Analyzing Infeasible Constraint Systems

by

Mark H. Liffiton

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2009

Graduate school, except I didn't have a pet lion.

Albrecht Dürer's *meisterstiche*:
St. Jerome in his Study; *Melancholia I*; and *Knight, Death, and the Devil*

to Mom and Dad, because they’re pretty great parents,

and they even listen when I try to explain this stuff to them.

Acknowledgments

I would like to acknowledge my debts to the following people:

- Rob Felty, for his LaTeX class file for Rackham-compliant dissertations. I owe him one week of my life, in which I did not have to wrestle with formatting, margins, numbering, or anything of the sort.
- Aaron Johnson, for introducing me to the joy of the frontispiece. I owe him one “borrowed” idea.
- Karem Sakallah, my advisor, for letting me find my own way through graduate school with appropriate nudges and advice when needed. I owe him a good portion of my confidence in my ability to perform independent research and at least a meal or two.
- The faculty members here at Michigan, at WPI, and at Simon’s Rock from whom I’ve taken classes or with whom I’ve interacted in some way. They all taught me a lot, whether I wanted them to at the time or not, and thanks to them I now get to become one of them. I’ll figure out what, exactly, I owe them for this after I’ve tried it out for a while.
- My friends, who all either helped me edit large pieces of this dissertation at some point, fed me (hey, it counts for a **lot**), or generally provided invaluable friendship for the past six years. After all of this, I owe them many visits, phone calls, and emails, all of which will further increase my debts as I crash on their couches, eat their food, and request their advice and counsel.
- My parents, for... oh, this one is obvious. I owe them more than can ever be repaid. I just hope they had some idea of what they were getting themselves into.

Table of Contents

List of Tables

List of Figures

List of Appendices

Abstract

Constraint systems, problems defined by sets of variables and constraints affecting the allowed assignments to those variables, arise in a wide range of real-world problems, from planning and scheduling to digital logic circuit verification. These problems have been studied widely in computer science and operations research, and a great deal of research has gone into developing algorithms that solve them quickly, producing assignments to variables that satisfy all of a system's constraints. Many constraint systems turn out to have no satisfying assignments – they are said to be overconstrained or infeasible – and work on analyzing these instances is much less mature. It is in this area that this work lies.

Most work on analyzing infeasibility has focused on producing singular, approximate, partial views of the infeasibility of a given problem; a distinguishing feature of this work is that it is focused on complete analyses. We have developed algorithms that compute the full structure of a problem's infeasibility in the form of minimal correction sets (MCSes) and minimal unsatisfiable subsets (MUSes), both of which are subsets of a system's constraints that irreducibly describe part of its infeasibility. The value of these complete analyses has been demonstrated by applications of this work to two different logic circuit verification tasks, in which the complete views these algorithms produce have been shown to be critical for performance. Spurred partially by the needs of the industrial applications and also by theoretical considerations regarding the innate intractability of the given problems, extensions of these algorithms have been developed in several directions, enhancing performance by integrating partial, approximate analyses as guides, solving related problems that avoid the intractability while still providing more information than the singular views offered by other work, and generally exploring the space of infeasibility analysis in novel ways.

Overall, this work helps to expand the field of constraint system research with new tools for analyzing infeasibility. This research, which has been kept as broadly-applicable as possible, serves as a base, opening up complete infeasibility analysis to researchers across the field of constraint research.

Chapter 1
Introduction and Background

A wide variety of real-world combinatorial problems can be modeled as constraint satisfaction problems (CSPs), ranging from scheduling to digital circuit verification. They have been used in industrial and commercial applications for several decades, with a great deal of research put into developing efficient algorithms for solving them. These constraint solvers almost always operate in a manner common to solving any problem in NP: by providing a short "proof" in the form of a satisfying solution when a problem is satisfiable and a one-word answer, "Unsatisfiable," if it is not. Recently, however, more people have focused on expanding that one-word answer; researchers have been investigating techniques for extracting useful information from unsatisfiable constraint systems.

The information in an unsatisfiable instance can take numerous forms depending on the application. It can provide a compact diagnosis of an error in some cases or separate irrelevant details from important constraints in an intentionally unsatisfiable instance in others. For example, in a scheduling problem, the infeasibility may be due to a human error in entering the constraints. Simply hearing "Unsatisfiable" from the solver will do little to help a user correct the error. Alternatively, in some circuit verification tasks, a circuit is modeled with constraints in such a way that the instance is known to be unsatisfiable, but the important information is which constraints make it so, with the rest being irrelevant to the task at hand.

Traditional constraint solvers provide no information in these cases, because fundamentally, all unsatisfiable constraint systems are equivalent in the sense that none have any solutions. While there is nothing to be learned from this empty solution space, the structure of the constraints themselves and their interrelations can provide valuable information about the problem being modeled. In the examples above, they provide insight into why an instance is unsatisfiable, and information about which parts *are* satisfiable and how to correct the infeasibility can be produced as well. Overall, the instances do implicitly possess information beyond "Unsatisfiable."

This work lies in this field of analyzing infeasible constraint systems. We have developed several tools that allow us to look into an unsatisfiable system and extract explanations, isolate conflicts, prune redundancies, and otherwise probe the structure behind the empty solution space. Starting from a theoretical foundation that links two different structural views of infeasibility, we developed an algorithmic base, called CAMUS (pronounced "ka-**moo**" after the French writer), for producing complete views of both types, as described in Chapter 2. From this base, several extensions and modifications have been made to enhance performance and solve related problems, which comprise Chapter 3. Finally, Chapter 4 outlines how these algorithms have been applied in two different industrially-relevant applications, demonstrating the value of the work and motivating further investigations in some of the directions mentioned in the conclusion.

The remainder of this chapter lays out the problem space of this work in more detail, describing constraint systems and some common types of constraints; presents related work in the general area of infeasibility analysis; and describes the contributions of this dissertation in the final section.

1.1 Problem Space: Constraint Systems

There are many ways to formally define a constraint system, even without getting into the multitude of constraint formalisms adopted by researchers in areas ranging from logic circuit analysis, to package routing, to assigning radio link frequencies within a limited spectrum while minimizing interference (19). One fairly generic description of a constraint system is as follows:

A constraint system C consists of a set of variables and a set of constraints. Each variable has a particular domain; in the case of Boolean Satisfiability (SAT), the domain of each variable is $\{$TRUE, FALSE$\}$, while in other cases the domain may be the set of integers, reals, or a finite set of arbitrary symbols. We can consider any assignment to the variables as an element in the cross product of the variables' domains, i.e., an assignment of a value to every variable. Then, each constraint partitions the assignments into two sets: those that satisfy the constraint and those that do not. This can be specified in many ways, for example by describing the sets implicitly ("$x \geq 5$") or by explicitly listing tuples that satisfy the constraint ("$\{\{x=2,y=4\},\{x=3,y=3\},\{x=4,y=2\}\}$"). Given, then, a set of variables and a set of constraints, one can state whether the constraint system is "satisfiable," if there exists some variable assignment that satisfies every constraint, or "unsatisfiable," if there is none.

1.1.1 Classes of Constraint Systems

Generally, one can describe a class of constraint systems by the domain of their variables and the form of their constraints. There are more classes than can be listed here, but what follows is a list of some of the most common and most important in terms of application.

Boolean Satisfiability (SAT): Every variable in a SAT instance can take one of two values, TRUE or FALSE. The constraints can be any expression in Boolean logic that will evaluate to TRUE (the constraint is satisfied) or FALSE (it is not). Often, SAT instances are encoded in *Conjunctive Normal Form* (CNF). Formally, a CNF formula φ is defined as follows:

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

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 often treat CNF formulas as sets of clauses (*clause sets*), so equivalently: $\varphi = \bigcup_{i=1...n} C_i$.

A CNF instance is said to be *satisfiable* if there exists some assignment to its variables that makes the formula evaluate to 1 or 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 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 most common solution technique in use today is based on an algorithm developed by Davis, Putnam, Logemann, and Loveland (24; 25), dubbed "DPLL." DPLL utilizes a backtracking search that selects a variable at every level and splits on the two possible values of that variable. When it reaches a point where it has chosen values for a set of variables such that a clause is invalidated, called a conflict, it backtracks and tries a different path. If it ever reaches a node in which all variables have been assigned values without reaching a conflict, it has found a satisfying assignment; otherwise, the search exhausts all branches and exits with a result of "Unsatisfiable." The basic backtracking search has been extended with numerous optimizations, which include learning conflicts to prune the search space, efficient propagation techniques for quickly determining effects of new variable assignments, and variable ordering heuristics that on average reduce the size of the search tree by reaching conflicts faster.

Linear Programs (LP): The domain of the variables in a linear program is the set of reals (often the positive reals, though the restriction that all variables are greater than or equal to 0 can be seen as an additional constraint on every variable), and every constraint is an inequality of the form $a_1x_1 + a_2x_2 + \cdots + a_nx_n \leq b$, where x_1 through x_n are the variables, and the a_i and b terms are constants. The solution space (the set of all satisfying assignments) can be seen as the interior of a convex n-dimensional polytope, for which every irredundant constraint defines a facet. Linear programs are most often used in optimization tasks with an objective function specified as a linear combination of the variables that is to be minimized or maximized. An optimal solution for an LP can be found very quickly using the well-known *Simplex* algorithm (though it has exponential worst-case complexity, it most often in practice finds solutions in polynomial time).

Integer Linear Programs (ILP): An ILP has the same type of constraints as an LP, but the domain of each variable is instead the integers. This additional restriction makes the problem much harder, and the problem of deciding the satisfiability of an ILP is NP-Complete.

Satisfiability Modulo Theories (SMT): SMT is a constraint formalism that subsumes all of the above constraint types. Formally, an SMT instance is a first-order logic (FOL) formula with particular interpretations for some predicates and functions. Essentially, it is a Boolean SAT formula where some variables have been replaced by constraints (formally, predicates) from any number of different "background theories." The expressiveness of SMT is limited only by the background theories; every SMT solver has capabilities to handle some set of theories. Examples of background theories include the theory of real numbers (thus subsuming LP and various "temporal" problem types from AI), the theory of integers (ILP), and the theory of bit vectors (allowing for efficient modeling of microprocessors, for example).

SMT can be solved by enhanced DPLL-style algorithms that make decisions about the truth values of a formula's predicates in the backtracking search described above and then can pass the collection of predicates assigned TRUE at any given point to a solver or solvers for the underlying theories to check their satisfiability together. Communication between the theory solvers and the overarching search allows for learning new constraints and additional optimizations that yield efficient SMT solvers suitable for many industrial applications across a variety of domains.

Constraint Satisfaction Problems (CSP): While the name is rather non-specific, “CSP” has traditionally been used in the artificial intelligence community to refer to problems whose variables have finite domains and whose constraints are enumerations of “allowed” tuples of values for sets of variables. For example, the graph coloring problem is a CSP of this type: every node is a variable whose domain is the set of allowed colors and the edges are constraints that only include tuples that assign the two connected nodes different colors. As both SAT and CSP are NP-complete problems, either can be reduced to the other, and straightforward translations exist; however, the two problems have historically been studied in separate communities focusing on different techniques and using different terminology. Throughout this dissertation, “CSP” will generally be used to refer to this type of finite-domain constraint satisfaction problem.

From these constraint types, the work in this dissertation has focused on implementing algorithms in the SAT and SMT domains. Boolean SAT, especially, is also an area with a great deal of work related to infeasible constraint systems, perhaps because analyses of infeasibility have found direct application in industrial logic circuit design and verification tasks. However, this work is broadly applicable to any constraint type, as one goal has been to produce general algorithms that do not rely on the details of any one constraint type or solver.

1.1.2 Important Concepts

Two important concepts common in much of the research on constraint systems are best described here, as they are referenced throughout this work:

AtMost Constraints are a type of counting or cardinality constraint. Given a set of n literals $\{l_1, l_2, \ldots, l_n\}$ and a positive integer k, s.t. $k < n$, 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.

This constraint can be encoded into Boolean CNF using encodings such as in (79), or it can be implemented efficiently in a SAT solver that employs watched variables. An implementation of an AtMost constraint can watch the assignments to the variables in the constraint and immediately propagate the negation of each remaining literal once k of them have been assigned TRUE. On a closed SAT solver that does not allow for a built-in implementation of the AtMost constraint, the CNF encoding can still be used.

Clause-Selector Variables can be used to augment a CNF formula in such a way that standard SAT solvers can manipulate and, in effect, reason about the formula's clauses without any modification to the solver itself. For two examples of the use of clause-selector variables, see the algorithms in (66) and (72).

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. The original constraint C_i is no longer enforced when y_i is assigned FALSE, though it may still be satisfied. This change gives a SAT solver the ability to enable and disable constraints as part of its normal search, checking the satisfiability of different subsets of constraints within a single backtracking search tree.

1.2 Related Work on Infeasibility

Work related to unsatisfiable constraint systems can be found in numerous independent fields of research. Furthermore, the terminology used to describe problems, techniques, and relevant concepts is often unique to each individual field. The following terms have all been used in different publications just to name the conceptual focus of this dissertation:

- Conflicting
- Defeasible
- Inconsistent
- Infeasible
- Overconstrained
- Overdetermined
- Unsatisfiable

With varied terminology and few citation links between different fields, discovering all related work is difficult. Many research contributions have arisen independently in two or more fields, each seemingly without knowledge of the other. In an attempt to give this work a solid base and avoid this duplication of effort as much as possible, this section is an attempt to collect an exhaustive list of existing related work, though there may be a related area of research not covered. We try to draw conceptual links between the references as much as possible.

