A Branch and Bound Algorithm for Extracting Smallest Minimal Unsatisfiable Subformulas

Mark Liffiton[1] Maher Mneimneh[1] Inês Lynce[2]
Zaher Andraus[1] João Marques-Silva[3]
Karem Sakallah[1]

[1] University of Michigan
{liffiton,maherm,zandrawi,karem}@umich.edu

[2] Technical University of Lisbon, Portugal
ines@sat.inesc-id.pt

[3] University of Southampton
jpms@ecs.soton.ac.uk

July 28, 2008

Abstract

Explaining the causes of infeasibility of Boolean formulas has practical applications in numerous fields, such as artificial intelligence (repairing inconsistent knowledge bases), formal verification (abstraction refinement and unbounded model checking), and electronic design (diagnosing and correcting infeasibility). Minimal unsatisfiable subformulas (MUSes) provide useful insights into the causes of infeasibility. An unsatisfiable formula often has many MUSes. Based on the application domain, however, MUSes with specific properties might be of interest. In this paper, we tackle the problem of finding a smallest-cardinality MUS (SMUS) of a given formula. An SMUS provides a succinct explanation of infeasibility and is valuable for applications that are heavily affected by the size of the explanation. We present (1) a baseline algorithm for finding an SMUS, founded on earlier work for finding all MUSes, and (2) a new branch-and-bound algorithm called Digger that computes a strong lower bound on the size of an SMUS and splits the problem into more tractable subformulas in a recursive search tree. Using two benchmark suites, we experimentally compare Digger to the baseline algorithm and to an existing incomplete genetic algorithm approach. Digger is shown to be faster in nearly all cases. It is also able to solve far more instances within a given runtime limit than either of the other approaches.

1 Introduction

Explaining the causes of infeasibility of Boolean formulas has practical applications in numerous fields, such as electronic design, formal verification, and artificial intelligence. In design applications, for example, a large Boolean function is formed such that a feasible design is obtained when the function is satisfiable, and design infeasibility is indicated when the function is unsatisfiable. An example of this is the routing of signal wires in a field-programmable gate array (FPGA) [26]. A solver returning "Unsatisfiable," without further analysis of the causes of unsatisfiability, provides no clue as to what design constraints must be modified to make the design feasible.

In formal verification, certain implementations [2, 3, 19] of the counterexample guided abstraction refinement (CEGAR) paradigm [21] utilize infeasibility analysis. In CEGAR, abstraction is used to derive a compact model from a given concrete model (e.g., a circuit), and a property of interest is verified on the abstract model. If the property can be violated on the abstract model, any counterexample must be checked for feasibility, as infeasible counterexamples may arise due to the loss of information inherent in most abstraction procedures. Pinpointing the infeasibility identifies the relevant difference between the abstract and concrete models w.r.t to the counterexample; this difference is used to refine the abstract model and eliminate the infeasible counterexample. Several CEGAR-based applications have used infeasibility analysis, some employing bit-level Boolean formulations (e.g., [2] and [19]) and others (e.g. [3]) utilizing word-level formulations such as Satisfiability Modulo Theories (SMT). In artificial intelligence, local inconsistencies of a knowledge base might arise after augmenting it with new information. Identifying and fixing the causes of such inconsistencies is required before the knowledge base can be used.

Unsatisfiable formulas often contain a great deal of information that is extraneous to their infeasibility; a "minimal" explanation of the infeasibility that excludes irrelevant information is useful in many applications. The definition of minimality varies among application domains. For Boolean formulas in conjunctive normal form (CNF), a useful notion of minimality is the following. Consider an unsatisfiable CNF formula φ. An unsatisfiable subformula (US) of φ is a *minimal unsatisfiable subformula* (MUS) if it becomes satisfiable whenever any of its clauses is removed. An unsatisfiable formula could have many MUSes, and, depending on the application domain, we might be interested in just those with certain properties.

For example, in identifying inconsistent parts of a knowledge base, we may be interested in finding the MUS with the smallest number of clauses. Since such an MUS is a succinct description of one inconsistency in the knowledge base, it can be of great value for the repairing process. In hardware verification applications, the quality of refinement affects the number of iterations in the abstraction-refinement flow [2, 3]. While a US represents a set of spurious behaviors (states or transitions), an MUS represents a larger set of spurious behaviors because it contains fewer constraints. For example, a verification flow could generate the US "$x + 1 = x$ AND $x = 3$," while an MUS of the system

is more general: "$x + 1 = x$." Thus, MUSes are more effective at reducing the number of refinement steps required.

In this paper, we tackle the problem of finding a smallest cardinality MUS (SMUS) of a given formula. We present two approaches we have developed to solve this problem. The first is a modification of an existing algorithm that was developed for computing all MUSes. Second, we present a new algorithm that utilizes similar techniques but splits the problem into small, tractable subformulas and employs a recursive branch-and-bound approach on these subproblems to increase efficiency. We compare these two approaches with each other and with a third algorithm from the literature that uses incomplete local search to find small, often minimum USes.

This paper is organized as follows. In Section 2, we introduce basic definitions and some of the concepts underlying our work. Section 3 reviews previous work in the area, including a description of an existing algorithm for approximating SMUSes using local search. We present our algorithms, the modification of an existing system as well as our new branch-and-bound approach, in Sections 4 and 5. Finally, we present empirical results and analysis in Section 6 and conclude in Section 7.

2 Preliminaries

This work is focused on finding SMUSes of Boolean formulas in conjunctive normal form (CNF), though later we briefly discuss how to apply our techniques to other forms of constraint systems as well. Throughout these definitions and the entire paper, we will use "minimum" and "maximum" to describe sets that are the smallest and largest (in terms of cardinality) with some defining property and "minimal" and "maximal" to describe sets that cannot be reduced or enlarged without losing some defining property.

2.1 Boolean Formulas and Satisfiability

Formally, a CNF formula φ is defined as follows:

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

where each *literal* a_{ij} is either a positive or negative instance of some Boolean variable (e.g., x_3 or $\neg x_3$, where the domain of x_3 is $\{0, 1\}$), the value 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. We will often treat CNF formulas as sets of clauses, so equivalently:

$$\varphi = \bigcup_{i=1\ldots n} C_i$$

A CNF instance is said to be *satisfiable* (SAT) if there exists some assignment to its variables that makes the formula evaluate to 1 or TRUE; otherwise, we call it *unsatisfiable* (UNSAT). The problem of deciding whether a given CNF instance is satisfiable is the canonical NP-Complete problem to which many other combinatorial problems can be polynomially reduced. A *SAT solver* evaluates the satisfiability of a given CNF formula and returns a satisfying assignment of its variables if it is satisfiable. The associated problem of MaxSAT is an optimization problem for which the goal is to find a satisfiable subset of clauses with maximum cardinality.

The following unsatisfiable CNF instance φ will be used as an example throughout this paper. We use a shorthand for representing CNF wherein we omit the conjunction operators ($\wedge$) between clauses, implicitly specifying the conjunction as is common with multiplication in arithmetic.

$$\varphi = (x_1 \vee x_2)(\neg x_1 \vee x_2)(\neg x_2 \vee x_3)(\neg x_2 \vee \neg x_3)(x_2 \vee x_4)(\neg x_4 \vee \neg x_5)(\neg x_4 \vee x_5)$$
$$(x_4 \vee x_6 \vee x_7)(x_4 \vee x_6 \vee \neg x_7)(x_4 \vee \neg x_6 \vee x_7)(x_4 \vee \neg x_6 \vee \neg x_7)$$

We will refer to individual clauses as C_i, where i refers to the position of the clause in the formula (e.g., $C_3 = (\neg x_2 \vee x_3)$).

2.2 MUSes and MCSes

The definition of a *Minimal Unsatisfiable Subformula* (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 clauses of an unsatisfiable formula that is unsatisfiable and cannot be made smaller without becoming satisfiable. An MCS is a subset of the clauses of an unsatisfiable formula whose removal from that formula results in a satisfiable formula ("correcting" the infeasibility) and that is minimal in the same sense that any proper subset does not have that defining property. Any unsatisfiable formula can have multiple MUSes and MCSes, potentially exponential in the number of constraints (see Appendix A for examples). Formally, given an unsatisfiable formula φ, its MUSes and MCSes are defined as follows:

Definition 1. Given an unsatisfiable formula φ, a set of clauses $U \subseteq \varphi$ is an MUS if U is unsatisfiable and $\forall C_i \in U,\ U - \{C_i\}$ is satisfiable.

Definition 2. Given an unsatisfiable formula φ, a set of clauses $M \subseteq \varphi$ is an MCS if $\varphi - M$ is satisfiable and $\forall C_i \in M,\ \varphi - (M - \{C_i\})$ is unsatisfiable.

Any MaxSAT solution will provide an MCS; the clauses not satisfied by a particular MaxSAT solution are an MCS. However, not all MCSes correspond with MaxSAT solutions, because MaxSAT is concerned with maximum cardinality only, while MCSes are defined in terms of irreducibility (inaugmentability of the complementary satisfiable set of clauses), which includes minimum cardinality correction sets as well as others of larger size.

