a good option if the constraint solver is more efficient than a modern SAT solver on the CNF encoding, and we have used this approach to good effect for the Reveal system with an implementation of CAMUS for SMT [2].

9. Related Work

The related research can be investigated in three separate areas. First, there is work on finding a minimal unsatisfiable subset of a constraint system in general as well as that on specifically finding all MUSes of a given system. Second, we look at research related specifically to the first half of CAMUS: finding the set of all MCSes. Finally, we present existing work related to the second half of CAMUS: the problem of finding all minimal hypergraph transversals or hitting sets.

9.1. MUSes

The problem of finding minimal unsatisfiable subsets of constraints has been studied mainly for Boolean satisfiability problems. Most techniques find a single unsatisfiable subset (US) or core, often not guar-

anteeing it to be minimal. For example, AMUSE [30], Bruni and Sassano's algorithm [8], and zCore [35] all use information from a SAT solver's resolution procedure to find a single US, but none guarantee its minimality. The work by Gershman, et. al. [17] further uses the resolution graph to aim for small cores, and an algorithm by Goldberg and Novikov [18] uses it to produce a core as a byproduct of verifying a proof of unsatisfiability; again, neither guarantees minimality. For all of these, a "Minimal Unsatisfiability Prover" [21] can be used to minimize the US into an MUS. Lynce and Marques-Silva [28] and Mneimneh, et. al. [29] both developed algorithms for finding the smallest MUS of a formula exactly, with the latter being far more efficient. Extending the work using SAT solvers' resolution procedures, Cimatti, et. al. produced an algorithm for finding small (not necessarily minimal) unsatisfiable cores of SMT instances [11]. Chinneck and Dravnieks [10] studied MUSes in the domain of linear and integer programs, calling them Irreducible Infeasible Subsets. Their algorithms return multiple, but not all, MUSes.

The work that is most relevant to ours, and the only that we know of that addresses the problem of finding *all* MUSes, is that by Bailey and Stuckey [4]. They developed an algorithm for finding all MUSes of a constraint system and applied it to the problem of type-error diagnosis in software verification. Their implementation differs from ours in that they employ different algorithms in an interleaved approach as compared to our serial, two-phase algorithm. In [27], we performed an experimental comparison of their approach, adapted to Boolean satisfiability, with CAMUS. We found that CAMUS was consistently faster by several orders of magnitude (and we have improved it since then). We have since learned that further improvements could be made to the published description of Bailey and Stuckey's algorithm (personal communication, J. Bailey, October 2005), but we do not believe they would completely erase the performance gap.

9.2. MSSes and MCSes

Maximal satisfiable subsets have been studied *apart* from their application to finding MUSes by Birnbaum and Lozinskii [6]. They are concerned with using MSSes (which they call *maximally consistent subsets* or *mc-subsets*) in knowledge systems, specifically to reason about inconsistent knowledge. As stated earlier, they noted the connection to MUSes (*minimally inconsistent subsets* in their paper), but they did not explore it further. They describe two algorithms, AMC1 and AMC2, for finding all MSSes (hence all MCSes) of a given CNF formula using a much different approach than that employed in CAMUS.

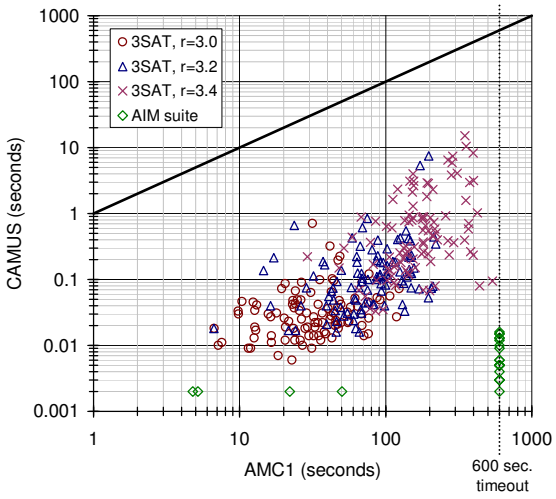

Figure 9. Comparing AMC1 and the first phase of CAMUS (3SAT instances have 30 variables)