Previous research into unsatisfiable constraint systems can be roughly partitioned along the lines of the "answer" that is sought, of course with some work straddling multiple classes. Most research related to analyzing infeasibility focuses on one of the following:

1. Proving a result of "Unsatisfiable" returned by a solver (resolution proof checkers).
2. Finding one small, minimum, or smallest subset or *core* of the instance that causes the infeasibility (one unsatisfiable subset (US) or minimal unsatisfiable subset (MUS)).
3. Finding *multiple* or *all* such subsets (many or all MUSes).
4. Determining how much of the instance can be satisfied if the remainder is ignored (MaxSAT/MaxCSP).
5. Discovering portions of the instance *not* involved in the infeasibility (autarkies).

Another partition of the research can be made into theoretical and applied classes. Theoretical work has focused on complexity analysis problems related to analyzing unsatisfiable instances, often considering subclasses of instances for which tractability results can be proven. The applied work involves the development and testing of algorithms for solving such problems and investigating particular aspects of overconstrained instances; each effort is usually focused on solving a particular problem for a particular type of constraint system.

And finally, most researchers work with a particular *type* of constraint system. A great deal of research on infeasibility has been done on Boolean satisfiability (SAT), often in relation to its application to logic circuit verification, while some has also been done on linear programs (LP) and integer programs (IP) in the field of operations research, along with scattered work in other domains.

The following is organized by the “answer” on which any given work focuses. Within each subsection, we first define relevant terminology (including synonymous terms from other fields that have studied the problem); then, we present both the major theoretical contributions as well as the most relevant applied work, indicating the domain or constraint type used in each.

The work in this dissertation has fallen mainly in the “applied” domain so we will present related work mainly from that group; we will discuss the more prominent theoretical work in the area as well as that which is directly related, but we will not explore theoretical results exhaustively. The constraint type to which we have the most exposure is Boolean Satisfiability, and we have implemented algorithms for that type of constraint; furthermore, this research focuses on extracting all MCSes, MUSes, and autarkies (all defined in the following subsections). This section looks at research related to this work directly, but it also presents major research that informs or is otherwise influential to the research described in later chapters.

1.2.1 Resolution Proof Checkers

Definition 1. *Resolution* in Boolean CNF formulas is a logical equivalence rule by which two clauses $a \vee x_1 \vee \cdots \vee x_m$ and $\neg a \vee y_1 \vee \cdots \vee y_n$ produce a *resolvent* clause $x_1 \vee \cdots \vee x_m \vee y_1 \vee \cdots \vee y_n$, which is implied by the original two clauses. The resolvent of two clauses in a formula can be added to the formula without changing its function.

Definition 2. A *Resolution Proof* of an unsatisfiable Boolean CNF formula F is a directed acyclic graph (DAG) whose source nodes are labeled with a subset of the clauses of F; every non-source node has two parents such that the node's label is the result of performing resolution on the parents' labels; and the sole sink node contains the empty clause, indicating a conflict.

A resolution proof of an unsatisfiable formula contains a complete trace of the resolution steps needed to form resolvent clauses implied by original clauses until a conflict (e.g., between x_i and $\neg x_i$) is found, proving the infeasibility of the formula.

Resolution proof checkers were introduced as a means of verifying "Unsatisfiable" results returned by SAT solvers. Affirmative solutions to any problem in NP have short proofs of the result; in the case of SAT, these are in the form of satisfying assignments that can be verified in polynomial time against the formula. No such short proof is known for unsatisfiable formulas, but a resolution proof is a generally tractable technique for verifying such solutions. Certain solvers operate by resolution internally to learn additional clauses as they progress and detect a conflict in the case of an infeasible formula. Their results can be verified by providing the resolution graphs they form to an external program. Van Gelder first proposed the idea of extracting resolution proofs from DPLL SAT solvers in order to verify their results (84). Later work, mentioned in the following subsection, extended this concept to apply it to the problem of extracting unsatisfiable subsets of Boolean formulas, which is the main connection to this dissertation.

1.2.2 Extracting Unsatisfiable Subsets

Definition 3. *Minimal Unsatisfiable Subset* (MUS): Given an unsatisfiable constraints system C, a set of clauses $U \subseteq C$ is an MUS if U is unsatisfiable and $\forall C_i \in U$, $U - \{C_i\}$ is satisfiable.

(Within the operations research domain, an MUS of a linear program is called an Irreducible Inconsistent/Infeasible Subset/Subsystem (IIS) (85).)
Additionally, many algorithms have been developed that extract an *Unsatisfiable Subset* (US), which is simply any unsatisfiable subset of an unsatisfiable constraint system. USes are often also referred to as unsatisfiable *cores* (UCes). Furthermore, some algorithms find a *smallest* MUS (SMUS) or ***minimum*** *unsatisfiable subset/core* of a formula.

This subsection covers work on finding generally *one* MUS or core of an unsatisfiable constraint system. This dissertation deals with the problem of extracting *all* MUSes, as discussed further in the next subsection, but the work arose from research on finding a single MUS. Therefore, the research presented here is quite relevant to this dissertation as underlying inspiration and influential theory.

Theoretical. Much of the theoretical work related to MUSes has proven complexity bounds for the problems of recognizing or finding MUSes both in general and in particular classes of CNF formulas.

A central theoretical result was produced by Papadimitriou and Wolfe (74), who proved that recognizing a minimal unsatisfiable formula (i.e., "is the formula F 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 (unless, of course P=NP).

Certain subclasses of unsatisfiable formulas do have tractable solutions to these problems, however. Minimally unsatisfiable formulas have positive deficiency (1) (a formula's *deficiency* is its number of clauses minus its number of variables). Davydov et al. (26) showed that MUSes with deficiency 1 can be recognized in quadratic time. Büning (18) showed that for a fixed integer k, recognizing an MUS with deficiency k is in NP, and he presented a cubic-time algorithm for recognizing MUSes with deficiency 2. Kullmann (55) proved that recognizing a minimal unsatisfiable formula with deficiency k is decidable in polynomial time, and Fleischner et al. (34) showed that such formulas can be recognized in $n^{\mathrm{O}(k)}$ time, where n is the number of variables. Szeider (82) improved this result and presented an algorithm with time complexity $\mathrm{O}(2^k n^4)$. Note that the problem of *finding* an MUS within a formula is more difficult than recognizing a formula as an MUS.

Applied. Algorithmic work on finding MUSes has been done primarily in the fields of linear programming and Boolean satisfiability.

LP/ILP. Some of the earliest work on extracting unsatisfiable cores was done in the domain of linear programming (LP). Gleeson and Ryan (40) describe a method to compute all IISes of a linear program by producing a polytope whose vertices correspond to the IISes, though no implementation was provided. Chinneck and Dravnieks (20) contemporaneously developed several *filtering* routines ("deletion," "additive," "elastic," etc.) that can be used to iteratively remove constraints from a system until those that remain constitute an IIS. Guieu and Chinneck (43) later proposed and implemented methods for finding IISes of mixed-integer and integer linear programs (MILP), relying on additive and deletion methods as the core techniques, essentially manipulating the inclusion of constraints until a small (not necessarily minimal) infeasible subset of constraints is found. This work is interesting because it bridges a gap between two domains, finding MUSes (IISes) of constraint problems whose decision problems are NP-complete (MILP, along with Boolean SAT and others) with techniques informed by work done with LPs, whose solutions can be found in polynomial time. Now, most large commercial LP solvers contain methods for finding IISes.

SAT. Aside from the work on IISes of LPs, most research on extracting unsatisfiable cores was done in the domain of Boolean CNF formulas, and this is the domain in which most work in this dissertation has been done. These have numerous applications in the logic circuit industry, as circuits can be directly encoded into CNF to solve various problems related to design and verification.

In one of the earliest implemented algorithms for computing small unsatisfiable subformulas, Bruni and Sassano (16) utilize heuristics to estimate the *hardness* of each clause in a formula. Their algorithm then performs an adaptive search for a small core, expanding and contracting a core, guided by the hardness scores of each clause, until an unsatisfiable core is found.

Oh et al. (72) 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 MUSes if they favor variables that did not appear in previously found MUSes.

Huang (49) produced a "Minimal Unsatisfiability Prover" (MUP), which can be used to prove that an MUS is minimal and to find an exact MUS of the formula when it is not. It works by augmenting the formula with new variables such that every assignment to the additional variables corresponds to the removal of one clause from the original formula. Unlike AMUSE, which adds one variable per clause, MUP only needs $\lceil \log(m+1) \rceil$ variables, where m is the number of clauses; this reduces the search space of the augmented formula greatly. If the augmented formula has a model for each assignment to the new variables, then the formula is an MUS, as this is equivalent to saying that removing any one clause makes the formula satisfiable. MUP can also be used to produce an MUS extractor (positioned as a post-processor to a core extractor like AMUSE that may not produce a minimal result) by identifying and removing clauses whose removal does not yield a satisfiable formula.

As mentioned in the previous subsection, several researchers have used resolution tree proofs as the basis of extracting unsatisfiable cores. Goldberg and Novikov (41) 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 (87; 88) also use a resolution proof generated by the SAT solver to derive an unsatisfiable core, proposing the idea independently of and concurrently with Goldberg and Novikov. The leaf nodes (original clauses) in the transitive fan-in of the sink node (the empty clause indicating a conflict) are a subset of the original clauses that can be used to produce a conflict via resolution, thus they must be unsatisfiable by themselves. They noted that their procedure could be repeated, by taking an output core as an input to another iteration of the algorithm, to possibly produce a smaller core. This can be repeated until a fixed point is reached at which no further clauses are removed. This repetition can also be applied to any approach that produces small but not minimal cores.

Motivated by the idea that smaller cores are more useful, as they more concisely describe and isolate a conflict within a formula, Dershowitz et al. (29) and Gershman et al. (39) have both developed techniques for improving on the results produced by Zhang and Malik's resolution proof approach. Both produce smaller cores at the expense of higher runtime. Gershman, et al. focused more on the *velocity* of their algorithm (clauses removed per time unit).

Grégoire et al. (42) 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).

Other Constraint Types. Work on extracting USes and MUSes has been done beyond LP and Boolean SAT, as well. Junker (51; 52) developed algorithms for extracting “conflicts” (MUSes) and “relaxations” (complements of MCSes) of arbitrary constraint systems. Much of his discussion about relaxations and conflicts skirts on the edge of the hitting set duality between the two (described in the next subsection) without ever fully recognizing it. His methods are “non-intrusive,” in that they do not require modifications to an underlying solver or consistency checker to function and they can be applied to any solver for any constraint type. The algorithms he presents are based on the basic additive method (as described by Chinneck and Dravnieks (20) and others): add constraints one-by-one until the set is inconsistent, the final constraint added is a member of an MUS. Then the algorithms differ on how to determine which of the chosen constraints form an MUS and which should be removed. His QuickXplain algorithm (first presented in (51), then more succinctly in (52)) employs a novel divide-and-conquer partitioning approach, à la quicksort, to reduce the total number of consistency checks needed. QuickXplain has an added feature of producing “preferred” conflicts, given a total order of the constraints and an essentially lexicographic ordering of conflict sets based on it.

Recently, Cimatti et al. (21) extended the resolution proof US extraction procedures to find unsatisfiable cores (still not necessarily minimal) of SMT instances.

SMUSes. Some research has also been directed towards algorithms for finding a *smallest* MUS (SMUS) of a constraint system, all within the domain of Boolean SAT, as far as we are aware. Lynce and Marques-Silva (66) augment a formula with clause-selector variables (as in AMUSE (72) and others) and search the space of all unsatisfiable subformulas for one of minimum size (a formula may have multiple SMUSes of equal size). Even with a few optimizations, this algorithm runs up against the size of the space of all unsatisfiable subformulas: exponential in the number of clauses. Mneimneh et al. (70) utilize a more efficient branch-and-bound search along with a strong lower-bound heuristic to search the same space with far more pruning, leading to much better runtimes. Most recently, Zhang et al. (86) developed an approach to compute an SMUS using CAMUS (Chapter 2), an algorithm developed for finding all MUSes (described in detail in Chapter 2), coupled with a greedy genetic algorithm (GGA) to find small MUSes. Because the genetic algorithm is an incomplete local search, their approach cannot 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.

1.2.3 Multiple / All Unsatisfiable Subsets

Definition 4. *Minimal Correction Subset* (MCS): Given an unsatisfiable constraint system C, a set of clauses $M \subseteq C$ is an MCS if $C - M$ is satisfiable and $\forall C_i \in M,\ C - (M - \{C_i\})$ is unsatisfiable.

In earlier work, we used "CoMSS" (the Complement of a Maximal Satisfiable Subset) instead of MCS, but "minimal correction set" is both simpler and more expressive. Most confusingly, the equivalent of an MUS in the field of model-based diagnosis is sometimes called a *minimal conflict set* (again, MCS), while the equivalent of a minimal correction set is known as a *diagnosis*. We will use the terms and acronyms from Definitions 3 and 4 in this dissertation, as those are most common within this research and the most closely related fields.

Definition 5. *Hitting Set*: Given a collection Ω of sets from some finite domain D, a hitting set H of Ω is $H \subseteq D$ such that $\forall S \in \Omega,\ H \cap S \neq \emptyset$. Intuitively, every set in Ω is *hit* by at least one element in H.
A *minimal* or *irreducible* hitting set is a hitting set whose proper subsets are not hitting sets.

The problem of finding all minimal hitting sets of a collection of sets is directly equivalent to several problems such as hypergraph transversal, monotone dualization, and others (32).