Our example formula φ and its three MUSes are depicted visually in Figure 1. The MUSes are also listed below (in general, we will use MUSes(φ) and

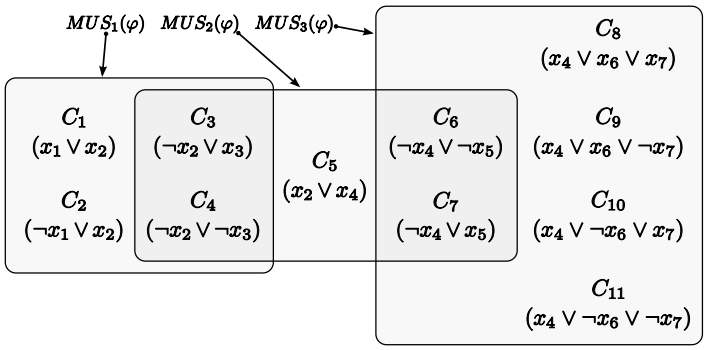

Figure 1: The example formula φ and its MUSes

MCSes(φ) to refer to the complete collections of each subset type for any given formula φ). Additionally, φ has 24 MCSes, not shown here: 16 with two clauses and 8 with three. Therefore, it also has 16 MaxSAT solutions, each of which satisfies all but two clauses.

$$\begin{aligned}
\text{MUSes}(\varphi) &= \{\{C_1, C_2, C_3, C_4\}, \{C_3, C_4, C_5, C_6, C_7\}, \{C_6, C_7, C_8, C_9, C_{10}, C_{11}\}\} \\
\text{MUS}_1(\varphi) &= (x_1 \vee x_2)(\neg x_1 \vee x_2)(\neg x_2 \vee x_3)(\neg x_2 \vee \neg x_3) \\
\text{MUS}_2(\varphi) &= (\neg x_2 \vee x_3)(\neg x_2 \vee \neg x_3)(x_2 \vee x_4)(\neg x_4 \vee \neg x_5)(\neg x_4 \vee x_5) \\
\text{MUS}_3(\varphi) &= (\neg x_4 \vee \neg x_5)(\neg x_4 \vee x_5)(x_4 \vee x_6 \vee x_7) \\
&\quad (x_4 \vee x_6 \vee \neg x_7)(x_4 \vee \neg x_6 \vee x_7)(x_4 \vee \neg x_6 \vee \neg x_7)
\end{aligned}$$

The algorithms discussed in this paper rely on an important connection between MUSes and MCSes noted in [17] and [5] and exploited in [4] and [22, 23] for finding all MUSes of constraint systems. Specifically, every MUS of a formula φ is a *minimal hitting set* of the complete set of φ's MCSes. A hitting set of a collection of sets C is a set that contains at least one element from every set in C. A minimal hitting set is a hitting set such that every one of its elements is the sole representative for at least one set in C; in other words, removing any one element would cause it to lose the "hitting set" property. So in the case of the MUS/MCS link, every MUS contains at least one clause from every MCS, and the minimality of the MUS mirrors the minimality of the fact that it is a minimal hitting set. This connection is useful in that it provides a method to produce all MUSes of a formula: first find all MCSes, then compute all minimal hitting sets of the MCSes.

To understand this connection more intuitively, note first that the presence of an MUS in a constraint system C makes C infeasible. One must somehow "neutralize" every MUS in C to produce a satisfiable system. Because it is minimal, an MUS can be neutralized, or made satisfiable, by removing any one constraint from it. Therefore, an MCS of C, whose removal "corrects" the system (making it satisfiable), must contain at least one constraint from every MUS in C: an MCS of C is a hitting set of all MUSes of C. Further, an MCS

is a *minimal* hitting set of the MUSes, as its minimality (remove one constraint and it no longer corrects the infeasibility because it no longer hits one of the MUSes) mirrors that of a minimal hitting set of MUSes. Every constraint in either an MCS or a minimal hitting set of the MUSes is necessary in that it hits some MUS that the other constraints in the set do not. A similar argument in the other direction can show how MUSes are minimal hitting sets of the MCSes of a system, though it is not as intuitive.

In addition to MUSes and MCSes, we will refer to unsatisfiable subsets (USes) and correction sets (CSes), which are simply MUSes and MCSes, respectively, without the criterion of minimality. As an example, given any unsatisfiable formula, the entire formula is both a US and a CS.

2.3 SMUSes

The target of the algorithms in this paper is a *Smallest* Minimal Unsatisfiable Subformula (SMUS).

Definition 3. Given an unsatisfiable formula φ, a set of clauses $S \subseteq \varphi$ is an SMUS if S is an MUS of φ and φ has no MUSes with fewer clauses than S.

The example formula has a single SMUS: $\text{MUS}_1(\varphi) = \{C_1, C_2, C_3, C_4\}$. With 4 clauses, it is smaller than the other two, with 5 and 6 clauses respectively. In general, a formula may have multiple SMUSes with equal cardinality. We use $|\text{SMUS}(\varphi)|$ to refer to the size of any SMUS of φ.

3 Previous Work

The previous work in the area of minimal unsatisfiable subformulas has generally been divided into that which is more theoretical, related to complexity bounds, etc., and that which is more applied, developing and implementing algorithms to be investigated experimentally.

In some of the earliest theoretical work related to MUSes, Papadimitriou and Wolfe [28] showed that recognizing a minimal unsatisfiable formula (i.e., is φ an MUS?) is D^P-complete. A D^P-complete problem is equivalent to solving a SAT-UNSAT problem defined as: given two formulas ϕ and ψ, in CNF, is it the case that ϕ is satisfiable and ψ is unsatisfiable? This result implies that algorithms for computing MUSes will require superpolynomial time.

Certain subclasses of unsatisfiable formulas do have tractable solutions to these problems, however. Minimally unsatisfiable formulas always have positive deficiency [1, 10], where deficiency is the difference between the number of clauses and variables. Davydov et al. [10] gave an efficient algorithm for recognizing minimal unsatisfiable formulas with deficiency 1. Büning [7] showed that if k is a fixed integer then the recognition problem with deficiency k is in NP, and he suggested a polynomial time algorithm for formulas with deficiency 2. Kullmann [20] proved that recognizing a minimal unsatisfiable formula with deficiency k is decidable in polynomial time, and Fleischner et al. [12] showed

that such formulas can be recognized in $n^{O(k)}$ time, where n is the number of variables. Szeider [31] improved this result and presented an algorithm with time complexity $O(2^k n^4)$.

The concept of deficiency has been applied to quantified Boolean formulas (QBFs) also; Büning and Zhao [8] present a definition of *minimal false formulas*. A minimal false formula is a QBF in CNF that is false but becomes true when any clause is removed. A notion of deficiency for minimal false formulas is defined and it is shown that any minimal false formula must have positive deficiency. In [8], a polynomial time algorithm for deciding minimal falsity for formulas with deficiency 1 was presented.

Dasgupta and Chandru [9] classify MUSes into three categories based on properties of their literals. They also present upper bounds on the sizes of MUSes for 2-CNF and 3-CNF formulas. For a 2-CNF formula, an m-literal MUS has at most $O(2m)$ clauses, and for a 3-CNF formula, an m-literal MUS has at most $O(1.62^{m-1})$ clauses.

In one of the earliest implemented algorithms for computing small unsatisfiable subformulas, Bruni and Sassano [6] employ an "adaptive core search" procedure that ranks clauses based on their hardness. The hardness of a clause is defined as a weighted sum of how often the clause is visited during a complete search algorithm and how often it is involved in conflicts. Starting from a small initial set of hard clauses, the unsatisfiable core is built by an iterative process that expands or contracts the current core by a fixed percentage of clauses (chosen based on hardness) until the core becomes unsatisfiable. The quality (i.e., size and minimality) of the unsatisfiable core produced by this procedure is highly dependent on the particular settings of three parameters that guide the process.

Other approaches to producing unsatisfiable cores have arisen from work aiming primarily to verify "unsatisfiable" results returned by SAT solvers. They share a common theme of identifying an unsatisfiable core by distinguishing those clauses involved in the proof of unsatisfiability. These techniques also do not guarantee minimality, and their outputs are often not minimal. Goldberg and Novikov [14] record conflict clauses during a SAT solver's search and verify each using Boolean constraint propagation. By verifying the conflict clauses in reverse chronological order and only checking those that are needed to form previously checked clauses, the process can identify a subset of the original clauses needed to form the final conflict; such a subset is an unsatisfiable core.

Zhang and Malik [33, 34] use a resolution proof generated by the SAT solver and use it to derive an unsatisfiable core in a similar way. A resolution proof is a directed acyclic graph (DAG) whose leaf vertices correspond to the formula's original clauses. Every other vertex has exactly two predecessors and corresponds to the consensus clause obtained from those of its predecessors. The set of original clauses in the transitive fan-in of the root (an empty clause representing the final conflict) are returned as the unsatisfiable core. For any of these procedures that produce non-minimal cores, smaller cores, though not necessarily MUSes, can be obtained by repeatedly applying the procedure until no further reduction in size is obtained.

Oh et al. [27] instrument a CNF formula with clause-selector variables: for any clause C_i, make it $C'_i = \neg y_i \vee C_i$; this allows a SAT solver to enable and disable clauses by assigning TRUE or FALSE, respectively, to any particular y_i. In their algorithm, AMUSE, they utilize information from a DPLL-style search to implicitly search for a US instead of a satisfying assignment of the original formula. They note that AMUSE can be biased to favor particular variables, enabling them to find multiple USes if they favor variables that did not appear in previously found USes.

Gershman et al. [13] extend the approach in [33, 34] to produce smaller cores. They use the concept of "dominators" in resolution proofs to look for reformulations of the graph with fewer leaf nodes (original clauses). Their algorithm, Trimmer, can produce smaller cores than Zhang and Malik's algorithm alone (though the results are still not necessarily minimal).

Huang [18] converts the problem of checking whether a formula is an MUS to a model counting problem on a new formula obtained by augmenting the original formula with "minterm" selector variables. Binary decision diagrams are used to perform model counting on the original formula. This approach can also be used to minimize unsatisfiable cores to proper MUSes.

Grégoire et al. [15] apply local search to the problem of computing MUSes. They employ a scoring heuristic based on clauses' relations to others with which they share literals to identify which clauses are more or less likely to be included in some MUS. Scores are recorded within a local search, and clauses deemed unlikely to participate in an MUS are removed. The last unsatisfiable set of clauses (before removing the final clause which makes it satisfiable) is an approximate MUS (unsatisfiable core). The authors extend this approach to 1) compute one MUS exactly (using a procedure that "fine tunes" the approximation), 2) compute "strict inconsistent covers" (sets of MUSes that share no clauses with one another), and 3) approximate the set of all MUSes (relying on removing a clause from the formula to find each subsequent MUSes, and thus finding at most a number of MUSes linear in the size of the formula, as compared to the potentially exponential number of MUSes present).