We were unable to obtain Birnbaum and Lozinskii's implementations of their algorithms, so a direct comparison of their results with ours is difficult. To perform a limited comparison, we implemented AMC1, the faster of the two in their results, on top of the same SAT solver infrastructure used for our implementation of CAMUS. We implemented the algorithm exactly as shown in the paper, along with the suggested variable ordering, optimizing as much as we could without altering the algorithm[5]. Figure 9 contains a comparison of the runtimes of our implementation of AMC1 with the first phase of CAMUS on random 3SAT instances (each with 30 variables and clause/variable ratios (r) as indicated in the legend) and unsatisfiable instances from the AIM benchmarks (a set of small benchmarks often used in MUS papers – part of the DIMACS suite). Even taking the implementation differences into account, it is clear that the first phase of CAMUS is faster than AMC1 by several orders of magnitude.

AMC1 is a DPLL-style algorithm, searching through the space of variable assignments. It essentially enumerates complete variable assignments, checking the set of clauses satisfied by each to see if it is an MSS. Its only pruning rule is the pure literal rule; otherwise, it searches the entire space. This leads to very poor scaling; it even finds

[5] Comparing the runtime of our implementation of AMC1 to their reported results for random 3SAT instances and correcting for processor differences, we estimate that the runtimes of our implementation of AMC1 are approximately 3 times those of their implementation – not enough to significantly affect the orders of magnitude result in Figure 9.

each solution multiple times, equal to the number of different complete assignments that satisfy the corresponding MSS.

AMC2 performs more poorly than AMC1 in Birnbaum and Lozinskii's results. It is similar to the first phase of CAMUS in that it searches subsets of clauses for satisfiability, but as with AMC1 it has limited pruning abilities. It also has a less sophisticated search than CAMUS, which searches subsets of clauses implicitly within a standard SAT search, exploiting the SAT solver's pruning and dynamic ordering heuristics automatically.

9.3. Hypergraph Transversals / Hitting Sets

As noted earlier, the second phase of CAMUS consists of computing minimal hypergraph transversals (also known as hitting sets), a general graph problem (resp. set covering problem) with a long history in mathematics and computer science research. See [16] for an overview of applications of hypergraph transversals and some theoretical complexity results.

The minimal hypergraph transversal algorithm we developed for computing MUSes was created from first principles independently of existing algorithms. Since creating this algorithm for our purposes, we have looked for other algorithms for learning and comparison purposes. Of the other algorithms we have found, those with the most efficient implementations are *Partition*, by Bailey, et. al. [3]; an algorithm by Kavvadias and Stavropoulos [24, 25] (KS); and another by Boros, et. al. [7] (BEGK). The three algorithms were experimentally compared in [25], and while KS generally performed well, it did not entirely dominate the results over either of the other two. We obtained executables for all three algorithms[6] to compare their performance to that of CAMUS' second phase on sets of MCSes.

Figure 10 compares the runtimes of Partition, KS, and BEGK against our algorithm on the sets of MCSes from the product configuration benchmarks. We have only included benchmarks for which at least one of the algorithms finished within the 600 second timeout. Each point plots the runtime in seconds of our algorithm (y-axis) against either Partition, KS, or BEGK (x-axis); points below the diagonal indicate CAMUS outperforming the other algorithm for that point. To preserve results measured as zero seconds on the logarithmic scales, zero second runtimes have been changed to 0.0002 seconds. In all but two benchmarks, our algorithm either matches or outperforms the others: on

[6] James Bailey sent us an executable for Partition, while KS and BEGK were downloaded from `http://lca.ceid.upatras.gr/~estavrop/transversal/` and `http://paul.rutgers.edu/~elbassio/dual.html`, respectively.