MCSes can be understood as generalizations of Max-SAT solutions (cf. Section 1.2.4). Given any Max-SAT solution in the form of a satisfiable subset of clauses, we can look at those clauses left unsatisfied as a *correction set* (CS), because removing them from the formula *corrects* it, making it satisfiable. Due to the maximum cardinality of a Max-SAT solution, its corresponding correction set has minimum cardinality; no smaller correction sets exist. We generalize this to the concept of minim**al** correction sets: An MCS is a correction set such that all of its proper subsets are *not* correction sets. MCSes are minimal, or irreducible, but not necessarily minim**um**. Every Max-SAT solution indicates an MCS, but there can be more MCSes than those that are complements of a Max-SAT solution.

Theoretical. An important connection between MCSes and MUSes was first noted in the model-based diagnosis community (27; 76): Every MCS is a minimal hitting set of the complete set of MUSes. This can be understood by considering that the complement of an MCS is a maximal subset of a formula's constraints that is satisfiable, i.e., it contains no MUSes. Conversely, those constraints in the MCS form a minimal set that can be removed to "neutralize" every MUS in the formula, thus it must "hit" every MUS. The field of diagnosis is not concerned with constraints as such; rather, the problems of interest are systems of components with defined behaviors, observations of correct and/or incorrect outputs of those systems, and techniques for determining a set of components whose combined failure could have caused an observed erroneous output. However, this has a direct mapping into infeasible constraint systems and satisfiability: Map every constraint into a component and a "Satisfiable" result returned by a solver into an erroneous output; a diagnosis is thus a minimal set of constraints whose failure (removal from the constraint system) could yield the "Satisfiable" result, i.e., an MCS.

Diagnosis techniques such as those pioneered by de Kleer and Williams (27) and Reiter (76) use this relation between MCSes and MUSes to find diagnoses: they first find all MUSes (conflict sets, in their terms) and compute MCSes (diagnoses) as minimal hitting sets of those MUSes. Interestingly, some work on finding all MUSes of constraint systems has taken exactly the opposite approach, utilizing this converse of the relationship identified and used by those in the diagnosis community: Every MUS is a minimal hitting set of the complete set of MCSes. This was noted by Birnbaum and Lozinskii (11), though they were interested in MCSes only, noting this result as an interesting fact but not exploiting it to find MUSes.

Applied. As noted earlier, Gleeson and Ryan (40) proposed a technique to compute all IISes of a linear program. They did not provide an implementation, however, and their approach is very specific to linear programming.

One series of algorithms for finding all MUSes of general constraints began in diagnosis with Hou (48), who developed a method to produce all MUSes while avoiding the production and investigation of all subsets of constraints. Hou defines a "conflict-set" (C-S) tree that, unpruned, enumerates all subsets of constraints. Hou's algorithm traverses the C-S tree depth-first, checking every subset for consistency (satisfiability). Any inconsistent set whose subsets are found to be consistent is an MUS. The algorithm employs several pruning rules to reduce the size of the C-S tree. An error in Hou's algorithm was corrected and further improvements were made by Han and Lee (45). Finally, de la Banda et al. (28) enhanced the C-S tree concept yet further with several optimizations and modifications, implementing their algorithm for Herbrand constraints, a class of constraint system used in a software verification task for which all MUSes are needed.

For the same software verification task, Bailey and Stuckey (8) then developed a new system using the MCS/MUS relationship in the opposite direction, exploiting it to produce MUSes from MCSes instead of the other way around. Their algorithm, Dualize and Advance (DAA) was based on an interleaved approach, originally produced for a data mining task, that computes MCSes from "seed" satisfiable subsets and searches for minimal hitting sets of the MCSes found thus far. Each such hitting set will either be an MUS or a satisfiable subset of constraints that can serve as a new seed.

Interestingly, Cohn and Barnhart (22) independently solved a version of this problem, in an airline crew scheduling operations research domain, using the same general algorithm as Bailey and Stuckey. They implemented the equivalent of the MCS search and hitting set computation with OR techniques, interleaving the two in the same manner as DAA to find all MUSes ("minimally infeasible sets" in their terminology). As another example of such "convergent evolution," their overall approach to solving the crew scheduling problem employed what is essentially *abstraction refinement* as used in the hardware verification community (described in Section 4.1), using the computation of all MUSes in the same manner as our algorithm for computing all MUSes has been used in an abstraction refinement flow for verifying microprocessor designs.

CAMUS (Compute All MUSes), our system of algorithms for finding all MCSes and MUSes, was developed at the same time as DAA. It exploits the MCS/MUS relationship in this direction also, though in a serial, two-phase approach described in more detail below. An experimental comparison of CAMUS to DAA adapted to Boolean Satisfiability (63) showed that CAMUS outperforms DAA in this domain.

Gasca et al. (37) have developed methods for computing all MUSes of overconstrained numerical CSPs (NCSPs). NCSPs consist of numeric variables defined over the reals and constraints in the form of inequalities or equalities between linear or polynomial combinations of the variables. Their approach uses the idea of exploring all subsets of constraints with rules for pruning unnecessary collections of subsets based on several concepts of "neighborhood" defined for subsets of constraints using structure specific to NCSPs. Results show that their approaches are superior to the naive check-everything approach, but their ideas are tied tightly to the structure of their numeric variables and constraints and may not generalize readily.

1.2.4 MaxSAT / MaxCSP

Definition 6. Given an unsatisfiable constraint system, *MaxSAT / MaxCSP / MaxFS* (for SAT, CSP, and LP/ILP, respectively) is the problem of identifying an assignment that satisfies the maximum possible number of constraints.

These problems can also be seen as the problem of identifying the largest set of satisfiable constraints. Whether approached with a focus on the assignment or the set of satisfiable constraints, the methods for solving the problem are the same, as one directly identifies the other.

Definition 7. Given a Boolean CNF formula with a weight for every clause, *Weighted MaxSAT* is the problem of finding an assignment that minimizes the sum of the weights of the unsatisfied clauses

Definition 8. Given a Boolean CNF formula with some clauses marked "hard" and the rest marked "soft," *Partial MaxSAT* is the problem of finding an assignment that satisfies all hard clauses and the maximum possible number of soft clauses.

The Weighted MaxSAT and Partial MaxSAT problems can be combined, naturally, into *Weighted Partial MaxSAT*. Weighted MaxSAT has an analog in the domain of CSPs: Weighted CSPs (WCSPs). *Partial Constraint Satisfaction* is a different concept than Partial MaxSAT, and it is described below. The MaxFS problem also has a weighted analog.

The MaxSAT (or "maximum satisfiability") problem has been studied since the earliest work on Boolean satisfiability and the study of NP-complete problems. As optimization problems, MaxSAT and MaxCSP have been tackled with the range of optimization techniques *du jour*: branch-and-bound, simulated annealing, genetic algorithms, ant colony optimization, tabu search, random walk, etc. Here, we lay out some of the fundamental results and state-of-the-art algorithms.

SAT. Until recently, MaxSAT algorithms could only tackle "small" problems (relative to the problems solvable in reasonable time by contemporary SAT solvers). Much applied work developed incomplete algorithms, usually based on some form of local search, as complete solvers did not scale well. Recent work, however, has reached the point where modern algorithms can solve "medium"-sized problems. For example, MiniMaxSAT (46) incorporates many different techniques, including those developed for standard SAT solvers, in a complete branch-and-bound search that performs well on industrial benchmarks. MiniMaxSAT can also solve the Weighted-, Partial-, and Partial Weighted MaxSAT problems.

More recently, researchers have developed a number of algorithms exploiting a connection between unsatisfiable subsets of constraints and Max-SAT, all using the inexpensive resolution proof method for generating unsatisfiable cores. Fu & Malik first introduced the idea of using unsatisfiable cores to assist in solving Max-SAT in (36). They described an algorithm based on "diagnosis" that repeatedly finds a core by the resolution proof method, adds clause-selector variables to the clauses in that core, places a one-hot constraint on those clause-selector variables, and searches for a satisfying solution to the modified problem. Essentially, the algorithm identifies a core in each iteration that must be neutralized (by the removal of a clause) in any Max-SAT solution.

Marques-Silva, Planes, and Manquinho (67; 68; 69) improved upon Fu & Malik's algorithm, which they dubbed MSU1, with several refinements and optimizations. In (68) and (69), Marques-Silva and Planes introduce algorithms MSU1.1, MSU3, and MSU4[1]. The MSU1.1 algorithm is a variant of MSU1 with two important modifications. First, it uses a better encoding for the one-hot constraints, namely a BDD representation of a counter converted to CNF with several optimizations. Second, MSU1.1 exploits the authors' observation that the one-hot constraints can actually be AtMost(1) constraints, because the identified cores are unsatisfiable if no clauses are removed; thus, omitting the constraint that requires at least one clause be removed will not alter the results. The authors also

[1]we have adopted the algorithm naming from the most recent paper (67), which is slightly changed from earlier papers.

Algorithm	Cardinality Constraints	Cardinality Encoding
MSU1	Per-core One-Hot	Adder tree
MSU1.1	Per-core AtMost	BDD to CNF
MSU1.2	Per-core AtMost	Bitwise on variables
MSU2	Per-core One-Hot	Bitwise on clauses
MSU3	Single AtMost	BDD to CNF
MSU4-v1	Single AtMost	BDD to CNF
MSU4-v2	Single AtMost	Sorting networks

Table 1.1 Comparison of all MSU* algorithms

describe MSU3, which avoids some of the size explosion of the additional variables and clauses created by MSU1 by using a single clause-selector variable per clause and a single AtMost constraint over all of them. In (69), the authors further introduce MSU4, essentially a modification of MSU3 that exploits relationships between unsatisfiable cores and bounds on Max-SAT solutions. Finally, Marques-Silva and Manquinho introduce MSU1.2 and MSU2 in (67). MSU1.2 uses a bitwise encoding, with a logarithmic number of auxiliary variables, for each cardinality constraint, and MSU2 takes that a step further, employing a bitwise one-hot encoding on the clause-selector variables themselves.

A parallel development of the concept of using unsatisfiable cores for Max-SAT was done in the domain of logic circuit debugging/diagnosis by Sülflow, et al. (81). Without explicitly noting the connection to Max-SAT, they developed a new SAT-based debugging framework that exploits unsatisfiable core extraction. Though the terminology is different, and the theories and algorithms are often described in terms of gates instead of constraints or clauses, SAT-based debugging is essentially the process of solving Max-SAT for circuit-derived CNF instances. Though this connection was noted in (77), the concept does not seem to be widespread in the SAT-based debugging community.

One difference between the work of Sülflow, et al. and the MSU* algorithms is that the debugging framework produces *all* Max-SAT results (equivalent to finding all minimum-cardinality MCSes) by an iterative solving procedure. Their use of cores seems closest to MSU3, with a single cardinality constraint covering all identified cores; however, they treat non-overlapping cores with separate cardinality constraints, as this limits the size of the search space with little overhead. They do mention alternative approaches for producing cardinality constraints, including one which creates a separate constraint for every intersection of any subset of the cores, but they dismiss these as not outperforming their chosen approach in most of their experiments.

CSP. Freuder and Wallace (35) presented what they called *Partial Constraint Satisfaction Problems* (PCSPs), a framework that subsumes MaxSAT and MaxCSP, as a search through a space of related problems defined by their solution spaces. In this framework, MaxCSP, for example, becomes the problem of finding a solvable problem as close to the original problem as possible, measured by the number of removed constraints. The metric one optimizes can be more complex than this, however, which allows the framework to encompass more than just maximizing the number of satisfied constraints. They focused on algorithms for maximal satisfaction (MaxCSP, essentially), but the work is applicable to related problems of *sufficient satisfaction* (terminate upon finding a sufficiently good solution) and *resource-bounded satisfaction* (terminate with the best solution found thus far when a resource bound is reached). Later, two other frameworks encompassing MaxCSP, Weighted CSP, Partial CSP, and more were introduced, one based on a semiring (13), the other on an ordered monoid (78). Both interpret the various types of constraint problems in terms of these mathematical structures, using them to inform the understanding and development of related algorithms; the two frameworks are compared in (12).

A good example of a state-of-the-art WCSP (and MaxSAT) solver is Toolbar (15), which has performed well in several recent evaluations and competitions of solvers. Toolbar implements a depth-first branch-and-bound search along with routines to maintain various forms of local consistency (a technique for manipulating a problem to an equivalent but simpler form, generally by propagating information through the system), and much of the recent work on WCSP solvers has been in developing more advanced local consistency maintenance techniques.

1.2.5 Autarkies

Definition 9. *Autarky*: An *autarky* (or *autark assignment*) of a Boolean CNF formula is an assignment to a subset of a formula's variables that satisfies every clause containing one of the assigned variables.

Following the meaning of the term in other fields, an autarky is a *self-sufficient* partial assignment. Because the work in this dissertation involves autarkies in a system that trims clause sets, we will refer to autarkies in terms of the clauses they satisfy. Thus, the maximum autarky is the largest set of clauses satisfiable by an autarky, as opposed to the largest partial assignment.

Definition 10. *Pure Literals*: One simple form of autarky arises from *pure literals*. A pure literal is a variable that occurs in only one polarity (either always positive or always negated) in a CNF formula.

Pure literals can be found in a linear time scan of a formula. Removing the clauses satisfied by pure literals may cause other literals to become pure in the formula, so repeatedly detecting, recording, and removing pure literals is a simple first step for any algorithm that finds autarkies. The process terminates when the formula no longer contains pure literals.

Theoretical. Monien and Speckenmeyer (71) first introduced the concept of an autark assignment or autarky, using autarkies in a modification of the DPLL satisfiability algorithm (24; 25) that reduced its complexity upper bound below 2^n splitting steps (for a formula with n variables).