Two systems for computing all MUSes of a constraint system exactly have been developed, both based on the connection between MCSes and MUSes noted in Section 2.2. Bailey and Stuckey [4] developed an algorithm for finding all MUSes of a constraint system and applied it to the problem of type-error diagnosis in software verification. They employ an interleaved approach, computing MUSes as MCSes are found. The other approach was developed by Liffiton et al. [22, 23] as a serial, two-phase algorithm called CAMUS (for Compute All Minimal Unsatisfiable Subsets). CAMUS is explained in detail in Section 4. In [22], the authors performed an experimental comparison of Bailey and Stuckey's approach, adapted to Boolean satisfiability, with CAMUS, finding that CAMUS was consistently faster by several orders of magnitude on those instances.

For the focus of this paper, computing an SMUS of a Boolean satisfiability instance, there are two previous approaches. Lynce et al. [24] presented an algorithm that computes an SMUS by implicitly searching the space of all unsatisfiable subformulas. This is achieved by adding "selector variables," called

S variables, and extending a DPLL-like SAT solver to search for an assignment that makes the formula unsatisfiable while minimizing the number of S variables set to 1; such an assignment corresponds to an SMUS. While this approach implicitly searches the space of the USes to find the SMUS, the algorithm we describe in this paper implicitly searches the space of the MUSes, which can be exponentially smaller, and employs several techniques to prune this space during the search.

Zhang et al. [32] developed an approach to compute an SMUS using the first phase of CAMUS coupled with a greedy genetic algorithm (GGA) to find small MUSes. They generate all MCSes of an instance using the first phase of CAMUS, thus facing the same intractability issues as CAMUS in that phase. If it can generate all MCSes, then GGA uses a greedy construction to produce seeds for a genetic algorithm (GA) that encodes sets of clauses as genomes and utilizes the standard GA approaches of mutations, crossover, etc. to search for a minimum hitting set of the MCSes. Because the genetic algorithm is an incomplete local search, GGA can not guarantee the minimality of the result; in practice it returns either an SMUS or a small US whose size is within a few percent of the number of clauses in an SMUS. We experimentally compare our algorithm to GGA in Section 6.

4 Computing an SMUS with CAMUS

One of the simplest approaches to finding an SMUS of a formula is to generate all of its MUSes and then select one with the fewest clauses. However, the complete set of MUSes can be intractably large, as the number of MUSes may be exponential in the size of the original formula. While this makes the approach intractable in the general case, it does provide a useful baseline against which the performance of other algorithms can be compared. Furthermore, our branch-and-bound algorithm utilizes this approach on *subsets* of the input formula, with the idea being that these subsets will be small enough that the exhaustive approach will finish in a reasonable amount of time.

The baseline algorithm we use for these purposes is a variant of the CAMUS algorithm presented in [23], modified to use branch-and-bound to avoid some of the intractability of generating all MUSes. CAMUS exploits the connection between MCSes and MUSes described in Section 2.2 to generate all MUSes of a given formula. Because satisfiable subsets of constraints are easier to find than unsatisfiable subsets, in the sense that problems in NP are easier than those in co-NP, CAMUS operates in two phases: 1) compute all MCSes of a given formula, 2) compute all MUSes of the formula by finding all minimal hitting sets of the MCSes in a recursive tree. The first phase is unchanged, but we have modified the second phase in this work. Instead of computing all minimal hitting sets (MUSes), we added a branch-and-bound capability to the recursion tree to prune large portions of the tree and produce only the smallest hitting set (an SMUS). We call this variant CAMUS-min.

4.1 Finding MCSes

The first phase of both CAMUS and CAMUS-min utilizes a standard SAT solver to find MCSes. Every clause C_i in φ is augmented with a *clause-selector variable* y_i to produce $C'_i = (y_i \rightarrow C_i) = (\neg y_i \vee C_i)$. An assignment of TRUE to a particular y_i implies the original clause C_i, essentially enabling or *hardening* that clause. Assigning FALSE to y_i likewise disables C_i by satisfying C'_i. The augmented clauses, conjoined, form φ'. Any satisfying solution to φ' thus indicates a satisfiable subset of φ by the set of y_i variables assigned TRUE. A satisfying solution with a maximal set of y_i variables assigned TRUE describes an MCS; the y_i variables assigned FALSE correspond to the clauses left unsatisfied or "removed," forming an MCS.

To find MCSes, we iteratively solve φ' with additional constraints that maximize the number of y_i variables assigned TRUE and blocking clauses that require at least one clause from every MCS found thus far be enabled. For example, to block the MCS $\{C_4, C_9, C_{17}\}$, the clause $(y_4 \vee y_9 \vee y_{17})$ is added to φ', requiring that no future solution disables all three of those clauses at once. This procedure finds all MCSes of a formula in increasing order of size, halting when φ' along with the blocking clauses is unsatisfiable. For complete details please refer to [23].

4.2 Computing (S)MUSes from MCSes

The algorithm in the second phase of CAMUS recursively generates all MUSes from the set of all MCSes produced in the first phase. At every recursive step, it selects a clause from the MCSes to include in a growing MUS and an MCS in which it appears. It then alters the remaining MCSes to remove any others that include that clause and to remove any clauses in the chosen MCS from other MCSes. The alterations ensure that no further choices would make that clause redundant within the constructed MUS. At every step, a clause and an MCS in which it occurs can be selected arbitrarily from the remaining set to produce different MUSes, and thus the algorithm branches on all such choices to recursively generate all MUSes.

For CAMUS-min, we calculate a lower bound on the size of the smallest MUS that can be constructed below any node by summing the number of clauses chosen above the node with the size of an approximation of the maximal independent set (MIS) of the remaining altered MCSes. Every node in the recursion tree is operating on a set of sets, either the complete set of MCSes in the root node or some smaller set of altered MCSes in the other nodes. An MIS of the (potentially altered) MCSes will be pairwise disjoint and thus the number of sets it contains is a lower bound on the number of clauses that must still be selected to hit all remaining MCSes. The approximation we use is a greedy heuristic called **MIS-quick** [16] that iteratively selects the smallest remaining set and removes any other sets that intersect it. It finishes when no sets remain, all having been removed due to either selection or intersection with a selected set.

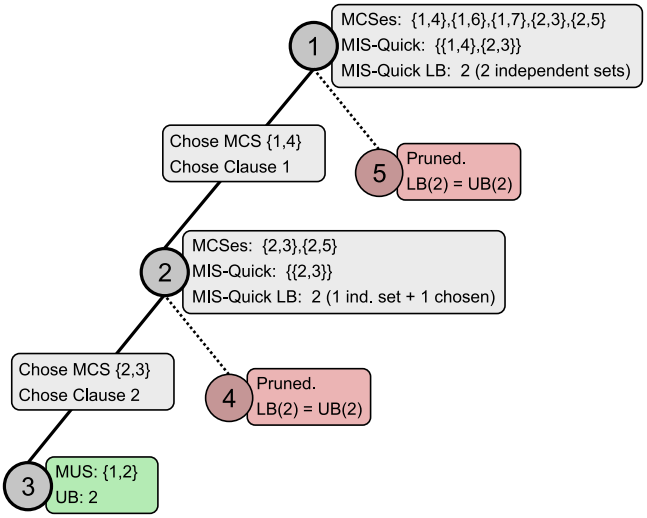

Figure 2: The operation of the recursive second phase of CAMUS-min using **MIS-quick** to compute lower bounds and prune subtrees.

Figure 2 illustrates an example of using this lower bound to prune portions of the recursive tree and return an SMUS. (We have not used the running example from Figure 1 because it would be too large.) In this example, the algorithm is given the set of MCSes $\{\{1,4\},\{1,6\},\{1,7\},\{2,3\},\{2,5\}\}$ (using "1" as shorthand for C_1, for example). The **MIS-quick** routine could return two independent sets at this node $\{\{1,4\},\{2,3\}\}$, indicating that the lower bound on the size of any MUS is 2. In the first branch, the MCS $\{1,4\}$ and clause 1 are chosen. The MCSes are altered by discarding those that contain 1; removing the other clause in the chosen MCS, 4, from those that remain; and removing any altered MCSes that are now supersets of another; thus node 2 of the recursive tree has the altered MCSes $\{2,3\}$ and $\{2,5\}$. Here, **MIS-quick** will return just a single set (e.g., $\{2,3\}$) and so the lower bound is still 2: one clause chosen in this path plus one independent set.

The next choice, of clause 2 and MCS $\{2,3\}$, results in an empty set of MCSes in node 3, thus the chosen clauses along this path are an MUS: $\{1,2\}$. The upper bound is set to its size, 2, and the algorithm backtracks. When it returns to node 2, the upper bound is equal to the lower bound estimated from **MIS-quick**, so any further branches below the node are pruned. Backtracking to node 1 produces a similar result, as the lower bound there is again equal to the upper bound. The algorithm terminates, having found an SMUS, $\{1,2\}$, and having pruned any subtrees in which an equal-sized or larger MUS would have been found.

With this lower bound provided by **MIS-quick**, CAMUS-min can prune any branches of the recursion tree that are proven to contain no MUSes smaller than the smallest found thus far. This prunes out large portions of the tree,

decreasing runtime substantially, and the last MUS produced will be an SMUS. The pruning induced by the lower bound does greatly decrease the runtime, but only for the second phase of CAMUS-min. The first phase is unaffected, and it still must generate all MCSes of the formula before the second phase can commence. This can be intractable, because the number of MCSes can be exponential in the size of the formula.

5 Computing an SMUS with Digger

As noted, any approach to computing an SMUS that generates all MUSes or even just all MCSes of a formula can hit severe scalability issues, as both of these sets can be exponentially large in the worst case. The algorithm we describe here (first presented in [25]) attempts to avoid this intractability with a branch-and-bound approach that breaks the problem into smaller, more tractable subproblems and uses a robust lower bound along with the standard "best so far" upper bound to significantly reduce the amount of work done in finding an SMUS. While the algorithm still searches the complete space of all MUSes, it is more efficient than existing alternatives due to these heuristics. We call this algorithm Digger.

The execution of Digger on our running example is shown in its entirety in Figure 6. We have placed it later because it is only fully explained by the combined contents of the following four subsections (Sections 5.1 through 5.4), but we will refer to relevant pieces of it throughout this section to illustrate the example. In the figure, we have used numbers in place of C_i for ease of reading; for example, 4 is used in place of C_4.