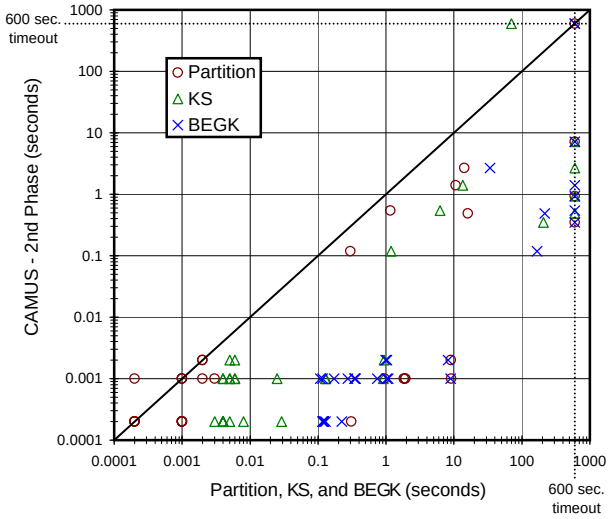

Figure 10. Comparing Partition, KS, and BEGK to the second phase of CAMUS on MCSes from automotive product configuration benchmarks

the MCSes of C202_FS_SZ_104, KS completed in 70 seconds while our algorithm (as well as the other two) timed out; the other point above the diagonal is a result that is below the threshold of timing accuracy in our experiment.

This is not meant to be a complete comparison of these algorithms, but rather it serves to motivate ours as a good choice for the types of hypergraphs (the MCSes) seen in CAMUS. In fact, the Partition algorithm is faster than ours on the machine-learning datasets for which *it* was developed (personal communication, J. Bailey, October 2005). Likewise, the results in [25] indicate that all three of the other algorithms have mixed performance rankings on different types of problems. Though it was not targeted directly, it is likely that the hypergraph transversal algorithm we developed for CAMUS is suited particularly well for some structural characteristic of sets of MCSes.

The algorithm most similar to ours is KS. It is similar to that in CAMUS in that it generates transversals (hitting sets) incrementally in a tree with complete transversals at the leaves. The algorithm therefore has the same good anytime properties as the second phase of CAMUS, taking negligible time to produce the first output and little time between each successive output. KS employs the concept of "generalized nodes," which treats sets of nodes that appear in the same set of hyperedges as a single generalized node to increase efficiency and reduce the size of the tree (each solution containing a generalized node may be expanded into several real solutions). Generalized nodes could be

implemented in our algorithm for a likely increase in efficiency. KS also employs a node-selection technique when adding nodes incrementally to prune redundant branches from the tree, as opposed to our algorithm, which accomplishes similar pruning by modifying the remaining subproblem at each point in the tree and storing "seen" branches in a hash table. This latter difference gives KS polynomial memory requirements as compared to the exponential memory requirements of our hash table. The memory usage of our algorithm has never presented a problem in practice, however, with experiments reaching practical timeouts well before practical memory limits were reached.

The hypergraph transversal / hitting set problem is also equivalent (with minor translation) to set covering, which has been studied extensively in the field of operations research (OR). Specifically, the problem we solve is closest to the *unicost* set covering problem; we have no weights or costs on the elements we are choosing. OR is mainly concerned with optimization problems, however, and any OR approaches of which we are aware are geared towards producing the *smallest* hitting set. For a recent example of one such approach to the unicost set covering problem and references to related techniques in OR, refer to [5]. OR approaches like this could be adapted to find all *irreducible* hitting sets by utilizing an iterative approach, blocking solutions as they are found as in the first phase of CAMUS, and this is an interesting direction for future research. However, preliminary experiments we performed using a similar incremental approach with MaxSAT to find all minimal hitting sets showed that it performs much worse than the algorithms we present in this paper. We believe that any procedure using repeated search/optimization to find all minimal hitting sets will not scale well.

10. Conclusions and Future Work

We have developed a broadly applicable framework for finding Minimal Unsatisfiable Subsets of constraint systems. The framework consists of a set of algorithms that we call CAMUS (**C**ompute **A**ll **M**inimal **U**nsatisfiable **S**ubsets). CAMUS is based on a set-covering relationship between maximal satisfiable subsets (MSSes) and minimal unsatisfiable subsets (MUSes).