More recently, Kullmann has investigated autarkies in several papers. In (56), he introduces the idea of *lean clause-sets*, sets of clauses that have no autarkies. The largest lean clause-set is the complement of the maximum autarky of a formula; all clauses can be partitioned into one or the other. Kullmann investigates a special case of autarky that can be found in polynomial time using linear programming, though this does not generalize to finding all autarkies. He also proves, with Theorem 3.16, that a set of clauses F is lean "if and only if every clause of F can be used by some resolution refutation of F." Conversely, a set of clauses $A \subseteq F$ is an autarky if and only if each clause in A can *not* be used in any resolution refutation of F.

Applied. Autarkies were used in a satisfiability algorithm by Van Gelder (83) named Modoc. Modoc integrates autarky pruning, removing those clauses satisfied by autarkies, into a resolution-based model elimination approach to satisfiability. Both Monien and Speckenmeyer's algorithm and Modoc find autarkies as side-effects of their operation, but neither is aiming to find the maximum autarky. Additionally, both find many more "conditional autarkies," i.e., autarkies that appear after propagating a partial assignment through the formula, than "top-level autarkies" for the entire formula.

Later, in (57), Kullmann uses this fact to develop an algorithm for computing the maximum autarky. Using a SAT solver that provides a resolution refutation for unsatisfiable instances, one can iteratively remove the variables included in some resolution proof. When the reduced formula becomes satisfiable, the satisfying assignment is an autarky of the original formula.

Finally, Kullmann, et al. (58) use both autarkies and MUSes as tools to describe and examine unsatisfiable formulas. They characterize clauses in such formulas into several classes based on each clause's involvement in MUSes, resolution refutations, and autarkies. Clauses contained in every MUS are called "necessary"; those in any MUS are "potentially necessary"; "usable" indicates a clause is in some resolution refutation; and thus "unusable" refers to clauses in an autarky. Complements and intersections of these classes are defined as well. They experimentally evaluate a set of industrial benchmarks from an automotive product configuration domain (80), reporting on the MUSes and clauses in the different levels of "necessity" in each instance. To compute all MUSes of the instances, they use CAMUS, and they found maximum autarkies using the algorithm described in (57), implemented using the ZChaff SAT solver's ability to produce resolution refutations (87).

1.3 Thesis Contributions

The contributions of this dissertation can be broadly categorized under the banner of advancing the field of analyzing infeasible constraint systems, and they fall roughly into the following categories.

Synthesis of Existing Work Throughout this research, connections often arose to existing work in diverse domains. Much of the work on infeasibility analysis, including some not explicitly recognized as such, is done in a domain-specific manner, focusing on implementations and implications for a particular application or type of constraint system; however, the work is often much more broadly applicable. This research has mainly been done in the domain of Boolean Satisfiability, for example, but an effort has been made to note the general applicability of any concepts or algorithms developed. Many of the connections found were not described elsewhere in the literature, and one of the contributions is thus to bring together the pieces, linking them explicitly and illuminating the connections for the benefit of other researchers. In this dissertation, many of these connections are described in this chapter in Sections 1.1 and 1.2. Other connections are mentioned as appropriate throughout the remaining chapters.

Algorithm Development In the pursuit of answers to the overarching question, "How can we analyze infeasible constraint systems?" this work has developed several algorithms to perform various analyses. All of these algorithms extract information about the infeasibility of a set of constraints, and the primary distinction between most of the related work and that here is that this involves algorithms that produce *complete* views of infeasibility. Existing work has mostly been concerned with extracting single, approximate unsatisfiable cores of constraint systems. In contrast to these partial, approximate views, the algorithms here produce complete views that enable much more detailed analyses of infeasibility. Much of the implementation has been specific to Boolean Satisfiability, but most of the algorithms are applicable to constraint systems in general. A foundational set of algorithms for computing both complete sets of MCSes and complete sets of MUSes (60; 62; 63; 64) is presented in Chapter 2. This is followed by several variations and additions to the base set in Chapter 3. We developed modified algorithms that relax the completeness criteria in order to avoid the potential intractability of computing complete sets of results (64), an extension that groups constraints for performance or to represent high-level constraints (64), and a variant that finds a smallest MUS of an instance (61). Further, we developed a novel algorithm for computing autarkies, using it to find and prune autarkies from an instance before applying the other algorithms (65), exploited unsatisfiable cores to boost the search for MCSes as a generalization of existing work on using unsatisfiable cores in Max-SAT, and investigated the potential of using symmetry information from a constraint system in order to aid the search for MCSes and to potentially compress sets of MCSes and MUSes.

Specific Applications Through collaborations with other researchers, these algorithms have been applied to industrially-relevant problems. Infeasibility analysis has found applications in several logic circuit verification tasks, and the algorithms in this work have been applied in particular to two of them. Both applications, described in Chapter 4, have shown impressive performance gains due directly to the application of this work, specifically relying on the completeness of the analyses the algorithms produce. The first, Reveal, a system for equivalence checking of Verilog circuit designs, uses MUSes in an abstraction refinement loop, and experiments have shown that using all MUSes of a generated constraint system is crucial for performance (3; 4; 5). The second application is a circuit error debugging system in which the desired output, candidate error locations, can be found by extracting MCSes of Boolean formulas generated from a circuit; to find all such candidates, all MCSes must be found (77).

Expanding Knowledge about the Structure of Infeasibility Woven throughout this work are a variety of simple but valuable observations about the nature of infeasibility in constraint systems. When this research began, knowledge about the structure of infeasibility was minimal and scattered in the literature. Early research presented several surprises, such as the simple fact that infeasible constraint systems can have a massive, in fact intractably exponential, number of "causes" of their infeasibility. This work, as some of the most detailed research looking at *complete* analyses of infeasibility, has brought ideas like this to light, not only for theoretical, pathological cases such as presented in Appendix A, but also in a large number of industrial instances as well.

Tool Development Several of the algorithms developed here are implemented in tools that are made available to any who request them. The tools enable other researchers to 1) investigate the structure of the infeasibility in constraint systems arising in their own work, making use of the *complete* analyses the algorithms produce, 2) determine the potential of applying infeasibility analyses to their research, and 3) extend the work or compare their own to it in order to further the field of infeasibility analysis.

Chapter 2
Foundations of CAMUS

This dissertation is centered around a system of algorithms called CAMUS (**C**ompute **A**ll **M**inimal **U**nsatisfiable **S**ubsets - pronounced “ka-**moo**” after the French writer). CAMUS is a set of sound and complete algorithms that, as the name indicates, produce a complete set of MUSes for a given unsatisfiable constraint system. Additionally, and as an integral intermediate result, CAMUS computes the complete set of MCSes for the constraint system. In this way, CAMUS provides two complete views of the infeasibility of a constraint system, each a different view of the same structure. CAMUS is a platform upon which we have developed further extensions and which we have integrated into real-world applications. This chapter lays out the theoretical foundations of CAMUS, describes its base algorithms, discusses empirical results illustrating its performance, and explains how it is generalizable to many types of constraint systems.

2.1 MUS/MCS Duality

An important connection between MUSes and MCSes was noted independently by Bailey and Stuckey (8), Birnbaum and Lozinskii (11), and Liffiton, et al. (63; 62). This relationship is the foundation of much of this work. 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 (11) 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 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 2.1 for an 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

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

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 2.1 Covering Problems Linking MCSes(φ) and MUSes(φ)

the minimal hitting sets.

In practice, it is easier to find satisfiable subsets of constraints than unsatisfiable subsets; thus, finding MCSes (equivalent to finding their complementary Maximal Satisfiable Subsets (MSSes)) is easier than finding MUSes directly. This follows from the relative simplicity of problems in NP (e.g., SAT), for which a single "solution" must be found, as compared to those in Co-NP (e.g., UNSAT), for which all "solutions" must be checked. 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 the algorithm 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 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. 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, 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.

2.2 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

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

Figure 2.2 Algorithm for finding all MCSes of a formula φ

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 2.2.

The implementation for Boolean satisfiability is integrated directly with a modern SAT solver (specifically MiniSAT (31) 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 1.1.2.

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 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[1]. 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 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 solutions will have most clause-selector variables set

[1]As opposed to, for example, linear constraints over the reals such as in (47), where the slack variables already used by the SIMPLEX algorithm can be used to similarly deactivate constraints.

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 the example formula in Figure 2.1. 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.

2.3 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 2.5.3).

The following subsections describe 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 chapter.

2.3.1 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 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 is taken here 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.

Figure 2.3 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

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

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 the 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 2.4. 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.

```
SingleMUS(MCSes)
1.  MUS ← $\emptyset$
2.  while (MCSes ≠ $\emptyset$)
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 2.4 Algorithm for computing a single MUS from a set of MCSes

```
AllMUSes(MCSes, currentMUS)
1.  if (MCSes = $\emptyset$)
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 2.5 Algorithm for computing the complete set of MUSes from a set of MCSes

2.3.2 Computing All MUSes

The algorithm for computing all MUSes was developed 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 2.4) 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 2.5.

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 2.1, 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 one 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 algorithm. Our implementation uses 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 product configuration benchmarks reported in Section 2.4.

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 (25; 24)) 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.

2.4 Performance

Table 2.1 contains experimental data produced with a set of CNF benchmarks from an automotive product configuration domain (Appendix B.1). We set a 600 second timeout on each phase of CAMUS. It was able to complete the stage of finding all MCSes within this

	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

Table 2.1 Experimental results for automotive product configuration benchmarks

timeout for 49 of the 84 instances. Table 2.1 reports on 39 of these 49 instances[2]. 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 that the output size was the primary factor in the hardness of those instances.

The first column of Table 2.1 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 (see Appendix A). 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

[2]The 10 instances excluded from this table all matched very closely in terms of runtimes and output to at least one instance that is included in the table. The full results for the benchmark set, as well as additional data such as MUS sizes and results for other benchmarks, are available online at: http://www.eecs.umich.edu/~liffiton/camus/

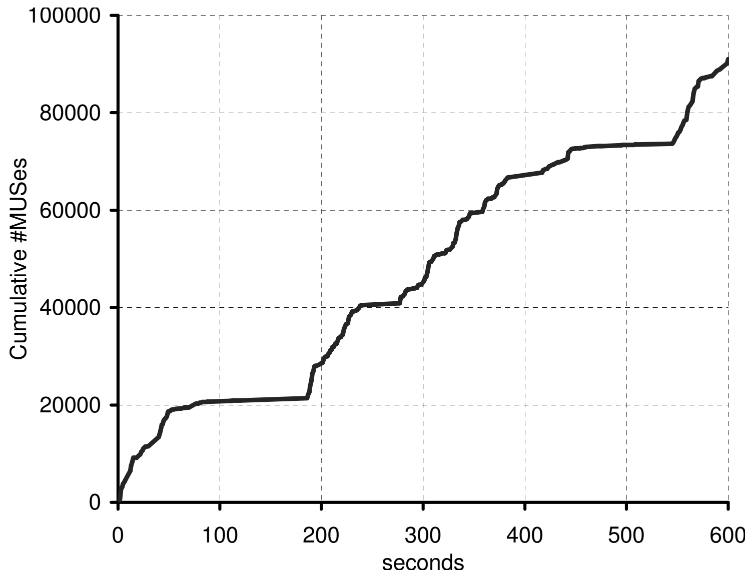

Figure 2.6 Anytime graph of computing MUSes for instance C208_FA_RZ_43

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 2.1 shows that, for these instances, the second phase of CAMUS generated close to 26,000 MUSes per second on average. Figure 2.6 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 the 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 one can generate pathological cases for which they do not hold (see Appendix A).

2.5 Comparison to Existing Work

The related research can be split into 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 looked 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.

2.5.1 All MUSes

Aside from the many algorithms for finding a single US or MUS, the work that is most relevant to this research, which addresses the problem of finding *all* MUSes, is the Dualize and Advance (DAA) algorithm by Bailey and Stuckey (8). They developed DAA for finding all MUSes of a system of Herbrand constraints and applied it to a software verification problem, namely the problem of type-error debugging of Haskell code.

DAA interleaves the use of the hitting-set duality with the search for what we call MCSes. Whereas CAMUS finds all MCSes before finding hitting sets of them, DAA computes hitting sets on a partial set of MCSes after finding each MCS – it outputs any MUSes found at that stage and also uses the results to direct the search for the next MCS.

We performed an experimental comparison of DAA with CAMUS. DAA was implemented by Baily and Stuckey for Herbrand constraints, though the basic algorithm applies to any type of constraint system, so we had to create a version for Boolean satisfiability. To keep things as fair as possible, we used the same SAT solver and the same basic solving methods in DAA as in CAMUS.