5.1 Disjoint MCSes

The operation of Digger's branching as well as its lower bound calculations come from a heuristic of finding *disjoint MCS covers*.

Definition 4. A disjoint MCS cover D for a formula φ is a subset of φ's MCSes such that (1) the intersection of any two MCSes in D is empty, and (2) the conjunction of all clauses in the MCSes in D, which we will call φ_D, is unsatisfiable (the MCSes in D together "cover" an entire MUS).

One can equivalently define a *disjoint CS cover* by simply removing the minimality criterion for the correction sets in the above definition (replace "MCS" by "CS").

5.1.1 Properties of Disjoint MCS Covers

A disjoint MCS cover D and its corresponding φ_D have several interesting and useful properties. First, because it is unsatisfiable, φ_D must contain at least one MUS, and, as it is a subset of φ, any MUSes of φ_D are also MUSes of φ. Secondly, D is easy to compute, relative to computing the complete set of

MCSes of φ. In the worst case, D is linear in size of the formula, which contrasts with the potentially exponential number of MCSes in φ. Every MCS in D must contain at least one clause, and it cannot contain any clauses in any other MCS in D; thus, there can be at most 1 MCS in D for every clause in φ.

A third useful property of D is stated in the following theorem:

Theorem 1. *Given a disjoint MCS cover D for a formula φ:*

$$|\text{SMUS}(\varphi)| \geq |D|$$

Proof. Any MUS in φ must contain at least one clause from every MCS of φ. Thus, any MUS in φ must contain at least one clause from every MCS in D. Because any two MCSes in D are disjoint, there is no way to select a clause that covers more than one MCS; therefore, any MUS in φ must contain at least $|D|$ clauses. □

This theorem states that $|D|$ is a lower bound on the size of SMUS(φ). This lower bound can be used to prove that a candidate SMUS is the smallest possible, as described in more detail in Section 5.3. This theorem is also valid for a disjoint CS cover (notice that the proof of the theorem does not rely on the minimality of the MCSes in any way), and we will make use of this fact briefly when discussing optimizations in Section 5.5.

5.1.2 Generating a Disjoint MCS Cover

A disjoint MCS cover can be generated by a procedure very similar to that used to find *all* MCSes in CAMUS (described in Section 4). The only difference is that instead of creating a blocking clause for each MCS that prevents finding the same MCS again, we block each clause in the MCS separately. For example, given that we find an MCS $\{C_4, C_9, C_{17}\}$, we create three new blocking clauses: (y_4), (y_9), and (y_{17}). This *hardens* the three clauses independently, requiring that they always be satisfied in future solutions. With these clauses hardened, any additional MCS found can not intersect any found previously. Eventually, these hard constraints will altogether include an MUS, at which point they will be unsatisfiable and the MCSes collected thus far will be a disjoint MCS cover. (Because of this similarity between the two approaches, we were able to use, with that one minor change, the same framework in our implementation of Digger as was used in CAMUS and CAMUS-min.)

A more generic way of understanding this procedure is to think of it in terms of MaxSAT. An initial MCS can be found by solving MaxSAT for the unsatisfiable formula φ. The clauses not satisfied by the solution are an MCS. Then, further MCSes are found by iterative solutions to MaxSAT for φ with the clauses in all MCSes found so far hardened. This will force the solutions to leave only *new* clauses unsatisfied, so the new MCSes found will not intersect any found earlier. Figure 3 shows pseudocode for this approach in terms of MaxSAT. Here, **MaxSAT**(φ,`hardened`) returns the set of clauses left unsatisfied when satisfying as many clauses in φ as possible given that those in `hardened` must be satisfied.

DisjointMCSes(φ)

1. `hardened` $\leftarrow \emptyset$
2. **while** (**isSAT**(`hardened`))
3. `newMCS` $\leftarrow$ **MaxSAT**(φ,`hardened`)
4. `hardened` $\leftarrow$ `hardened` $\cup\{$`newMCS`$\}$
5. **return** `hardened`

Figure 3: An algorithm for finding a disjoint MCS cover of φ

On our running example formula in Figure 6, execution of this algorithm could first find any one of the several 2-clause MCSes in φ (each of the 16 MaxSAT solutions for φ satisfies all but 2 clauses). For example, it could find $\{C_3, C_7\}$, which is then added to `hardened`. Those two clauses are satisfiable, so the loop iterates, this time finding a MaxSAT solution where C_3 and C_7 are satisfied. The MCS returned in this case is $\{C_4, C_6\}$. Now, there are no MaxSAT solutions that satisfy all but 2 clauses *and* satisfy C_3, C_4, C_6, and C_7, but there are ways to satisfy all of these while leaving 3 clauses unsatisfied. So **MaxSAT** returns $\{C_1, C_5, C_8\}$. At this point, the clauses accumulated in hardened become unsatisfiable, because they contain $\text{MUS}_2(\varphi) = \{C_3, C_4, C_5, C_6, C_7\}$, and the loop terminates. The subroutine returns the disjoint MCS cover it found: $D = \{\{C_3, C_7\}, \{C_4, C_6\}, \{C_1, C_5, C_8\}\}$.

5.2 Branching

Digger utilizes a recursive branch-and-bound tree to find an SMUS. Every node in the tree (each recursive call) takes as input a subset of the original formula's clauses and upper and lower bounds on the size of any SMUS within that subset. The output of a node (returned to the parent) is the smallest MUS found in that subset if any are found that are smaller than the upper bound. Therefore, running the algorithm consists of a single call to start the recursion at the root node with inputs of the entire formula and initial, unconstrained bounds.

Unlike a standard branch-and-bound tree, in which solutions are only found at the leaves, Digger investigates some subset of the MUSes in every node of the tree, including the root node. Within each node, it uses the disjoint MCS heuristic to select a sample set of clauses φ_D. As discussed above, φ_D contains at least one MUS. The CAMUS-min algorithm described in Section 4 is used to exhaustively generate all MCSes of φ_D and compute an SMUS. This SMUS of φ_D is then a candidate SMUS of φ.

In our running example in Figure 6, we have so far found a disjoint MCS cover $D = \{\{C_3, C_7\}, \{C_4, C_6\}, \{C_1, C_5, C_8\}\}$, so $\varphi_D = \{C_1, C_3, C_4, C_5, C_6, C_7, C_8\}$. φ_D contains only one MUS, which CAMUS-min finds as an SMUS. This MUS,

$\{C_3, C_4, C_5, C_6, C_7\}$, is now a candidate SMUS for φ.

To branch, Digger creates subformulas to pass to recursive calls below the current node. Ideally, these subformulas should:

1. all together contain every MUS not in the subset checked in the node;
2. not share MUSes with the subset investigated within the node;
3. not have any MUSes in common between themselves.

Only property 1 is required, because it ensures that the recursion will investigate every MUS in the formula passed to a node, either within the node itself or in one of the branches below it. Properties 2 and 3 are good for efficiency, as both avoid redundant work. The branching heuristic used in Digger has properties 1 and 2 but not 3, so it may investigate an MUS in more than one branch.

Digger forms a subformula for each branch by removing the clauses in one MCS of φ_D from φ. We will refer to the MCSes of φ_D as $\{\delta_1, \delta_2, \ldots, \delta_n\}$, so each subformula is some $\varphi - \delta_i$. Note that the MCSes of φ_D are not necessarily the same as the MCSes in D itself (which are MCSes of φ). Removing any MCS of φ from φ yields a satisfiable subformula by the definition of MCS, but removing an MCS of φ_D from φ could yield a still-unsatisfiable subformula with more MUSes to investigate.

Property 2 holds for these subformulas because removing any MCS of φ_D removes at least one clause from every MUS in φ_D from φ. This guarantees that no MUSes of φ_D will be present in $\varphi - \delta_i$.

To prove that this algorithm is complete, we must prove that every MUS in φ is contained in φ_D or at least one of $\varphi - \delta_i$. To do this, we will make use of the following theorem from [23], with a proof in [5]:

Theorem 2. *The set of all minimal hitting sets of the MCSes of a formula φ is equal to the set of all MUSes of φ.*

A corollary of this theorem is directly useful in our completeness proof:

Corollary 1. *Any hitting set of* $\mathrm{MCSes}(\varphi)$ *is an US of φ.*

This follows from the facts that every hitting set must contain a minimal hitting set and every US must contain at least one MUS.

Additionally, we make use of the following lemma:

Lemma 1. *Given $\vartheta = \varphi - \{c_1, c_2, \ldots, c_i\}$:*

$$\mathrm{MUSes}(\vartheta) = \{x : x \in \mathrm{MUSes}(\varphi) \wedge x \cap \{c_1, c_2, \ldots, c_i\} = \emptyset\}$$

This states that the MUSes of a formula ϑ formed by removing clauses from another, φ, will be a subset of the MUSes of φ: those that do not contain any of the removed clauses.

With these facts, we can prove the following theorem, which proves property 1 for Digger's branching heuristic:

Theorem 3. *Given a CNF formula φ, a subformula $\varphi_D \subseteq \varphi$, and the MCSes of φ_D $\{\delta_1, \delta_2, \ldots, \delta_n\}$:*

$$\text{MUSes}(\varphi) = \text{MUSes}(\varphi_D) \cup \bigcup_{i=1}^{n} \text{MUSes}(\varphi - \delta_i)$$

Proof. By contradiction: Assume there is an MUS $M \in \text{MUSes}(\varphi)$ contained in neither φ_D nor any of the $\varphi - \delta_i$ subformulas.

M must contain at least one clause from every δ_i, otherwise it would be contained in a $\varphi - \delta_i$ subformula, by Lemma 1. Therefore M must contain a hitting set of $\{\delta_1, \delta_2, \ldots, \delta_n\}$, the MCSes of φ_D.

By Corollary 1, a hitting set of $MCSes(\varphi_D)$ contains an MUS of φ_D. Therefore, M must contain an MUS of φ_D, and because M is an MUS itself, then M must *be* an MUS of φ_D, a contradiction. □

Therefore, we know that a recursive search that investigates all MUSes in φ_D and all $\varphi - \delta_i$ subformulas as described must investigate every MUS in φ.