We have shown how the algorithms in CAMUS can be modified to suit different types of instances and reach different goals. Most importantly, we described a variation that can avoid the general intractability of finding all MUSes of a constraint system (the output of which can be exponential in the size of the original problem) by relaxing the completeness criteria without resorting to approximations

or sacrificing correctness. We believe that the framework described in this paper is suited to further modifications and additions like those we have illustrated, and future work will include improving upon the efficiency of these algorithms.

One such addition has already been made to the algorithm in the first phase of CAMUS. Gregoire, et al., have boosted the **FindMCSes** algorithm with a local search oracle to increase its performance [19]. They use local search to identify candidate MCSes before the exhaustive search of **FindMCSes**. The local search is more efficient than the complete search in terms of the time taken to find candidates, but it is not guaranteed to be complete or correct (i.e., it may miss MCSes and some candidates it identifies may not be MCSes), so the search in **FindMCSes** is still required to verify candidates and complete the search.

The relationship between MSSes and MUSes applies to any type of constraint system, and the algorithms are easily generalized to work with different types of constraint solvers. In addition to the implementation for Boolean satisfiability instances that we have focused on in this paper, we have implemented the algorithms A) on a solver for Disjunctive Temporal Problems (DTPs) [26] and B) using YICES [14], an efficient Satisfiability Modulo Theories (SMT) solver which covers a wide range of constraint types, in a hardware verification system [2]. In all of these cases, very few, if any, modifications were made to the underlying solver. The algorithms in CAMUS can thus be implemented on top of new constraint solvers to immediately take advantage of advances in the field of constraint satisfaction and to provide the functionality of finding MUSes for new types of constraints.

Experimental results performed with our implementation of CAMUS for Boolean satisfiability show that it is suited for use on real-world problems. We show that even those instances whose complete results are intractably large can be tackled by the simple algorithm variants we have presented. Furthermore, we have shown that the algorithms in CAMUS perform better, often by several orders of magnitude, than existing algorithms for either of its two phases (finding all MCSes and computing irreducible hitting sets).

One interesting question worth exploring is whether one is more interested in MCSes or MUSes when faced with unsatisfiable constraint systems. As discussed, they are each a different "encoding" of the same information, but one will generally be more directly useful than the other. Indeed, it depends on the application; we have used the algorithms in CAMUS in two different hardware verification systems with exactly this differentiation. In the first, a design debugging / fault diagnosis system [31], the MCSes of the unsatisfiable instances

are the desired solutions directly; the first phase of CAMUS is used in this system with its grouping capability to boost an exact search by providing over-approximate solutions. The second system is the Reveal system described earlier, in which MCSes are not directly useful, while the MUSes provide exactly the information needed for abstraction refinement.

In the future, we plan to use CAMUS to investigate the structure of UNSAT instances and their complete sets of MUSes; little experimental work has looked at MUSes, as they have only recently been found in practical applications. We will continue to implement CAMUS with other constraint solvers and new constraint types to apply it in new domains. Linear programming and operations research is one possible direction, as the area has some MUS-related research [10] that appears disconnected from that in the fields of constraint processing, and we are unaware of any techniques for finding all MUSes of linear programs.

Finally, our experience indicates that the general intractability of the problem of finding all MUSes presents a real problem in practice, so we intend to expand on relaxing the completeness criterion. Improvements to **AllMUSes** are not as important as work on **MCSes** and **PCSes**, because it has never been the slow phase in practice. Due to the potentially exponential number of MUSes, however, we will investigate techniques for computing a "useful" subset of the MUSes in the second phase, where "useful" could mean diverse, containing some desired structural characteristic, or otherwise based on the application. The main algorithmic development will further refine **PCSes**, exploiting the potential of more complex implementations of the **Truncate** subroutine, because it is the practical way forward due to the intractability of the first, crucial phase of CAMUS.