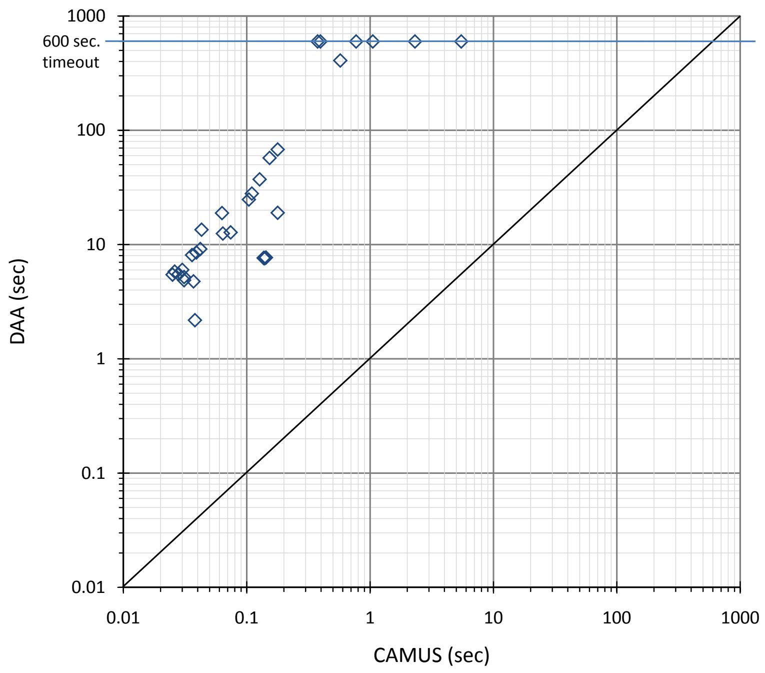

Figure 2.7 Comparing DAA and CAMUS (automotive product configuration benchmarks)

We compared DAA to CAMUS on the automotive product configuration benchmarks (Appendix B.1), using both to attempt to generate the complete set of MUSes for each instance. The tests were run in Linux on a 3.0GHz Intel Core 2 Due E6850 with 3GB of physical RAM. A 600 second timeout was set for every instance. Figure 2.7 displays the results of the comparison for the 32 out of 84 benchmarks on which either algorithm completed within the 600 second timeout. CAMUS found all MUSes for the 32 instances, while DAA completed 26; the relative runtimes show that CAMUS is consistently faster by several orders of magnitude. We have learned that further improvements could be made to the published description of Bailey and Stuckey's algorithm (personal communication, J. Bailey, October 2005), but it is unlikely that they would completely erase the performance gap.

The performance numbers paint a clear picture that CAMUS is faster than DAA for Boolean constraints. However, the performance of each algorithm is dependent on a number of factors.

One difference contributing greatly to the performance of CAMUS is its tight integration with a modern SAT solver. By formulating the problem with clause-selector variables, the SAT solver handles the search for MCSes itself. Additionally, by finding multiple MCSes (of a single size) within a single search tree, CAMUS immediately takes advantage of all of the features of modern SAT solvers, especially learned clauses. While DAA can use an incremental search within the search for a single MCS, it must restart the search after any added constraint makes the growing MCS unsatisfiable. It also restarts with a new search tree for every MCS, as compared to CAMUS which only restarts the search after all MCSes of a particular size have been found.

Note that while CAMUS is more heavily integrated with a SAT solver, it is still fairly independent of the particular solver itself. It can be implemented with any SAT solver that provides an incremental solving interface, allows the addition of constraints mid-search, and supports the AtMost constraint. (While the last requirement is not standard, its implementation in MiniSAT is quite simple, and as noted earlier, the effect can be obtained by modifying other SAT solvers with little difficulty.) The DAA algorithm simply calls a standard solver as a subroutine, making it even simpler to implement with different solvers.

Another large difference between CAMUS and DAA is the distinction between CAMUS' serial, two-phase algorithm and DAA's interleaved approach. Obtaining MUSes before computing the entire set of MCSes is beneficial in applications that do not require the complete set of MCSes nor all MUSes because it can provide results sooner. The interleaved approach could easily be adopted in CAMUS. Hitting sets of the partial set of MCSes could be calculated after every MCS is found, between stages of the incremental search (when incrementing the bound on MCS size), or at any desired interval. This could add a great deal of overhead, however, especially if every potential MUS had to be checked for unsatisfiability. This seems to be the case for DAA, as the instance for which it had its fastest runtime also had the fewest MCSes, and thus the fewest incremental hitting-set calculations. Though CAMUS could be interleaved to potentially gain efficiency, DAA could not be "deinterleaved," as it depends on the set of potential MUSes to provide the seed

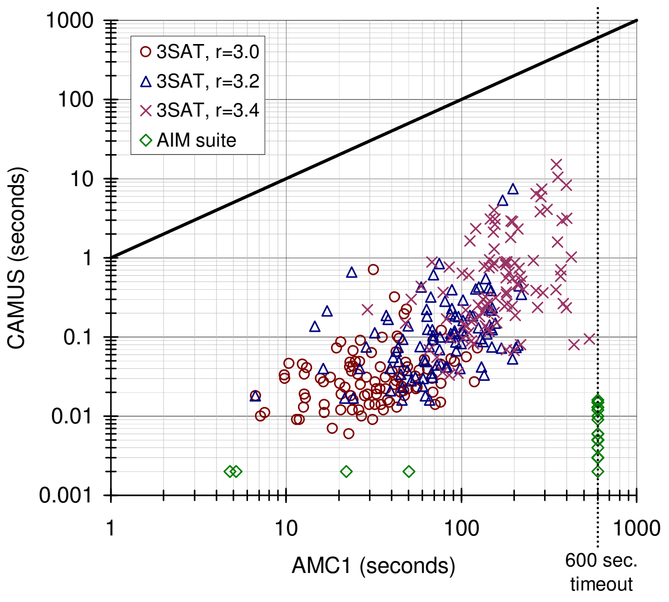

Figure 2.8 Comparing AMC1 and the first phase of CAMUS (3SAT instances have 30 variables)

for the next iteration and to determine when it has found all MCSes.

2.5.2 MSSes and MCSes

Maximal satisfiable subsets have been studied *apart* from their application to finding MUSes by Birnbaum and Lozinskii (11). 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.

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[3]. Figure 2.8 contains a comparison of the runtimes of our implementation of AMC1 with the first phase of CAMUS on random 3SAT instances (Appendix B.8; each with 30 variables and clause/variable ratios (r) as indicated in the legend) and unsatisfiable instances from the AIM benchmarks (Appendix B.4). We selected these smaller benchmarks for this comparison because AMC1 could not scale to solve the larger industrial instances. 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 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.

2.5.3 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 (33) for an

[3]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 2.8.

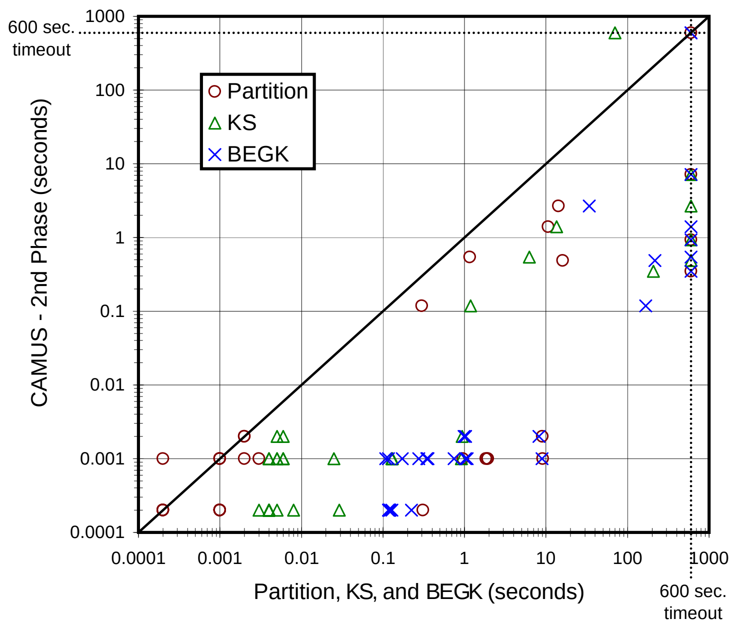

Figure 2.9 Comparing Partition, KS, and BEGK to the second phase of CAMUS on MCSes from automotive product configuration benchmarks

overview of applications of hypergraph transversals and some theoretical complexity results.

Our minimal hypergraph transversal algorithm for computing MUSes was created from first principles independently of existing algorithms. Since creating this algorithm for our purposes, we looked for other algorithms for learning and comparison purposes. Of the other algorithms found, those with the most efficient implementations are *Partition*, by Bailey, et. al. (7); an algorithm by Kavvadias and Stavropoulos (53; 54) (KS); and another by Boros, et. al. (14) (BEGK). The three algorithms were experimentally compared in (54), 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[4] to compare their performance to that of CAMUS' second phase on sets of MCSes.

Figure 2.9 compares the runtimes of Partition, KS, and BEGK against the MCSes algorithm in CAMUS on the sets of MCSes from the automotive product configuration benchmarks. We have only included benchmarks for which at least one of the algorithms

[4]James Bailey provided 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.

finished within the 600 second timeout. Each point plots the runtime in seconds of MCSes (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, MCSes either matches or outperforms the others: on the MCSes of C202_FS_SZ_104, KS completed in 70 seconds while MCSes (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 this 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 (54) 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 a hash table of seen branches. 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 (9). 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 using a similar incremental approach with MAXSAT to find all minimal hitting sets showed that it performs much worse than the algorithms presented here. It is likely that any procedure using repeated search/optimization to find all minimal hitting sets will not scale well.

2.6 Other Constraint Types

The results of the previous section were generated by the primary implementation of CAMUS, which operates on Boolean satisfiability instances. The algorithms in CAMUS generalize easily to other types of constraint systems, as well. Here, we describe the requirements of such generalizations and describe two different implementations we developed for other constraint types.

The second phase of CAMUS operates entirely independently of the first phase, and its input is nothing more than a collection of sets of discrete elements, which may as well be anonymous integers. Therefore, the MUSes algorithm can be reused in any implementation of CAMUS with no changes whatsoever. Additionally, other hypergraph transversal algorithms can be used in its place easily. The first phase of CAMUS does operate on the constraints directly, and this it is the piece for which changes must be made to tackle different constraint types.

As described, the MCSes algorithm uses clause-selector variables and the solver's ability to handle AtMost constraints and an incremental solving method. The incremental solving is a performance optimization, however, and the algorithm only relies on clause-selector variables and AtMost constraints. Both of these can be implemented in many different types of constraints. For example, in an integer program, the constraint $\sum_i a_i x_i \leq b$ can be augmented with a new binary variable y to form $cy + \sum_i a_i x_i \leq b$, where $c > b$. Other constraint types can be similarly augmented with variables for which particular assignments either satisfy the constraint or propagate out to imply the original constraint, pre-augmentation.

AtMost constraints can likewise be directly expressed, e.g. the definition of an AtMost constraint given in Section 1.1.2 is an integer programming constraint, or they can be encoded into the underlying constraint type, as in the CNF encodings of cardinality constraints described in (79). With either of these options, CAMUS can be implemented on top of an existing constraint solver with no modifications to the solver itself. Better performance may be obtained, however, by adding a "native" AtMost constraint to the solver, as we did with MiniSAT in our implementation of CAMUS. As only one AtMost constraint is needed at any given time, the implementation needs only keep a single counter, incremented when one of the variables in its scope is assigned. Any backtracking solver with some form of propagation can then propagate the negation of the remaining literals when the AtMost bound is reached.

Our first alternative implementation of CAMUS follows the approach of making minor modifications to an existing solver. In joint work with Michael Moffitt (62), we adapted

Maxilitis, a solver developed for the Max-CSP problem for Disjunctive Temporal Problems (DTPs) to find all MCSes. DTP constraints are defined as disjunctions of difference inequalities: $\bigvee_i x_i - y_i \leq b_i$. Maxilitis, as it existed prior to this work, already utilized a form of clause-selector variables, and the AtMost constraint was built directly into the solver, as described above.

The basic form of MCSes can be implemented with any solver that provides some way of maximizing the number of satisfied constraints and avoiding previous solutions. Our second alternative implementation of CAMUS, in the Satisfiability Modulo Theories (SMT) domain, performed well following this approach. As described in more detail in Section 1.1, every constraint in an SMT instance is a disjunction of predicates from any number of "background theories." A clause-selector variable can be added naturally to any SMT constraint by adding a new disjunct to a constraint that is simply a single, new variable, just as we do for CNF instances. AtMost constraints can be expressed as long as the SMT solver supports integer inequalities. We found, experimentally, that YICES (30), the SMT solver we used, performed better when using its own built-in Max-SAT functionality than when we added clause-selector variables and AtMost constraints ourselves.

The YICES-based SMT implementation of CAMUS, then, calls the Max-SAT routine in YICES repeatedly to generate MCSes. YICES supports weighted constraints, which allows for blocking constraints to be made "hard," such that the Max-SAT routine must satisfy them. A blocking constraint is created for every MCS found by creating a disjunction of the disjuncts in all constraints in the MCS. This requires that at least one of the constraints in the MCS be satisfied in any later solution, just as blocking clauses created from clause-selector variables do.

Note that SMT subsumes both Boolean SAT and DTPs (difference inequalities are a common "background theory" for SMT solvers). However, the SAT-specific and DTP-specific implementations of CAMUS outperform the more general YICES-based version. Domain-specific implementations definitely have their place, but the SMT implementation is a good general-purpose tool for exploring infeasibility in domains for which a specific implementation does not exist, such as linear or integer programming.

Chapter 3
Extending CAMUS

Over time, CAMUS has been extended with new capabilities to solve additional problems related to finding all MCSes and MUSes. Note that many of these modifications are made to the general CAMUS algorithms, not any specific implementation, and thus they can be applied to any specific implementation for a given type of constraint.

3.1 Relaxing Completeness

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 (see Appendix A for details). 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 Section 2.2 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 that pruning can in fact 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.