In our running example, the MCSes of φ_D are $\{C_3\}$, $\{C_4\}$, $\{C_5\}$, $\{C_6\}$, and $\{C_7\}$. The recursion will thus branch from this node to explore $\varphi - C_3$, $\varphi - C_4$, and so on. In Figure 6, we have chosen to branch on $\{C_5\}$ first, so the second search node is passed $\varphi - C_5$.

5.3 Bounds

Upper and lower bounds on the size of the SMUSes are calculated for the current subproblem at every search node. The upper bound is the size of the smallest MUS found thus far in any previous node in the tree. The lower bound calculation is derived from the heuristic for selecting a subset of clauses. The number of disjoint MCSes found in the initial discovery of D to form φ_D is a lower bound on the size of the MUSes in φ, as proven in Theorem 1. Therefore, any MUS found of size $|D|$ must be an SMUS.

Furthermore, we can improve this bound slightly in some cases:

Theorem 4. *Given a disjoint MCS cover D for a formula φ and a subformula φ_D formed by conjoining the clauses in every MCS in D:*

$$\textit{If } |\text{SMUS}(\varphi_D)| \neq |D|, \textit{ then } |\text{SMUS}(\varphi)| \geq |D| + 1$$

Proof. By contrapositive: Assume the negation of the consequent of the implication, namely $|\text{SMUS}(\varphi)| < |D| + 1$.

From Theorem 1, we know that $|\text{SMUS}(\varphi)| \geq |D|$; therefore, given our assumption, we know that $|\text{SMUS}(\varphi)| = |D|$. Every MUS of φ must contain at least one clause from each MCS in D, thus every one of the $|D|$ clauses in any SMUS of φ must be in some MCS of D. For this to be true, these MUSes of φ, each of size $|D|$, must be contained in φ_D. Because $|\text{SMUS}(\varphi_D)| \geq |D|$ and φ_D contains an MUS of size $|D|$, it must be the case that $|\text{SMUS}(\varphi_D)| = |D|$. We have thus proven the contrapositive, so the original implication is true. □

Another way of stating this is the following: if φ has an MUS of size $|D|$, then that MUS must be contained in φ_D. We now have a lower bound on the size of an MUS of φ in the case where an SMUS of φ_D has more than $|D|$ clauses. This can be used in two ways: (1) When an SMUS of φ_D has size $|D|+1$, it must be an SMUS of φ as well; (2) If no SMUS of φ_D reaches the lower bound, then any MUS of a subformula $\varphi - \delta_i$ (see Theorem 3) of size $|D|+1$ is an SMUS of φ.

The running example shown in Figure 6 returns an MUS of size 4 from the second search node. Because this is one more than its parent's lower bound (4), the search can terminate immediately, having proven with Theorem 4 that it is an SMUS of φ.

5.4 The Algorithm

The pieces mentioned thus far are composed into the Digger algorithm shown in pseudocode in Figure 4. It is a recursive algorithm that operates on subsets of the original formula, returning an SMUS of its input formula φ if one exists that is smaller than the given upper bound `UB` and $\emptyset$ otherwise. The initial call to Digger is given the complete formula φ for which we wish to find an SMUS and an unrestrictive upper bound such as `UB` $= |\varphi| + 1$. The computations performed in each node and the resulting branching are shown in Figure 5. Figure 6 illustrates a possible run of Digger on the example formula φ.

First, Digger performs a simple satisfiability check, returning $\emptyset$ if φ is SAT. This is required because some of the formulas passed to the routine by recursive calls could be satisfiable, containing no MUSes. Continuing, the algorithm explores a subset of the MUSes of the current subproblem using D, a disjoint MCS cover of φ. First, the lower bound on $|SMUS(\varphi)|$ is calculated as $|D|$, and $\emptyset$ is returned if this proves that φ has no MUSes smaller than `UB` (the smallest size found earlier). If the current branch is not terminated due to `LB` $\geq$ `UB`, the algorithm further uses D to produce a subformula of φ, denoted φ_D, by taking the union of every clause in every MCS in D. This subformula is explored by using CAMUS-min to produce an SMUS of φ_D, and all MCSes of φ_D are generated in the process. The SMUS of φ_D is returned if it is proven to be the smallest by hitting the lower bound on size; otherwise, if it is an improvement, it is stored in `candidate` and its size is recorded as a new upper bound.

At this point, the algorithm has calculated a lower bound on the size of an SMUS of φ and explored a subset of its MUSes, producing a candidate SMUS. If the candidate has not been proven to be an SMUS, then the algorithm explores the remaining MUSes of φ by generating subformulas as described in Section 5.2 that taken together will contain all of the MUSes of φ not in φ_D. To do this, the algorithm generates one subformula for each MCS δ_i of φ_D: $\varphi - \delta_i$ (removing all clauses in δ_i from φ). Figure 5 illustrates this branching and the computations that produce it.

A recursive call to Digger passed $\varphi - \delta_i$ and an upper bound from the smallest MUS found thus far will produce an SMUS of $\varphi - \delta_i$ or $\emptyset$ if it has no MUSes smaller than the bound. An SMUS of $\varphi - \delta_i$ is a candidate for being an SMUS

Digger(φ, UB)

1. **if** (**isSAT**(φ))
2. **return** $\emptyset$
3. $D \leftarrow$ **DisjointMCSes**(φ)
4. LB $\leftarrow |D|$
5. **if** (LB $\geq$ UB)
6. **return** $\emptyset$
7. $\varphi_D \leftarrow \bigcup$ MCS : MCS $\in D$
8. (SMUS$_D$, allMCSes$_D$) $\leftarrow$ CAMUS-min(φ_D)
9. **if** (|SMUS$_D$| = LB **or** |SMUS$_D$| = LB + 1)
10. **return** SMUS$_D$
11. candidate $\leftarrow \emptyset$
12. **if** (|SMUS$_D$| < UB)
13. candidate $\leftarrow$ SMUS$_D$
14. UB $\leftarrow$ |SMUS$_D$|
15. **foreach** ($\delta_i \in$ allMCSes$_D$)
16. SMUS$_{branch}$ $\leftarrow$ Digger($\varphi - \delta_i$, UB)
17. **if** (SMUS$_{branch} \neq \emptyset$)
18. **if** (|SMUS$_{branch}$| = LB + 1)
19. **return** SMUS$_{branch}$
20. candidate $\leftarrow$ SMUS$_{branch}$
21. UB $\leftarrow$ |SMUS$_{branch}$|
22. **return** candidate

Figure 4: The recursive algorithm for finding an SMUS of a formula φ

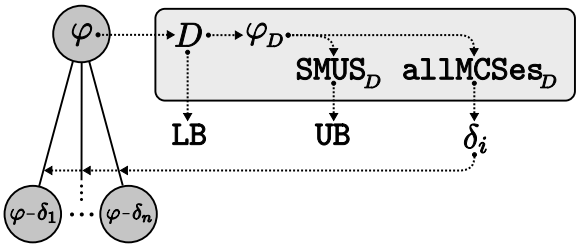

Figure 5: An illustration of the computation and branching of one call to Digger

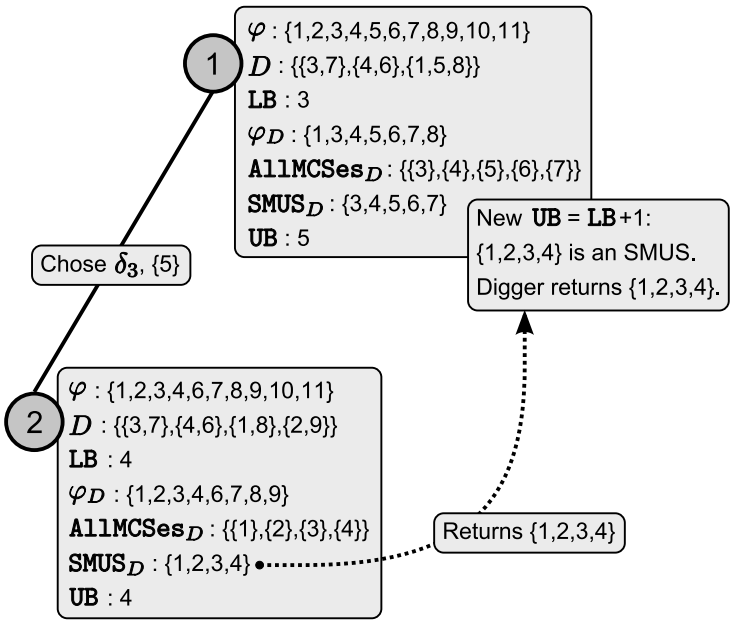

Figure 6: An illustration of Digger's search tree on the example formula φ

of φ. If any SMUS of a particular $\varphi - \delta_i$ is proven to be an SMUS of φ by the lower bound, it is returned immediately. Otherwise, the candidate SMUS and upper bound are updated (Digger would have returned $\emptyset$ if it was unable to improve on the upper bound it was given). If, after exhaustively searching all branch subformulas, no candidate has reached the lower bound and ended the search at this node early, then the smallest MUS found (stored in `candidate`) will contain an SMUS for φ, and so it is returned.

5.5 Optimizations

The algorithm has been presented in a basic form to aid explanation and understanding. The following optimizations are not necessary for its functioning, but they do provide greater performance, mainly by avoiding redundant work. As presented, every recursive call to Digger is passed nothing more than a subformula $\varphi - \delta_i$ and the size of the best SMUS candidate found so far. A great deal of information from the parent can be reused in the child node, however.

Reusing D: Because $\text{MUSes}(\varphi - \delta_i) \subseteq \text{MUSes}(\varphi)$ (by Lemma 1), we know that the lower bound of the parent applies to the child as well. Furthermore, the source of this lower bound, D, is still relevant. Every MCS in D from the parent, after removing any clauses in δ_i, will be a correction set (not necessarily minimal) of $\varphi - \delta_i$, and because we simply removed clauses, these sets will still be disjoint. Therefore, we can use D and `LB` derived from it as seeds for finding

a disjoint CS cover for $\varphi - \delta_i$. This can greatly reduce the time spent finding MCSes in lower levels of the recursion tree.