Acknowledgments

We would like to thank James Bailey for making his implementation of the Partition algorithm available to us. We also thank the reviewers for their comments and insights. This material is based upon work supported by the National Science Foundation under ITR Grant No. 0205288. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the view of the National Science Foundation (NSF).

References

1. Andraus, Z. S., M. H. Liffiton, and K. A. Sakallah: 2006, 'Refinement strategies for verification methods based on datapath abstraction'. In: *Proceedings of the 2006 conference on Asia South Pacific design automation (ASP-DAC'06)*. pp. 19–24.
2. Andraus, Z. S., M. H. Liffiton, and K. A. Sakallah: 2007, 'CEGAR-Based formal hardware verification: a case study'. Technical Report CSE-TR-531-07, University of Michigan.
3. Bailey, J., T. Manoukian, and K. Ramamohanarao: 2003, 'A Fast Algorithm for Computing Hypergraph Transversals and its Application in Mining Emerging Patterns'. In: *Proceedings of the Third IEEE International Conference on Data Mining (ICDM'03)*. p. 485.
4. Bailey, J. and P. J. Stuckey: 2005, 'Discovery of Minimal Unsatisfiable Subsets of Constraints Using Hitting Set Dualization'. In: *Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages (PADL'05)*, Vol. 3350 of *LNCS*.
5. Bautista, J. and J. Pereira: 2007, 'A GRASP algorithm to solve the unicost set covering problem'. *Computers and Operations Research* **34**(10), 3162–3173.
6. Birnbaum, E. and E. L. Lozinskii: 2003, 'Consistent subsets of inconsistent systems: structure and behaviour'. *Journal of Experimental and Theoretical Artificial Intelligence* **15**, 25–46.
7. Boros, E., K. M. Elbassioni, V. Gurvich, and L. Khachiyan: 2003, 'An Efficient Implementation of a Quasi-polynomial Algorithm for Generating Hypergraph Transversals'. In: *Proceedings of the 11th European Symposioum on Algorithms (ESA 2003)*, Vol. 2832 of *LNCS*. pp. 556–567.
8. Bruni, R. and A. Sassano: 2001, 'Restoring Satisfiability or Maintaining Unsatisfiability by Finding Small Unsatisfiable Subformulae'. In: *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT-2001)*.
9. Bryant, R. E., S. K. Lahiri, and S. A. Seshia: 2002, 'Modeling and Verifying Systems Using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions'. In: *Proceedings of the 14th International Conference on Computer Aided Verification (CAV'02)*. pp. 78–92.
10. Chinneck, J. W. and E. W. Dravnieks: 1991, 'Locating Minimal Infeasible Constraint Sets in Linear Programs'. *ORSA Journal on Computing* **3**(2), 157–168.
11. Cimatti, A., A. Griggio, and R. Sebastiani: 2007, 'A Simple and Flexible Way of Computing Small Unsatisfiable Cores in SAT Modulo Theories'. In: *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing (SAT-2007)*. pp. 334–339.
12. Davis, M., G. Logemann, and D. Loveland: 1962, 'A machine program for theorem-proving'. *Communications of the ACM* **5**(7), 394–397.
13. Davis, M. and H. Putnam: 1960, 'A Computing Procedure for Quantification Theory'. *Journal of the ACM* **7**(3), 201–215.
14. Dutertre, B. and L. M. de Moura: 2006, 'A Fast Linear-Arithmetic Solver for DPLL(T)'. In: *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*. pp. 81–94.
15. Eén, N. and N. Sörensson: 2003, 'An Extensible SAT-solver'. In: *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT-2003)*, Vol. 2919 of *LNCS*. pp. 502–518.