Figure 3.1 contains pseudocode for a modified MCSes algorithm, called 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 11. 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

```
PCSes($\varphi$)
 1.  $\varphi' \leftarrow$ AddYVars($\varphi$)
 2.  k $\leftarrow 1$
 3.  PCSes $\leftarrow \emptyset$
 4.  while (SAT($\varphi'$))
 5.      $\varphi'_k \leftarrow \varphi' \wedge \text{AtMost}(\{\neg y_1, \neg y_2, \ldots, \neg y_n\}, \texttt{k})$
 6.      while (newMCS $\leftarrow$ IncrementalSAT($\varphi'_k$))
⋆7.          (keptClauses,removedClauses) $\leftarrow$ Truncate(newMCS)
⋆8.          newPCS $\leftarrow$ keptClauses
⋆9.          $\varphi'_k \leftarrow$ RemoveClauses($\varphi'_k$, removedClauses)
⋆10.         $\varphi' \leftarrow$ RemoveClauses($\varphi'$, removedClauses)
 11.         PCSes $\leftarrow$ PCSes $\cup$ {newPCS}
 12.         $\varphi'_k \leftarrow \varphi'_k \wedge$ BlockingClause(newPCS)
 13.         $\varphi' \leftarrow \varphi' \wedge$ BlockingClause(newPCS)
 14.     end while
 15.     k $\leftarrow$ k + 1
 16. end while
⋆17. PCSes $\leftarrow$ RemoveSubsumed(PCSes)
 18. return PCSes
```

Figure 3.1 A generalization of MCSes, capable of finding PCSes (Partial Correction Subsets) of a formula φ [A $\star$ indicates a line not in MCSes]

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[1].

[1]Another way to handle this conflict is to remove any clauses in the `removedClauses` set from any

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 behaviors can be 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

previously computed PCSes as well. We have implemented the approach that prevents previously kept clauses from being removed.

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.

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

	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 3.2 Running PCSes on an example formula φ (see Fig. 2.1) – three separate runs with the truncation limit set to 1 kept clause, biasing clause selection by previous selection frequency

Figure 3.2 shows the execution of this variant of PCSes on the example formula from Figure 2.1. The algorithm is run three times with a truncation limit 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.

3.1.1 Performance

To demonstrate the value of the PCSes algorithm, we used it to find MUSes of the automotive product configuration benchmarks (Appendix B.1) for which MCSes times out after 600 seconds without producing all MCSes. 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 3.1 lists results on the 35 automotive benchmark instances that time out in MCSes. We report the runtime of PCSes in seconds and the number of MUSes constructed from the

	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	$> 2.7 \cdot 10^6$

Table 3.1 Using PCSes to compute MUSes of the more difficult product configuration benchmarks

PCSes found for two different truncation bounds. We have not reported the runtime of the AllMUSes algorithm in this case; the strong correlation between its runtime and the number of MUSes produced has already been established. 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**, 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. 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 3.1 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., (72) and (87)). 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).

3.2 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 (50) and the CLU logic (17) 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 (3) (which uses the CLU logic), because the running time of CAMUS was unusably high without this optimization.

3.2.1 Performance

The effectiveness of constraint grouping is demonstrated using a set of benchmarks from Reveal, a hardware design verification system (Appendix B.2). As described in the Appendix, The Reveal flow (3) 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 $[x, y, \text{ and } z]$ 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 constraint grouping and computing MUSes

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

Table 3.2 Computing MCSes for Reveal benchmarks using constraint groups (“G”) and without (“noG”)

of the high-level constraints directly. Table 3.2 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 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 automotive product configuration benchmarks used in earlier experiments with MCSes, 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 (4).

3.3 Finding Smallest MUSes

Minimal Unsatisfiable Subsets are of interest mainly because of their minimality, because they provide concise, compact representations or explanations of infeasibility. Following this logic, a *smallest* MUS (SMUS) can be seen as the ultimate result in conflict-explanation. This is not always the case, as SMUSes are defined by their minimum cardinality, which doesn't take into account the complexity of the included constraints, nor does the size necessarily relate to ease of understanding or analysis, but the SMUS presents a reasonable goal.

A constraint system may, and often will, contain multiple MUSes of minimum cardinality. In these cases, we are interested in searching for any one of them, a single SMUS out of the many.

CAMUS can be used to find an SMUS in a trivial fashion by generating all MUSes and selecting the smallest. Better performance, however, can be achieved with a variant of CAMUS, 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 2.1 to generate all MUSes of a given formula. 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. For finding an SMUS, 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.

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.

The second phase of CAMUS-min calculates 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. For the approximation, we use a greedy heuristic called MIS-quick (44) 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 3.3 illustrates an example of using this lower bound to prune portions of the recursive tree and return an SMUS. 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

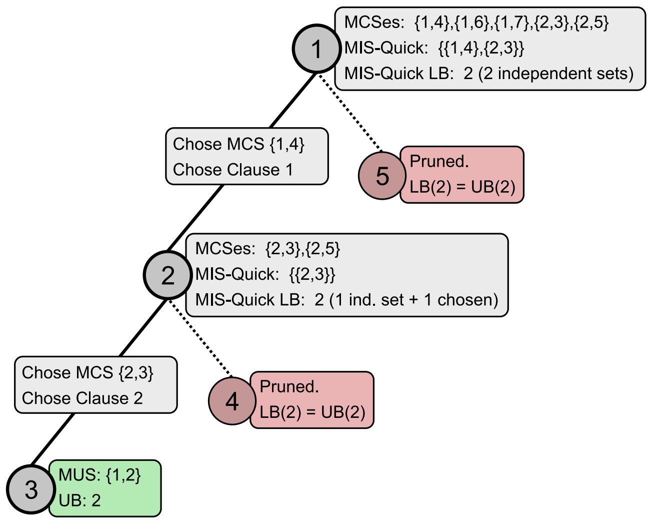

Figure 3.3 The operation of the recursive second phase of CAMUS-min using MIS-quick to compute lower bounds and prune subtrees.

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.

CAMUS-min can be used to find an SMUS directly, but it is even more useful as a component of another algorithm for finding SMUSes by Mneimneh, et al. (70; 61). The algorithm, Digger, uses some consequences of the same MCS/MUS duality used by CAMUS in order to split a CNF formula into more tractable subformulas and to compute strong upper and lower bounds on the size of an SMUS. CAMUS-min is used inside Digger to compute SMUSes of those smaller subformulas that Digger produces. An initial implementation of Digger (70) used CAMUS to enumerate all MUSes of each subformula and selected the smallest, while a later implementation (61) used CAMUS-min, along with other improvements, to gain orders of magnitude speedups over the first.

3.4 Finding / Pruning Autarkies

Autarkies provide another tool for looking into the structure of an unsatisfiable formula; they essentially provide information about portions of the formula that can be considered independent of the infeasibility. Autarkies have recently been linked to MUSes in (58), where Kullmann, et al., develop a classification of clauses in Boolean formulas based on their involvement in MUSes, autarkies, and resolution refutations. They use CAMUS and the only existing full approach for finding autarkies of which we are aware (first introduced in (57)) to investigate the complete set of MUSes and the autarkies, respectively, of a set of industrial benchmarks. They do not report runtime results, and we are not aware of any other experimental research on algorithms for finding the largest, or maximum, autarky of an instance.

In (58), the authors suggest two directions of research that are undertaken in this work:

1. An algorithm that directly searches for autarkies could be developed and compared to their algorithm, which makes use of a “duality” between autarkies and resolution refutations to find autarkies indirectly.
2. As clauses involved in autarkies are never contained in any MUS, such clauses can be removed as a preprocessing step for computing MUSes of a formula. (This also holds

for MCSes, as they are comprised of the same clauses as MUSes.)

We have developed a novel algorithm, named Sifter, that directly performs a complete search for maximum autarkies, and we compare it to the existing approach based on resolution proofs. We also investigate the use of this algorithm as a preprocessing step to trim autarkies from unsatisfiable instances before searching for MUSes or MCSes.

The approach taken here to the problem of finding the maximum autarky for a formula treats it as an optimization problem. We search for the largest partial assignment that satisfies the clauses it touches, i.e., the largest autarky, by explicitly searching in the space of all partial assignments and maximizing the size of the result (in terms of the number of satisfied clauses). Specifically, we "instrument" the formula to give a standard SAT solver the ability to enable and disable individual clauses and variables within its normal search, and we use AtMost constraints to perform a sliding objective maximization of the autarky size. This draws inspiration from the similar technique we employ for finding MCSes, which uses a less-involved instrumentation and the same optimization technique to allow a SAT solver to search for maximal satisfiable subsets of clauses. This can directly exploit the efficiency gains made in SAT solvers in recent years by using an "off-the-shelf" solver; the algorithm works with any solver[2], so it can benefit from future improvements as well.

3.4.1 Algorithm

To give a SAT solver the ability to search for autarkies, we instrument a formula C with the following modifications:

1. We replace every literal in the formula with a *literal-substitute*; x_j in the formula becomes x_j^1, while $\neg x_j$ is replaced with x_j^0.
2. Each clause C_i is augmented with a *clause-selector* y_i to form a new clause $C_i' = (y_i \rightarrow C_i) = (\neg y_i \vee C_i)$.
3. We create a *variable-selector* x_j^+ for every variable x_j. When x_j^+ is TRUE, x_j will be enabled, and it is disabled otherwise. For every variable x_j, we add clauses to relate its variable-selector x_j^+, its two literal-substitutes x_j^0 and x_j^1, and the value of the variable itself, x_j. In short, we want each literal-substitute to be TRUE when the variable is

[2]SAT solvers that implement AtMost constraints internally will likely perform better than those that require using a CNF encoding of them, but all will work.

enabled (x_j^+ is TRUE) and x_j has the corresponding value. This leads to new clauses encoding the following: $(x_j^1 = x_j^+ \wedge x_j)$ and $(x_j^0 = x_j^+ \wedge \neg x_j)$.

4. Finally, we add clauses to require that a clause be enabled (y_i = TRUE) if any one of its variables is enabled. Thus, for any x_j present in clause C_i, we add a clause $(x_j^+ \rightarrow y_i) = (\neg x_j^+ \vee y_i)$.

This is not the only option for instrumenting the formula; other encodings have the same effect. However, while preliminary experiments showed that similar encodings yield slightly different runtimes, the differences in efficiency were not substantial.

The complete instrumented formula for any example formula is too large to be useful here, but here we show the constraints produced from a single clause, assuming it is C_2, the second clause in the formula:

$$C_2, (\neg x_1 \vee x_2) \Longrightarrow \left\{ \begin{array}{ll} \text{1 \& 2:} & (\neg y_2 \vee x_1^0 \vee x_2^1) \\[1em] \text{3:} & \begin{array}{l} (x_1^1 = x_1^+ \wedge x_1)(x_1^0 = x_1^+ \wedge \neg x_1) \\ (x_2^1 = x_2^+ \wedge x_2)(x_2^0 = x_2^+ \wedge \neg x_2) \end{array} \\[1em] \text{4:} & (\neg x_1^+ \vee y_2)(\neg x_2^+ \vee y_2) \end{array} \right\}$$

The clause derived from modifications 1 and 2 replaces the original clause, while the rest are additions. The clauses from modification 3 (presented in shorthand as equalities; each is three clauses in CNF) are specific to variables, and the complete formula will only contain each set once per variable. The final two clauses, resulting from modification 4, are specific to C_2.

With the formula instrumented in this way, any satisfying assignment will indicate an autarky of the original formula. The x_j^+ variables indicate which variables are "activated," i.e., included in the autarky; the original variables contain the autarky assignment; and the clauses satisfied by the autarky are represented by those y_i variables set to TRUE. One such assignment is the trivial solution in which all variables and all clauses are disabled. To find the maximum autarky, we must maximize the number of enabled clauses.

```
Sifter($C$)
 1. ($C$, autarky) $\leftarrow$ PureLits($C$)
 2. $C' \leftarrow$ Instrument($C$)
 3. bound $\leftarrow |C| - 1$
 4. loop
 5.     $C'_b \leftarrow C' \wedge$ AtMost($\{\neg y_1, \neg y_2, \ldots, \neg y_n\}$, bound)
 6.     (isSAT, model) $\leftarrow$ Solve($C'_b$)
 7.     if not isSAT
 8.         return autarky
 9.     autarky $\leftarrow$ autarky $\cup$ SatisfiedClauses(model)
10.     bound $\leftarrow |C| - |$autarky$| - 1$
```

Figure 3.4 Sifter finds the maximum autarky of a CNF formula C by "instrumenting" the instance and using a SAT solver to search for satisfying partial assignments.

We maximize the number of enabled clauses (y_i variables assigned TRUE) by way of an iterative optimization approach. We use AtMost constraints to bound the number of disabled clauses, tightening the bound as solutions are found. If an autarky is found that leaves n clauses disabled, we start the search for a larger autarky by bounding the disabled clauses to $n-1$. Eventually, if the instance is unsatisfiable, we will reach a bound k for which no solution can be found. At this point, we have proven that there exists an autarky of size $k-1$ and none with size k, thus the previously found autarky is the maximum autarky.

Figure 3.4 contains pseudocode for the complete algorithm, called Sifter. First, it repeatedly scans for pure literals, recording and removing them as described in Section 1.2.5: the call to **PureLits** returns 1) C with any clauses containing pure literals removed and 2) the set of such clauses as an initial autarky. The algorithm then instruments the formula and uses the sliding objective method described above to find the rest of the maximum autarky or to prove that the pure literal approach found it in its entirety. The **Instrument** subroutine produces instrumented clauses via the modifications described above. The bound on the number of disabled clauses is set initially to $|C|-1$ to begin the search by looking for an autarky that satisfies at least one clause, and the loop then proceeds by searching for a satisfying assignment, `model`, of the instrumented, bounded formula, C'_b. If none is found (`isSAT` is false), the algorithm returns `autarky`, which must be the maximum autarky. Otherwise, the satisfied clauses are added to `autarky`, the bound is set to search for an autarky that satisfies at least one more clause, and the loop repeats.

3.4.2 Performance

The two experimental goals were 1) to compare and contrast Sifter, our direct search-based approach for finding the maximum autarky, with the earlier iterative technique using resolution refutation trees (57), and 2) to investigate the value of trimming autarkies as a preprocessing step for finding MUSes and MCSes.