With our running example as shown in Figure 6, this optimization would allow us to take $D = \{\{C_3, C_7\}, \{C_4, C_6\}, \{C_1, C_5, C_8\}\}$ and remove $\delta_3 = \{C_5\}$ to get $D' = \{\{C_3, C_7\}, \{C_4, C_6\}, \{C_1, C_8\}\}$. Each of the three sets in D' is a correction set of $\varphi - \delta_3$, so the search for MCSes in node 2 can be seeded with D' (by adding blocking clauses for the clauses in D' before any search) and an initial lower bound of 3. Then the search is only needed to find one further MCS, $\{C_2, C_9\}$, which is added to D and increments the lower bound to 4.

This optimization is not a guaranteed performance increase, because the CSes in D' may include a large number of unneeded clauses due to their non-minimality, thus increasing the size of φ_D in a lower level, but in our experiments it almost always boosts performance.

Further exploiting Theorem 4: The upper and lower bounds can be used to prune branches one level earlier than described in some cases. If $\texttt{UB} = \texttt{LB} + 1$ going into the **foreach** loop on line 15, then the subroutine can return early. From Theorem 4, we know that if no SMUS of size $|D|$ was found in φ_D, any SMUS of φ must have at least $|D| + 1$ (i.e. LB+1) clauses. Thus, if we reach line 15 and $\texttt{UB} = \texttt{LB} + 1$, any SMUS of φ must be at least as large as UB, the best found earlier, so no further search is required. This will happen in branches where a good upper bound MUS has been found earlier but the lower bounds are not quite high enough to prune further search in other ways.

Improved use of bounds: The upper and lower bounds can also be used along with the optimization of reusing D to cut short calls to subroutines. **DisjointMCSes** can use the upper bound and the lower bound received from the parent to exit early with a failure if it finds a number of new disjoint MCSes (disjoint from the MCSes received from the parent as well) equal to the difference of the two bounds.

For example, if the algorithm trace shown in Figure 6 were to branch using $\delta_1 = \{C_3\}$ first (instead of $\delta_3 = \{C_5\}$ used in the figure), the optimization of reusing D would pass $D' = \{\{C_7\}, \{C_4, C_6\}, \{C_1, C_5, C_8\}\}$ and a seed lower bound of 3 to node 2 of the recursive tree. The call to **DisjointMCSes** would normally find 3 more MCSes with this seed: $\{C_9\}$, $\{C_{10}\}$, and $\{C_{11}\}$; however, because $\texttt{UB} = 5$, it can terminate after finding just two, because this already proves that LB will be *at least* equal to UB, even though we do not know the exact lower bound. It has accumulated a total number of disjoint CSes equal to the size of the smallest MUS found thus far, proving that any MUS of the current subformula must contain at least as many clauses.

Likewise, both UB and LB can be passed to CAMUS-min to provide the branch-and-bound construction in its second phase with both an initial upper bound (UB-1) to help early pruning and a lower bound for exiting early if it is reached.

5.6 Further Recursion

One of the products of the call to CAMUS-min on line 8 is an SMUS of φ_D. As mentioned, however, CAMUS-min can suffer from intractability in some cases. Digger returns an SMUS, so why not use it there instead?

The fact is that Digger itself, without that call to CAMUS-min, has no ability to produce an SMUS. It provides a method for splitting up the problem into smaller tractable subformulas and performing a branch-and-bound search, but it relies on CAMUS-min to compute MUSes.

Simply replacing that call with another call to Digger would thus result in infinite recursion, never progressing past that line. However, Digger can still be used there to a limited extent. For example, one could call Digger in that spot until a certain recursion level has been reached, at which point CAMUS-min would be used instead to produce an SMUS and proceed. Alternatively, the algorithm could switch to CAMUS-min once the repeated calls to **DisjointMCSes** reach a fixed point, returning the same set of MCSes on two successive calls. We refer to this as the "fixed point heuristic."

This modification of Digger causes increased overhead by replacing a single call to CAMUS-min with an embedded recursive search; however, it can avoid passing CAMUS-min an intractable instance by pruning down the problem even further prior to passing it on. We report on a few instances in which this approach is beneficial in the following section.

6 Experimental Evaluation

We experimentally evaluated both the CAMUS-min and the GGA approaches alongside the Digger algorithm to compare their performance and to gain insight into the performance of Digger on real-world instances. For the latter point, we were interested in determining how effective Digger's heuristics are at pruning its search space by examining the shape of the recursive search trees. We do not perform a comparison to the earlier SMUS algorithm presented in [24]; its reported results for a set of very small instances (60 or 75 variables, 120 clauses) have runtimes of hundreds or thousands of seconds, and its search of the space of USes is much less efficient, algorithmically, than searching the space of MUSes with Digger's pruning heuristics.

We implemented both CAMUS-min and Digger using code that was as similar as possible; both were written in C++ on top of MiniSAT version 1.12b [11]. Digger uses CAMUS-min explicitly in one of its steps, and we wrote the **DisjointMCSes** function with the same approach used to find MCSes in the CAMUS family. All executables were compiled using GCC 4.1.2. We used the same executable for the first phases of CAMUS-min and GGA, while an executable for GGA's second phase was provided by the author, Jianmin Zhang (though we were only able to obtain an early, not-fully-optimized implementation). All experiments were run under Linux (Fedora 7) on a 3.0GHz Intel Core 2 Duo E6850 with 4GB of RAM.

6.1 Overall Performance

We evaluated the performance of the algorithms on two sets of real-world unsatisfiable CNF instances. The first is a set of automotive product configuration benchmarks [29, 30] that have previously been shown to have a wide range of characteristics with respect to each instance's set of MUSes. While all of the instances have around 4000–8000 clauses[1], they range from having just a single MUS to intractably large sets, such as C202_FW_SZ_118, for which analysis of the MCSes shows that it has exactly 2^{127} (about 10^{38}) MUSes. Likewise, the sizes of the MUSes range from 8 clauses up to at least 670, representing between about 0.1% and 13% of an instance's clauses.

Table 1 shows the results for all 84 instances. The first three columns list the instance name and its size. Every instance was run on each of the three algorithms with a timeout of 600 seconds (for CAMUS-min and GGA, each phase was given a separate 600 second timeout). The runtimes are indicated in the next three columns, with a "-" indicating a timeout was reached without finding an SMUS. The fastest of the three results is bolded. The size of the SMUS found by Digger is reported in the next column. For instances in which Digger did not finish, we report the lower bound on the size of any SMUS as well as an upper bound (if any was found before the timeout) as "(lb:ub)". Finally, we report the size of the US found by GGA in the last column.

Of the 84 instances, CAMUS-min was able to finish 44 within the timeout, while Digger completed all of those and others for a total of 75. In certain cases, CAMUS-min was slightly faster than Digger; in the vast majority of the instances, however, Digger was much faster, sometimes by several orders of magnitude. GGA is generally competitive with CAMUS-min, solving just 5 more instances within its timeouts, and neither is consistently faster than the other. GGA is saddled with the same requirement to find all MCSes as CAMUS-min, facing the same intractability problems in its first phase. Its incomplete local search frequently results in finding an SMUS, and it was almost always within one or two clauses of the minimum. However, on those instances where GGA outperforms the other algorithms, it never finds an SMUS (for C220_FV_SZ_55, this is proven in a later experiment reported in Table 4 below).

Note that these runtimes for Digger are also several orders of magnitude faster than those reported in [25] for an earlier implementation of the same algorithm. This substantial difference can mainly be attributed to two implementation differences: (1) the implementations use different solvers for the **DisjointMCSes** subroutine (MiniSAT here versus PBS in [25]) and (2) here we use CAMUS-min with branch-and-bound in the second phase, as opposed to using the basic CAMUS algorithm to generate all MUSes and selecting the smallest, as the earlier implementation did.

Table 2 contains results for running the algorithms on eight benchmarks from the DIMACS set arising from circuit diagnosis. They are similar in size to the product configuration instances except for ssa6288-047, which has approxi-

[1] This is after removing the duplicate clauses that artificially inflate the size of each instance and its set of MUSes.

			Runtimes (sec)				
Name	#V	#C	CAMUS-min	GGA	Digger	|SMUS|	|GGA|
C168_FW_SZ_107	1583	5939	–	–	**0.27**	47	
C168_FW_SZ_128	1583	4777	–	–	**0.184**	92	
C168_FW_SZ_41	1583	4727	–	–	**0.185**	26	
C168_FW_SZ_66	1583	4751	–	–	**0.083**	16	
C168_FW_SZ_75	1583	4744	–	–	**0.128**	48	
C168_FW_UT_2463	1804	6756	–	–	**1.06**	35	
C168_FW_UT_2468	1804	6754	–	–	**0.729**	33	
C168_FW_UT_2469	1804	6767	–	–	**2.5**	32	
C168_FW_UT_714	1804	6754	–	–	**0.059**	9	
C168_FW_UT_851	1804	6758	0.127	0.127	**0.087**	8	8
C168_FW_UT_852	1804	6756	0.126	0.127	**0.087**	8	8
C168_FW_UT_854	1804	6753	0.122	0.122	**0.086**	8	8
C168_FW_UT_855	1804	6752	0.129	0.13	**0.086**	8	8
C170_FR_RZ_32	1528	4067	**0.133**	0.493	0.191	227	228
C170_FR_SZ_58	1528	4083	0.215	0.184	**0.073**	46	47
C170_FR_SZ_92	1528	4195	**0.065**	0.133	0.109	131	131
C170_FR_SZ_95	1528	4068	**0.113**	0.122	0.809	52	54
C170_FR_SZ_96	1528	4068	2.452	3.142	**0.387**	53	55
C202_FS_RZ_44	1556	5399	100.45	3.909	**0.093**	18	21
C202_FS_SZ_104	1556	5405	16.02	6.847	**0.2**	24	24
C202_FS_SZ_121	1556	5387	**0.037**	0.038	0.053	22	22
C202_FS_SZ_122	1556	5385	**0.04**	0.041	0.055	33	33
C202_FS_SZ_74	1556	5561	–	–	**0.179**	150	
C202_FS_SZ_84	1556	5479	–	–	–	(212:)	
C202_FS_SZ_95	1556	5388	–	782.376	**0.091**	7	8
C202_FS_SZ_97	1556	5452	35.9	20.018	**0.128**	28	30
C202_FW_RZ_57	1561	7434	**0.181**	0.459	0.275	213	213
C202_FW_SZ_100	1561	7484	–	–	**0.182**	23	
C202_FW_SZ_103	1561	9024	–	–	**8.64**	148	
C202_FW_SZ_118	1561	7562	0.855	**0.364**	0.955	129	130
C202_FW_SZ_123	1561	7437	**0.063**	0.066	0.094	36	36
C202_FW_SZ_124	1561	7435	**0.046**	0.048	0.077	33	33
C202_FW_SZ_61	1561	7490	–	–	**0.323**	18	
C202_FW_SZ_77	1561	7611	–	–	**0.262**	156	
C202_FW_SZ_87	1561	7696	–	–	–	(356:)	
C202_FW_SZ_96	1561	7599	–	–	–	(207:)	
C202_FW_SZ_98	1561	7438	–	–	**0.119**	7	
C202_FW_UT_2814	1820	9957	–	–	–	(13:16)	
C202_FW_UT_2815	1820	9957	–	–	–	(13:16)	