16. Eiter, T. and G. Gottlob: 2002, 'Hypergraph transversal computation and related problems in logic and AI'. In: *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA 2002)*. pp. 549–564.
17. Gershman, R., M. Koifman, and O. Strichman: 2006, 'Deriving Small Unsatisfiable Cores with Dominators'. In: *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*. pp. 109–122.
18. Goldberg, E. and Y. Novikov: 2003, 'Verification of Proofs of Unsatisfiability for CNF Formulas'. In: *Proceedings of the Conference on Design, Automation, and Test in Europe (DATE'03)*. pp. 10886–10891.
19. Gregoire, E., B. Mazure, and C. Piette: 2007, 'Boosting a Complete Technique to Find MSSes and MUSes thanks to a Local Search Oracle'. In: *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, Vol. 2. Hyderabad (India), pp. 2300–2305.
20. Holzbaur, C., F. Menezes, and P. Barahona: 1996, 'Defeasibility in CLP(Q) through Generalized Slack Variables'. In: *Proceedings of the of the Second International Conference on Principles and Practice of Constraint Programming (CP'96)*. pp. 209–223.
21. Huang, J.: 2005, 'MUP: A Minimal Unsatisfiability Prover'. In: *Proceedings of the Tenth Asia and South Pacific Design Automation Conference (ASP-DAC'05)*. pp. 432–437.
22. Jackson, D.: 2002, 'Alloy: a lightweight object modelling notation'. *ACM Trans. Softw. Eng. Methodol.* **11**(2), 256–290.
23. Karp, R. M.: 1972, 'Reducibility among combinatorial problems'. In: *Complexity of Computer Computations*. Plenum Press, pp. 85–103.
24. Kavvadias, D. J. and E. C. Stavropoulos: 1999, 'Evaluation of an Algorithm for the Transversal Hypergraph Problem'. In: *Proceedings of the 3rd Workshop on Algorithm Engineering (WAE'99)*. pp. 72–84.
25. Kavvadias, D. J. and E. C. Stavropoulos: 2005, 'An Efficient Algorithm for the Transversal Hypergraph Generation'. *Journal of Graph Algorithms and Applications* **9**(2), 239–264.
26. Liffiton, M. H., M. D. Moffitt, M. E. Pollack, and K. A. Sakallah: 2005, 'Identifying Conflicts in Overconstrained Temporal Problems'. In: *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI'05)*. pp. 205–211.
27. Liffiton, M. H. and K. A. Sakallah: 2005, 'On Finding All Minimally Unsatisfiable Subformulas'. In: *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT-2005)*, Vol. 3569 of *LNCS*. pp. 173–186.
28. Lynce, I. and J. Marques-Silva: 2004, 'On Computing Minimum Unsatisfiable Cores'. In: *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT-2004)*, Vol. 3542 of *LNCS*. pp. 305–310.
29. Mneimneh, M. N., I. Lynce, Z. S. Andraus, J. P. M. Silva, and K. A. Sakallah: 2005, 'A Branch-and-Bound Algorithm for Extracting Smallest Minimal Unsatisfiable Formulas'. In: *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT-2005)*. pp. 467–474.
30. Oh, Y., M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, and I. L. Markov: 2004, 'AMUSE: a minimally-unsatisfiable subformula extractor'. In: *Proceedings of the 41st Annual Conference on Design Automation (DAC'04)*. pp. 518–523.

31. Safarpour, S., M. Liffiton, H. Mangassarian, A. Veneris, and K. Sakallah: 2007, 'Improved Design Debugging using Maximum Satisfiability'. In: *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD'07)*. In press.
32. Sinz, C., 'SAT benchmarks from Automotive Product Configuration'. Website. `http://www-sr.informatik.uni-tuebingen.de/~sinz/DC/`.
33. Sinz, C.: 2005, 'Towards an Optimal CNF Encoding of Boolean Cardinality Constraints'. In: *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP'05)*. pp. 827–831.
34. Sinz, C., A. Kaiser, and W. Küchlin: 2003, 'Formal methods for the validation of automotive product configuration data'. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* **17**(1), 75–97.
35. Zhang, L. and S. Malik: 2003, 'Extracting Small Unsatisfiable Cores from Unsatisfiable Boolean Formula'. Presented at the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT-2003).