Comparing Search to an Iterated Resolution Proof Approach

Sifter was implemented in C++ using MiniSAT (31) version 1.12b (the last version containing support for AtMost constraints). We wrote the iterative approach (57), which we will call Scraper, as a Perl script. First, Scraper uses the pure literal elimination written for Sifter, making that phase equivalent in both implementations. Then, it employs the tools `zchaff` and `zverify_df` (87) from the ZChaff distribution `zchaff.64bit.2007.3.12` to repeatedly produce resolution refutations and eliminate the involved variables until the instance becomes satisfiable. All executables were compiled for the x86-64 instruction set using GCC 4.1.2 with standard optimizations, and all experiments were run under Linux (Fedora 7) on a 3.0GHz Intel Core 2 Duo E6850 with 4GB of RAM.

Figure 3.5 contains a log-log scatterplot comparing the runtimes of Sifter and Scraper on a variety of industrial benchmarks. Runtimes for Sifter are represented on the y-axis, so points lying below the diagonal indicate instances in which Sifter outperforms Scraper. A timeout of 600 seconds was used for every run, indicated by the dashed lines on the extremes of the chart; points on these lines indicate that a timeout was reached by the

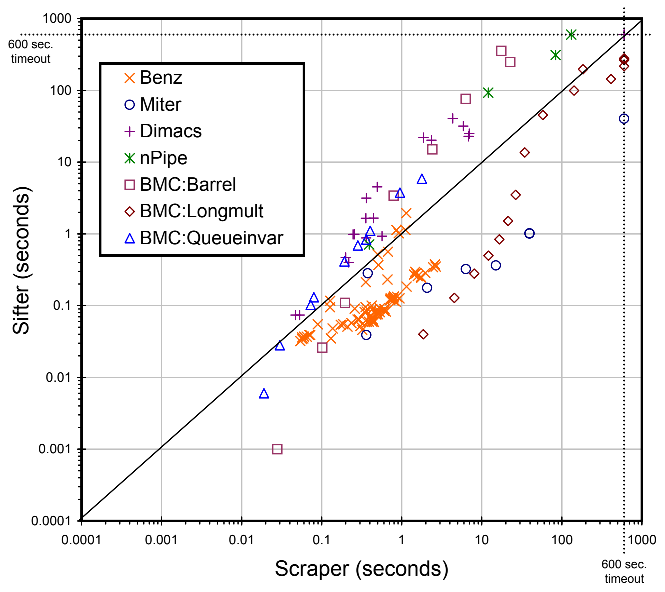

Figure 3.5 Comparing the performance of Sifter and Scraper on a variety of benchmarks

Family	Variables		Clauses		|autarky|	
	min	max	min	max	min	max
Benz	1,513	1,891	4,013	9,957	2,097	7,025
Miter	1,266	17,303	1,027	34,238	1	1,831
Dimacs	389	7,767	1,115	20,812	0	0
nPipe	861	15,469	6,695	394,739	0	0
BMC:Barrel	50	8,903	159	36,606	0	0
BMC:Longmult	437	7,807	1,206	24,351	2	2
BMC:Queueinvar	116	2,435	399	20,671	0	0

Table 3.3 Formula Sizes vs Autarky Sizes

corresponding algorithms. The reported runtimes are processor time, which for Sifter are essentially equivalent to wall-clock time. Our implementation of Scraper, however, stores several intermediate results to disk; we ignore this I/O time in these results to estimate the runtime of a more efficient approach that retains everything in memory.

To provide a more complete understanding of these results, Table 3.3 lists some overall characteristics of each benchmark family. The table lists the minimum and maximum number of variables, number of clauses, and size of the maximum autarky (in clauses) for the instances in each family. Appendix B contains further details and descriptions of these benchmark families.

From Figure 3.5 and Table 3.3, we can draw several conclusions:

1. Across all of the benchmarks, neither Sifter nor Scraper dominates the other in terms of runtime. In some benchmarks, Scraper is faster, up to 20x, while in others, Sifter is faster, up to 46x.
2. In just those benchmarks with non-trivial autarkies, however, the Sifter algorithm is faster in nearly every instance. Specifically, looking at the Benz and Miter families (the autarkies covering 2 clauses in each BMC:Longmult instance are all found by pure-literal elimination alone), we see that Sifter outperforms Scraper by approximately one order of magnitude.
3. The presence and size of autarkies is fairly consistent *within* benchmark families. Each particular family in Dimacs, nPipe, and BMC:[] has either no autarkies in any instance or an autarky that covers 2 clauses in each. The Benz family consistently has autarkies that cover a large portion (between 32 and 98 percent) of each instance's clauses. Every instance in the Miter family has a non-empty autarky, though the autarky sizes vary more than they do in the Benz instances.

Overall, these conclusions imply a strategy for exploiting autarkies in practice. First, by searching for autarkies on a small representative set of instances from a particular application, one can determine whether the instances in that domain have autarkies at all. If none of the test set have autarkies of any appreciable size, then it is likely that none generated in the application will, in which case autarkies will be of no use. This is likely in applications such as bounded model checking, where performing a cone of influence reduction of the circuit will likely eliminate all autarkies. In these applications, checking for autarkies could be a simple test of the sanity of the CNF encoding. In the other case, in which instances do contain autarkies, it is probable that most if not all instances will have autarkies, and Sifter is likely the more efficient algorithm to use.

Trimming Autarkies to Boost Searching for MUSes and MCSes

Trimming autarkies holds the most promise for boosting algorithms that have a high complexity and are affected heavily by the number of clauses in an instance. An algorithm for finding any single unsatisfiable subformula, such as that developed in ZChaff (87), is unlikely to benefit from such boosting, as the time taken to find the maximum autarky will likely dwarf the runtime of the unboosted algorithm.

We identified two algorithms that *are* good candidates for this boosting. One is the first phase of CAMUS (Section 2.2), and the other is an algorithm developed by Mneimneh, et. al. (70; 61) for computing an SMUS directly, which we will refer to as SMUS. Both of these candidate algorithms use clause-selector variables (as used in Sifter and described in Section 3.4.1) and use a SAT solver to implicitly search through subsets of clauses. Therefore, both can benefit from the reduced search space produced by a reduction in the number of input clauses.

We investigated the impact of trimming autarkies on both of these algorithms for the Benz benchmarks (Appendix B.1), which have the largest autarkies, and the results are displayed in Figures 3.6 and 3.7. Each figure is a log-log scatterplot that charts the runtime of the specified algorithm alone on the x-axis against the runtime of the boosted version on the y-axis. The runtime reported for the boosted version is the sum of finding an instance's maximum autarky with Sifter and running the algorithm on the trimmed instance. A point below the diagonal indicates an instance for which the boosting produced a net decrease in runtime.

The results are mixed. In Figure 3.6, we see that the boosting does not produce markedly better or worse results overall for finding SMUSes with SMUS. While the runtimes for SMUS alone (not shown) do improve in nearly all cases when it is run on the trimmed instances, the runtime of Sifter outweighs this gain in many cases. There are two outliers: one in which SMUS's runtime improves by over two orders of magnitude when run on the trimmed instance, and another that takes less than 10 seconds on the untrimmed instance yet times out at 600 seconds on the trimmed version. These are artifacts of the susceptibility of combinatorial search algorithms like SMUS to variations in runtime due to minor ordering changes and similar effects.

The results for boosting the first phase of CAMUS, shown in Figure 3.7, show that the boosting does have value in some cases. For this algorithm, the runtime of Sifter can outweigh the decrease in runtime due to the boosting in cases with *small* runtimes (below 1 second in these benchmarks), but the boosted algorithm always outperforms the original

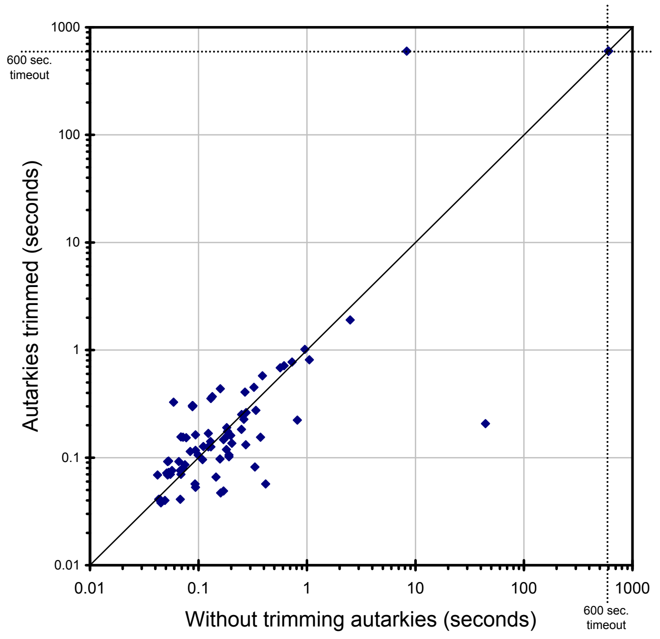

Figure 3.6 Boosting SMUS by trimming autarkies for the Benz benchmarks

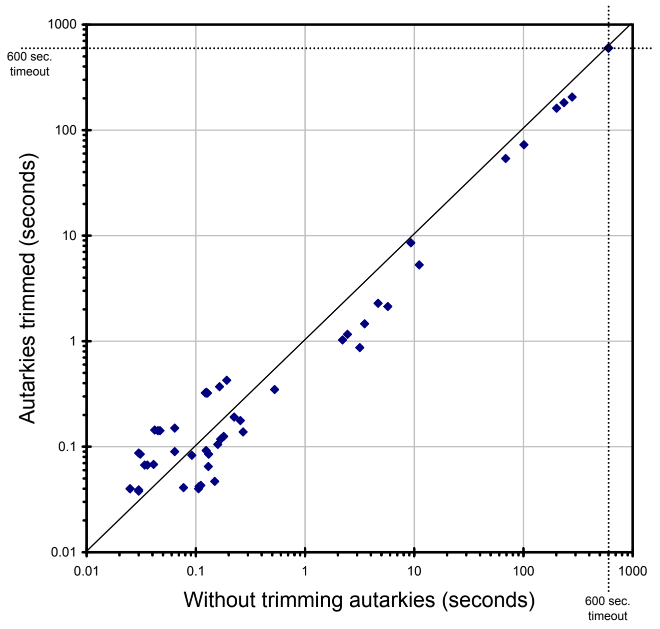

Figure 3.7 Boosting CAMUS (first phase) by trimming autarkies for the Benz benchmarks

algorithm in cases with longer runtimes. Taken as a whole, this is a net benefit, because the runtime increases in some "small" instances are far outweighed by the gains in the "large" instances. The total runtime, over all instances that did not time out in both techniques, decreased from 931 seconds on untrimmed instances to 704 seconds for the boosted algorithm, a 24% decrease in total runtime.

3.5 Exploiting Unsatisfiable Cores

In this section, we take one of the recent advances in analyzing infeasible instances, namely unsatisfiable-core-guided maximum satisfiability (core-guided Max-SAT), and generalize it to solve a related analysis with direct industrial applications: the identification of minimal correction sets (MCSes).

The concept of core-guided Max-SAT was first developed by Fu & Malik (36) and later enhanced and optimized by Marques-Silva and others (67; 68; 69); the algorithms and differences in their approaches are detailed in Section 1.2.4. The technique relies on and exploits one of the relationships between satisfiable and unsatisfiable subsets of infeasible systems that are discussed in Chapter 2. Briefly, an unsatisfiable instance will contain one or more *unsatisfiable cores*. No satisfiable subset of such an instance can contain any complete cores; therefore, any Max-SAT solution must must leave unsatisfied at least one clause from every core. The core-guided Max-SAT approach thus identifies unsatisfiable cores of an instance and only considers clauses within those cores as potential "removals," limiting the search space dramatically.

The use of unsatisfiable cores in solving Max-SAT yields drastically different performance than other current Max-SAT techniques, which are generally based on branch-and-bound. In the 2008 Max-SAT Evaluation (6), core-guided Max-SAT algorithms performed extremely well in the industrial Max-SAT category (one solving 72 of 112 instances within the timeout, when other approaches solved 0-3 and, in one case, 10 instances within the timeout), while performing among the bottom of the pack on random and crafted instances.

The industrial Max-SAT instances in the Max-SAT Evaluation are in fact produced by the circuit debugging system in (77), in which the desired result is actually MCSes of the CNF instances. In that work, the MCSes algorithm (Section 2.2) is used as a preprocessing step, identifying approximations of MCSes which are then used to boost a complete SAT-based search. This work, motivated by the success of core-guided Max-SAT on these instances, generalizes the core-guided Max-SAT approach to apply it to the problem of finding MCSes of CNF instances. A new algorithm, MCSes-U, is described and its correctness proven in Section 3.5.1, and we present experimental results showing its improvement over MCSes in Section 3.5.2.

3.5.1 Using Cores to Find MCSes

Our algorithm is a synthesis of 1) the MCSes algorithm for finding all MCSes of an infeasible constraint system and 2) the application of unsatisfiable cores to the Max-SAT problem as first shown by Fu & Malik (36) and refined by Marques-Silva, et al. (67; 68; 69). Because finding MCSes is a generalization of the Max-SAT problem (cf. Section 1.2.3), this combination is a natural one. In fact, the MCSes algorithm is very similar to the MSU3 algorithm described in (68).

Briefly, the overall approach of both MCSes and MSU3 is to instrument clauses in an unsatisfiable clause set with clause-selector variables, then to use cardinality constraints on those clause-selector variables to search for small subsets of clauses whose removal leaves the remaining set satisfiable. For Max-SAT, the goal is to find such a set of the smallest cardinality; finding MCSes requires finding all such sets that are minimal or irreducible. Therefore, it is reasonable to assume that an approach used to solve Max-SAT, especially one that has been paired with a basic algorithm so similar to that used for finding MCSes, could be applied to an algorithm for finding MCSes.