Table 1: Experimental results for automotive product configuration benchmarks

			Runtimes (sec)				
Name	#V	#C	CAMUS-min	GGA	Digger	|SMUS|	|GGA|
C208_FA_RZ_43	1516	4254	4.699	5.389	**0.067**	8	10
C208_FA_RZ_64	1516	4246	**0.116**	0.384	0.16	212	212
C208_FA_SZ_120	1516	4247	**0.043**	0.045	0.048	34	34
C208_FA_SZ_121	1516	4247	**0.039**	0.041	0.047	32	32
C208_FA_SZ_87	1516	4255	0.227	**0.208**	0.412	18	19
C208_FA_UT_3254	1805	6153	0.173	0.179	**0.129**	40	42
C208_FA_UT_3255	1805	6156	0.197	0.205	**0.133**	40	41
C208_FC_RZ_65	1513	4491	–	–	**0.073**	12	
C208_FC_RZ_70	1513	4468	**0.138**	0.414	0.17	212	212
C208_FC_SZ_107	1513	4554	–	–	**0.098**	44	
C208_FC_SZ_127	1513	4469	**0.026**	0.028	0.043	34	34
C208_FC_SZ_128	1513	4469	**0.028**	0.031	0.043	32	32
C210_FS_RZ_23	1608	4911	–	–	**0.37**	31	
C210_FS_RZ_38	1607	4900	537.3	133.854	**0.331**	25	27
C210_FS_RZ_40	1607	4891	**0.131**	0.256	0.144	140	140
C210_FS_SZ_103	1607	4915	–	–	**0.108**	45	
C210_FS_SZ_107	1607	4902	–	186.341	**0.067**	15	15
C210_FS_SZ_123	1607	5062	1.527	0.612	**0.251**	176	176
C210_FS_SZ_129	1607	4894	**0.032**	0.035	0.052	33	33
C210_FS_SZ_130	1607	4894	**0.03**	0.033	0.051	31	31
C210_FS_SZ_55	1608	4917	–	–	**0.169**	41	
C210_FS_SZ_78	1607	5071	–	–	**0.197**	170	
C210_FW_RZ_30	1629	6407	–	–	**43.7**	35	
C210_FW_RZ_57	1628	6390	–	348.361	**0.12**	25	27
C210_FW_RZ_59	1628	6381	**0.158**	0.284	0.191	140	140
C210_FW_SZ_106	1628	6405	–	–	**0.248**	49	
C210_FW_SZ_111	1628	6393	–	450.489	**0.093**	15	15
C210_FW_SZ_128	1628	6401	–	–	**0.161**	22	
C210_FW_SZ_129	1628	6595	1.888	0.964	**0.341**	176	176
C210_FW_SZ_135	1628	6384	**0.047**	0.048	0.072	33	33
C210_FW_SZ_136	1628	6384	**0.042**	0.046	0.07	31	31
C210_FW_SZ_80	1628	6560	–	–	**0.274**	171	
C210_FW_SZ_90	1628	6977	–	–	–	(272:)	
C210_FW_SZ_91	1628	6709	–	–	–	(268:)	
C210_FW_UT_8630	1891	8511	–	–	**0.561**	30	
C210_FW_UT_8634	1891	8504	–	–	**0.617**	23	
C220_FV_RZ_12	1530	4017	0.135	0.13	**0.057**	11	11
C220_FV_RZ_13	1530	4014	0.097	0.095	**0.053**	10	10
C220_FV_RZ_14	1530	4013	**0.035**	0.037	0.041	11	11
C220_FV_SZ_114	1530	4305	9.77	20.049	**0.161**	132	132
C220_FV_SZ_121	1530	4035	0.077	0.088	**0.068**	58	60
C220_FV_SZ_39	1530	4788	–	–	–	(202:205)	
C220_FV_SZ_46	1530	4014	3.162	2.789	**0.066**	17	17
C220_FV_SZ_55	1530	5281	–	**18.62**	–	(297:)	302
C220_FV_SZ_65	1530	4014	0.289	0.288	**0.051**	23	23

Table 1: Experimental results (cont.)

			Runtimes (sec)				
Name	#V	#C	CAMUS-min	GGA	Digger	|SMUS|	|GGA|
bf0432-007	1318	3668	708	650.559	**6.4**	1151	1161
bf1355-075	3761	6778	–	–	**0.258**	150	
bf1355-638	3760	6768	–	–	**0.269**	152	
bf2670-001	4853	3434	–	–	**1.72**	132	
ssa0432-003	1266	1027	2.69	6.494	**0.124**	309	317
ssa2670-130	4849	3321	–	–	**1.36**	656	
ssa2670-141	4843	2315	**1.204**	65.371	1.77	1246	1246
ssa6288-047	17303	34238	–	–	–		

Table 2: Experimental results for circuit diagnosis benchmarks

mately ten times as many variables and six times as many clauses compared to the average sizes of the automotive benchmarks. CAMUS-min and GGA are able to solve three of them given the 600 second timeouts, while Digger solves all but one (the largest). Again, Digger is much faster than both CAMUS-min and GGA except in ssa2670-141, where CAMUS-min is slightly faster. The sizes of the SMUSes in these instances are larger than those of the product configuration benchmarks, ranging from 2% to 50% of each instance's clauses.

Overall, we see that the three algorithms have adequate performance on many instances, but Digger greatly outperforms the alternatives both in runtime and in the number of solved instances. Its manner of splitting up the problem and using a branch-and-bound approach, implemented on top of the same techniques as CAMUS-min, are providing this increased performance; in the following, we look more closely at the search tree itself.

6.2 Search Tree Statistics

The shape of the search tree gives us a better idea of how the Digger algorithm operates and can point to potential improvements. We recorded search tree data for all of the trials we ran, and from them we drew the following observations:

On all 7 of the circuit benchmarks and 70 of the 74 product configuration instances that finished within the 600 second timeout, Digger required no branching or search to find an SMUS. For all of these, the SMUS was contained in φ_D and the lower bound obtained from D proved it was minimum (Digger returned on line 10 of the pseudocode in Figure 4). We see that the disjoint MCS heuristic often provides a means to quickly generate a subformula that contains an SMUS as well as a robust lower bound to prove its minimality.

Eight of the product configuration instances had more than one search tree node; 5 finished, and 3 timed out. Table 3 contains search tree statistics for these instances. We report whether the instance finished within the timeout, the number of nodes explored, the percentages of these that were SAT (returning in line 2 in Figure 4) or pruned by the bounds (returning in line 6 in Figure 4 or due to the pruning optimization described in Section 5.5), and the depth and average branching factor of the search tree.

Name	Fin.	Nodes	%SAT	%Prune	Depth	AvgBranch
C168_FW_UT_2469	Y	39	46.2	51.3	2	38.0
C202_FW_UT_2814	N	1687	2.6	96.9	4	248.9
C202_FW_UT_2815	N	1686	2.6	96.9	4	248.9
C208_FA_SZ_87	Y	23	56.5	39.1	2	22.0
C210_FS_RZ_23	Y	22	54.5	36.4	3	40.0
C210_FS_RZ_38	Y	19	57.9	31.6	3	34.0
C210_FW_RZ_30	Y	942	41.7	56.2	4	47.1
C220_FV_SZ_39	N	6566	76.9	22.7	4	240.1

Table 3: Search tree statistics for instances with more than one node

From these statistics, we can see that the trees are uniformly wide and shallow. Most nodes are not expanded due to being either satisfiable or pruned because the lower bound computed for the subformula exceeded the size of the smallest MUS found thus far. Those that are expanded to search further have a fairly high branching factor, however, and in those instances on which Digger reached the timeout, it only reached a maximum tree depth of 4, despite exploring thousands of nodes. The five instances that did finish all had much lower branching factors and thus smaller search trees.

Digger timed out in the first call to CAMUS-min in six of the product configuration instances. These are the instances for which no upper bound on |SMUS| is reported in Table 1, because no MUS was produced. While the disjoint MCS heuristic did reduce the size of the instance greatly, φ_D was still too large for CAMUS-min to solve within the timeout. It is in these cases that the variant of Digger described in Section 5.6 becomes quite useful.

Table 4 contains results for running that variant of Digger (using the fixed point heuristic to control the recursion) on these six instances. The "LB" column lists the size of the lower bound, "UB" gives the size of the smallest MUS found, and the final column contains the time in seconds at which the MUS that set that upper bound was found. (In the instances not reported in Table 4, the variant generally performed similarly to the results in Table 1, but it had slightly higher runtimes due to the increased overhead.)

			Time (sec)
Name	LB	UB	to UB
C202_FS_SZ_84	212	214	2.95
C202_FW_SZ_87	356	361	12.49
C202_FW_SZ_96	207	209	6.08
C210_FW_SZ_90	272	276	3.35
C210_FW_SZ_91	268	273	2.83
C220_FV_SZ_55	297	297	1.55

Table 4: Results for the variant of Digger described in Section 5.6

On C220_FV_SZ_55, this variant of Digger was able to find an SMUS in 1.55 seconds, which compares well with the standard version that timed out after 600

seconds. The variant quickly found small MUSes, close to the lower bound, for all of the other five instances, but it was unable to prove that they were SMUSes (by exhausting the search space of remaining MUSes) within the timeout. In those cases, an improved lower bound could prune a great deal of that search, and it could in fact prove that those candidates found so early on (the source of the UB size in Table 4) are SMUSes to avoid that search entirely. Overall, we can see that this variant provides a means to quickly obtain a small MUS close to the lower bound in cases where the standard version times out in a call to CAMUS-min.

7 Conclusions and Future Work

Understanding the causes of infeasibility of Boolean formulas is a problem of interest in various theoretical and practical areas of computer science. Minimal unsatisfiable subformulas provide useful explanations of infeasibility. We have presented two algorithms for finding an SMUS of a Boolean formula, i.e., an MUS with the least number of clauses. Both algorithms utilize the relationship between MCSes and MUSes in their operation. The first, CAMUS-min, follows an approach to generate all MUSes from the complete set of MCSes but searches for the smallest of these, and it presents baseline results for a "simple" solution, in the sense that it follows directly from a system for enumerating all MUSes. The second algorithm, Digger, uses the relationship to partition a given problem into more tractable subformulas as well as to compute a strong lower bound on the size of an SMUS as it searches. This bound is the basis for a branch-and-bound procedure that finds an SMUS by recursively branching on subformulas.

We have presented novel experimental results on two benchmark suites comparing these two algorithms with each other and with a third from the literature that is similar to our first algorithm but which uses a genetic algorithm to perform an incomplete local search to minimize the size of the MUS found. The results show that Digger greatly outperforms both of the other approaches, primarily due to its ability to avoid the intractability of computing all MCSes of a formula, which is a requirement of both of the other techniques. Further, experimental data shows how our lower bound computation is robust, helping to find an SMUS and often sufficing to prove its minimality without branching on subformulas at all.

One direction for future work is to improve the lower bound used in Digger. In those instances where an SMUS is not found in the first search node, thus requiring branching, the size of an SMUS is often very close to the lower bound. If the lower bound could be improved by just a few clauses in these cases, the algorithm would avoid a great deal of work spent searching to prove that the candidate found early on is in fact the smallest.

Finally, we should note that all of these algorithms can be easily extended to find SMUSes of constraint types beyond Boolean Satisfiability. All three are built on the basic operation of finding MCSes of a constraint system and performing computations on those MCSes. Finding an MCS is a simple opti-

mization problem, very similar to MaxSAT, which can be easily implemented using existing solvers for any constraint type. The operations performed on the MCSes themselves need no knowledge of the semantics of the MCSes, and they are treated as nothing more complex than sets of elements from a finite domain. Therefore, all three algorithms can be adapted to other constraint types by simply replacing their implementation of the MCS-related subroutine. Another area of future work will be to do just this, applying these techniques to other constraint types and applications thereof.

Acknowledgments

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 no t necessarily reflect the view of the National Science Foundation (NSF). This work was funded in part by the DARPA/MARCO Gigascale Systems Research Center. This work is partially supported by Fundação para a Ciência e Tecnologia under research project POSC/EIA/61852/2004 and by EU project IST/214898.

References

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

[2] Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah. Refinement strategies for verification methods based on datapath abstraction. In *Proceedings of the 2006 conference on Asia South Pacific design automation (ASP-DAC'06)*, pages 19–24, 2006.

[3] Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah. CEGAR-based formal hardware verification: a case study. Technical Report CSE-TR-531-07, University of Michigan, 2007.

[4] J. Bailey and P. J. Stuckey. 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)*, volume 3350 of *LNCS*, pages 174–186, 2005.

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

[6] R. Bruni and A. Sassano. Restoring satisfiability or maintaining unsatisfiability by finding small unsatisfiable subformulae. In *LICS 2001 Workshop*

on Theory and Applications of Satisfiability Testing (SAT-2001), volume 9 of *Electronic Notes in Discrete Mathematics*, pages 162–173, 2001.

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

[8] H. K. Büning and X. Zhao. Minimal falsity for QBF with deficiency one. Workshop on Theory and Applications of Quantified Boolean Formulas, 2001.

[9] S. Dasgupta and V. Chandru. Minimal unsatisfiable sets: Classification and bounds. In M. J. Maher, editor, *Advances in Computer Science - ASIAN 2004*, volume 3321 of *LNCS*, pages 330–342. Springer, 2004.

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

[11] N. Eén and N. Sörensson. An extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT-2003)*, volume 2919 of *LNCS*, pages 502–518, 2003.

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

[13] R. Gershman, M. Koifman, and O. Strichman. Deriving small unsatisfiable cores with dominators. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*, pages 109–122, 2006.

[14] E. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Proceedings of the Conference on Design, Automation, and Test in Europe (DATE'03)*, pages 10886–10891, 2003.

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

[16] G. D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, 1996.

[17] B. Han and S.-J. Lee. Deriving minimal conflict sets by CS-trees with mark set in diagnosis from first principles. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 29(2):281–286, April 1999.

[18] J. Huang. MUP: A minimal unsatisfiability prover. In *Proceedings of the 10th Asia and South Pacific Design Automation Conference (ASP-DAC'05)*, pages 432–437, 2005.

[19] H. Jain, D. Kroening, N. Sharygina, and E. Clarke. Word level predicate abstraction and refinement for verifying RTL verilog. In *Proceedings of the 42nd annual conference on Design automation (DAC'05)*, pages 445–450, 2005.

[20] O. Kullmann. An application of matroid theory to the SAT problem. In *15th Annual IEEE Conference on Computational Complexity*, pages 116–124, July 2000.

[21] R. P. Kurshan. *Computer Aided Verification of Coordinating Processes*. Princeton University Press, Princeton, NJ, 1994.

[22] M. H. Liffiton and K. A. Sakallah. On finding all minimally unsatisfiable subformulas. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT-2005)*, volume 3569 of *LNCS*, pages 173–186, 2005.

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

[24] I. Lynce and J. Marques-Silva. On computing minimum unsatisfiable cores. In *The 7th International Conference on Theory and Applications of Satisfiability Testing (SAT-2004)*, 2004.

[25] M. N. Mneimneh, I. Lynce, Z. S. Andraus, J. P. M. Silva, and K. A. Sakallah. 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)*, volume 3569 of *LNCS*, pages 467–474, 2005.

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

[27] Y. Oh, M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, and I. L. Markov. AMUSE: a minimally-unsatisfiable subformula extractor. In *Proceedings of the 41st Annual Conference on Design Automation (DAC'04)*, pages 518–523, 2004.

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

[29] C. Sinz. SAT benchmarks from automotive product configuration. Website. http://www-sr.informatik.uni-tuebingen.de/~sinz/DC/.

[30] C. Sinz, A. Kaiser, and W. Küchlin. Formal methods for the validation of automotive product configuration data. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 17(1):75–97, 2003.

[31] S. Szeider. Minimal unsatisfiable formulas with bounded clause-variable difference are fixed-parameter tractable. *Journal of Computer and System Sciences*, 69(4):656–674, December 2004.

[32] J. Zhang, S. Li, and S. Shen. Extracting minimum unsatisfiable cores with a greedy genetic algorithm. In *AI 2006: Advances in Artificial Intelligence*, volume 4304 of *LNCS*, pages 847–856, 2006.

[33] L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable Boolean formula. In *The 6th International Conference on Theory and Applications of Satisfiability Testing (SAT-2003)*, 2003.

[34] L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Proceedings of the Conference on Design, Automation, and Test in Europe (DATE'03)*, pages 10880–10885, 2003.

A Example CNF Formulas

A.1 Exponential Number of MUSes

The following formula, parameterized for n, has 2^n MUSes, each of size $2n+1$. (Clauses have been written as implications for clarity; each implication $(a \rightarrow b)$ is simply $(\neg a \vee b)$ in CNF.)

$$\begin{aligned}\varphi_{expMUSes} = \; & (c_0) \\ & \wedge (c_0 \rightarrow a_1) \wedge (c_0 \rightarrow b_1) \wedge (a_1 \rightarrow c_1) \wedge (b_1 \rightarrow c_1) \\ & \wedge (c_1 \rightarrow a_2) \wedge (c_1 \rightarrow b_2) \wedge (a_2 \rightarrow c_2) \wedge (b_2 \rightarrow c_2) \\ & \vdots \\ & \wedge (c_{n-1} \rightarrow a_n) \wedge (c_{n-1} \rightarrow b_n) \wedge (a_n \rightarrow \neg c_0) \wedge (b_n \rightarrow \neg c_0)\end{aligned}$$

This formula has $3n$ variables and $4n+1$ clauses. Every MUS contains (c_0) and, for each of the n groups of 4 related implications, either the 2-clause implication chain through a_i or that through b_i. These n binary choices lead to the instance containing 2^n MUSes.

A.2 Exponential Number of MCSes

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, any 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. One simple example is:

$$\begin{aligned}\varphi_{expMCSes} = \; & (x_{1,1}) \wedge (x_{1,1} \rightarrow x_{1,2}) \wedge (x_{1,2} \rightarrow x_{1,3}) \wedge \cdots \wedge (x_{1,k-1} \rightarrow \neg x_{1,1}) \\ & \wedge (x_{2,1}) \wedge (x_{2,1} \rightarrow x_{2,2}) \wedge (x_{2,2} \rightarrow x_{2,3}) \wedge \cdots \wedge (x_{2,k-1} \rightarrow \neg x_{2,1}) \\ & \vdots \\ & \wedge (x_{n,1}) \wedge (x_{n,1} \rightarrow x_{n,2}) \wedge (x_{n,2} \rightarrow x_{n,3}) \wedge \cdots \wedge (x_{n,k-1} \rightarrow \neg x_{n,1})\end{aligned}$$

Each line is independent, sharing no variables with the others, and each is an MUS. There are $n \cdot k$ clauses, n MUSes, and k^n MCSes.