Figure 3.8 contains pseudocode for the algorithm, dubbed MCSes-U (the -U signifies its

```
MCSes-U($\varphi$)
1. $k \leftarrow 1$    ◁ iteration counter
2. MCSes $\leftarrow \emptyset$    ◁ growing set of results
3. $\texttt{Core}_k \leftarrow$ Core($\varphi$)    ◁ any unsatisfiable core (preferably small) of $\varphi$
4. while (InstrumentAll($\varphi$) + Blocking(MCSes)) is satisfiable
       ▽ clauses contained in $\texttt{Core}_k$ are instrumented with clause-selector variables
5.     $\varphi_k \leftarrow$ Instrument($\varphi$, $\texttt{Core}_k$) + AtMost($\texttt{Core}_k$, $k$)
       ▽ AllSAT finds all models of $\varphi_k$ corresponding to MCSes of size $k$
6.     MCSes $\leftarrow$ MCSes + AllSAT($\varphi_k$)
       ▽ the Core function projects instrumented clauses onto clauses of $\varphi$
7.     $\texttt{Core}_{k+1} \leftarrow \texttt{Core}_k$ + Core($\varphi_k$ + Blocking(MCSes))
8.     $k \leftarrow k + 1$
9. return MCSes
```

Figure 3.8 The MCSes-U algorithm finds all MCSes of an unsatisfiable formula φ using unsatisfiable cores.

use of **u**nsatisfiable cores). Two persistent variables, k and `MCSes`, keep track of the current iteration and the set of results, respectively. In any particular iteration of the **do/while** loop, $\texttt{Core}_k$ contains the set of clauses that will be considered for removal, and thus potentially included in an MCS, in that iteration. The input formula is instrumented with clause-selector variables on those clauses contained within $\texttt{Core}_k$, and an AtMost constraint is added on those selector variables with the current bound k. The **AllSAT** function in MCSes-U behaves exactly like the incremental solving employed in MCSes: find a solution, record the MCS, block that MCS from future solutions with a blocking clause formed from its clause-selector variables, and continue until no solutions remain.

The core extraction in line 7 produces an unsatisfiable core of the combination of the instrumented formula φ_k with the blocking clauses produced from the set of MCSes found thus far (φ_k itself is satisfiable). This core is mapped back to clauses in the original clause set φ and added to $\texttt{Core}_k$ to make $\texttt{Core}_{k+1}$ for the following iteration. The process repeats as long as further MCSes remain, which can be determined by checking whether there is *any* way to make φ satisfiable by removing clauses *without* removing any MCS identified thus far.

For comparison purposes, consider that the previous algorithm MCSes is equivalent to MCSes-U under the condition that **Core** always returns the complete set of clauses in φ. In this situation, the entire formula will be instrumented with clause-selector variables in each iteration, and the AtMost bound will always apply to all of the clause-selector variables as well. The primary difference between MCSes and MCSes-U is that here we are using unsatisfiable cores to identify subsets of the clause set in which we know the MCSes must be found, or, conversely, we determine subsets that we know must *not* contain any MCSes. The following section contains a proof of the completeness and correctness of this use of unsatisfiable cores.

Completeness/Correctness Proof

Fu and Malik proved that their use of unsatisfiable cores in Max-SAT is correct in (36); however, that proof does not carry over to our algorithm other than to prove that the first result returned will be a Max-SAT solution. We must further prove both 1) that every result returned by MCSes-U is an MCS (correctness) and 2) that all MCSes are found by the algorithm (completeness). These two points are interrelated:

Theorem 2. *Given an unsatisfiable clause set φ and a positive integer k:*

If all MCSes of φ of size less *than k are found, then every result of size k returned by MCSes-U(φ) is an MCS of φ.*

This theorem is stated without a formal proof, but it follows from the correctness of the underlying algorithm for finding MCSes, described fully in Section 2.2, that we have adapted in this work. Briefly, the algorithm finds MCSes in increasing order of size; as every MCS of a size less than k is found, it is blocked from future solutions, and any correction set of size k that is found then must be minimal. With this theorem, we see that the algorithm's correctness here hinges on its completeness. We will prove that MCSes-U is complete in the following.

To prove that the algorithm produces all MCSes of an instance, we will presuppose the completeness of the base algorithm as described in Section 2.2 and focus on the effect of the use of unsatisfiable cores. The base algorithm is equivalent to that presented in Figure 3.8 if we take $\texttt{Core}_k$ to be the complete formula φ in every iteration of the **while** loop (i.e., with no limitation on the clauses considered for finding MCSes). Therefore, we will prove here that the MCSes-U algorithm is complete in that it does not miss any MCSes due to restricting the search for MCSes to the clauses in $\texttt{Core}_k$.

First, we must define a useful term, "k-correction," and prove a useful lemma linking k-corrections to MCSes.

Definition 12. A k-correction of a set of clauses C is a set of k or fewer clauses whose removal makes C satisfiable.

Lemma 1. *Given an unsatisfiable subset C of a clause set φ and an integer k:*

If every $(k-1)$-correction of C contains some MCS of φ, then C contains all MCSes of φ with size k.

Proof. By contradiction: Assume that there exists some MCS M of φ with size k that is *not* contained entirely within C. We will denote the subset of M contained within C by $M' = M \cap C$. Thus, the assumption requires $|M'| \leq k-1$.

Because M is an MCS of φ and C is a subset of φ, M' must be a correction set of C. Formally, if $\varphi - M$ is satisfiable, then $C \cap (\varphi - M)$ must be as well. This can be transformed:

$$\begin{aligned} C \cap (\varphi - M) &= (\varphi \cap C) - (M \cap C) \\ &= C - M' \end{aligned}$$

And so M' is a correction set of C, because $C - M'$ is satisfiable.

Furthermore, M' is a $(k-1)$-correction of C, because $|M'| \leq k-1$. By the antecedent of this lemma, we know that M' must contain some MCS of φ. Because M is a proper superset of M', which contains an MCS, M cannot be a *minimal* correction set of φ. This is a contradiction, and therefore we have proven that any MCS M of φ with size k must be contained entirely within C.

$\square$

With this lemma, we can prove the completeness of our algorithm by induction. We wish to prove that the MCSes-U algorithm finds all MCSes of size k in the k^{th} iteration of its loop. We will first prove by induction that every $(k-1)$-correction of $\texttt{Core}_k$ contains an MCS of φ. Then, using Lemma 1, we can directly show that $\texttt{Core}_k$ contains all MCSes of size k, for all k. First, we will prove the base case of the inductive portion of the proof, for $k = 1$.

Lemma 2. *In the MCSes-U algorithm, every 0-correction*[3] *of* $\texttt{Core}_1$ *contains an MCS of* φ.

Proof. $\texttt{Core}_1$ is an unsatisfiable clause set; therefore, $\texttt{Core}_1$ has no 0-corrections, and the lemma is trivially true.

$\square$

With Lemmas 1 and 2, we see that the algorithm is complete for $k = 1$. $\texttt{Core}_1$ contains all single-clause MCSes of φ, and the algorithm produces all MCSes of size 1. This can be seen from a different perspective by noting that an MCS of size 1 is a single clause, c, contained in every MUS of a formula, and thus $\texttt{Core}_k$, which is some unsatisfiable core of φ, must contain every MCS of size 1.

With the base case proven in Lemma 2, we now prove the inductive step.

Lemma 3. *Given some positive integer k:*

In the MCSes-U algorithm, if every $(k-1)$*-correction of* $\texttt{Core}_k$ *contains an MCS of* φ*, then every k-correction of* $\texttt{Core}_{k+1}$ *contains an MCS of* φ.

[3]Following the definition, a 0-correction must be the empty set. Unsatisfiable clause sets have no 0-corrections, as removing 0 clauses cannot make them satisfiable.

Proof. Proof by cases, depending on the k-corrections of $\texttt{Core}_k$:

This proof relates to the following lines from the MCSes-U algorithm:

```
   ▽ clauses contained in $\texttt{Core}_k$ are instrumented with clause-selector variables
5. $\varphi_k \leftarrow \textbf{Instrument}(\varphi, \texttt{Core}_k) + \textbf{AtMost}(\texttt{Core}_k, k)$
6. ...
   ▽ the Core function projects instrumented clauses onto clauses of $\varphi$
7. $\texttt{Core}_{k+1} \leftarrow \texttt{Core}_k + \textbf{Core}(\varphi_k + \textbf{Blocking}(\texttt{MCSes}))$
```

Case 1: $\texttt{Core}_k$ has no k-corrections.

The algorithm includes $\texttt{Core}_k$ in $\texttt{Core}_{k+1}$. Therefore, in this case, $\texttt{Core}_{k+1}$ will have no k-corrections, as it is a superset of $\texttt{Core}_k$. Thus, trivially, every k-correction of $\texttt{Core}_{k+1}$ contains an MCS of φ.

Case 2: Every k-correction of $\texttt{Core}_k$ contains an MCS of φ.

Again, due to the fact that $\texttt{Core}_k \subseteq \texttt{Core}_{k+1}$, every k-correction of $\texttt{Core}_{k+1}$ is also a k-correction of $\texttt{Core}_k$, and thus every k-correction of $\texttt{Core}_{k+1}$ must contain some MCS of φ.

Case 3: At least one k-correction, δ, of $\texttt{Core}_k$ contains no MCSes of φ.

Because δ does not contain any MCSes of φ, the blocking clauses added to φ_k based on the MCSes of φ will all allow the relaxation of the clauses in δ. We will say that δ is thus an *unblocked* k-correction. When going into line 6 of the algorithm, there exists at least one complete assignment for φ_k that relaxes all MUSes contained within $\texttt{Core}_k$ without violating the AtMost bound on relaxed constraints. Namely, the clauses in any unblocked k-correction can be relaxed.

However, φ_k is unsatisfiable at this point, after the addition of all blocking clauses for the MCSes found thus far (up to size k). Therefore, for any complete assignment that satisfies the blocking clauses and relaxes all MUSes contained in $\texttt{Core}_k$, there must be some MUS of φ that is not relaxed by that assignment. Any unsatisfiable core of φ_k will necessarily include one MUS of φ that is not relaxed for every such assignment. That is, any unblocked k-correction δ of $\texttt{Core}_k$ must be "counteracted" by including in $\texttt{Core}_{k+1}$ an MUS of φ untouched by δ.

Any k-correction of $\texttt{Core}_{k+1}$ must contain a k-correction of $\texttt{Core}_k$, because $\texttt{Core}_k \subseteq \texttt{Core}_{k+1}$. Any unblocked k-correction of $\texttt{Core}_k$ necessarily leaves at least one MUS in $\texttt{Core}_{k+1}$ untouched (by the construction of $\texttt{Core}_{k+1}$ in the paragraph above). Thus, unblocked k-corrections of $\texttt{Core}_k$ cannot be k-corrections of $\texttt{Core}_{k+1}$. This leaves only "blocked" k-corrections, which all contain at least one MCS of φ.

Therefore, every k-correction of $\texttt{Core}_{k+1}$ must contain an MCS of φ.

These cases cover all possibilities, and, in every case, every k-correction of $\texttt{Core}_{k+1}$ contains an MCS of φ.

□

Now, we can finally prove the completeness of MCSes-U.

Theorem 3. *For any positive integer k: the* MCSes-U *algorithm finds all MCSes of size k in the* k^{th} *iteration of its loop.*

Proof. By Lemmas 2 and 3, we have that every $(k-1)$-correction of $\texttt{Core}_k$ contains an MCS of φ, for all k. With Lemma 1, then, $\texttt{Core}_k$ contains every MCS of φ of size k for all k.

□

Theorem 3 proves that MCSes-U is complete, and, with Theorem 2, this proves that it is correct as well.

3.5.2 Performance

The primary experimental goal was to determine the value of using unsatisfiable cores to guide the search for MCSes in practice; specifically, we wished to compare the performance of MCSes and MCSes-U on industrial instances. In the course of running these experiments, we noticed an interesting situation in which using cores was in fact detrimental to the performance of Max-SAT algorithms but the MCSes-U algorithm still benefited, and we explore this case here as well.

Experimental Setup: All experiments were run in Linux (Fedora 9) on a 3.0GHz Intel Core 2 Duo E6850 with 3GB of physical RAM. The MCSes and MCSes-U algorithms were implemented in C++ using MiniSAT version 1.12b (31), which allows "native" AtMost constraints (instead of CNF encodings thereof). We added unsatisfiable core extraction to this version of MiniSAT using the resolution-graph method (87), storing the parents of each learned clause in memory. Binaries for MSU1.1 and MSU1.2 were supplied by João Marques-Silva.

Benchmark Families: We selected four sets of unsatisfiable industrial CNF benchmarks for these experiments, described in detail in Appendix B: "Diagnosis," "Reveal," "FVP-UNSAT.2.0," and "DC."

The value of using cores is evident when we look at the results for finding multiple MCSes across all of these instances. Because the complete set of MCSes can be intractably large, we look at the *velocity* of finding MCSes: the number of MCSes found per second until all have been found or until a set timeout (600 seconds, here) has been reached. Many applications do not require the complete set of MCSes: the diagnosis task in (77) finds MCSes up to a certain cardinality, and the application of CAMUS in Reveal can use a subset of the MCSes to find a subset of the MUSes of an instance (cf. Section 3.1). Figure 3.9 compares the velocity of MCSes (w/o cores) to that of MCSes-U (w/ cores) on these

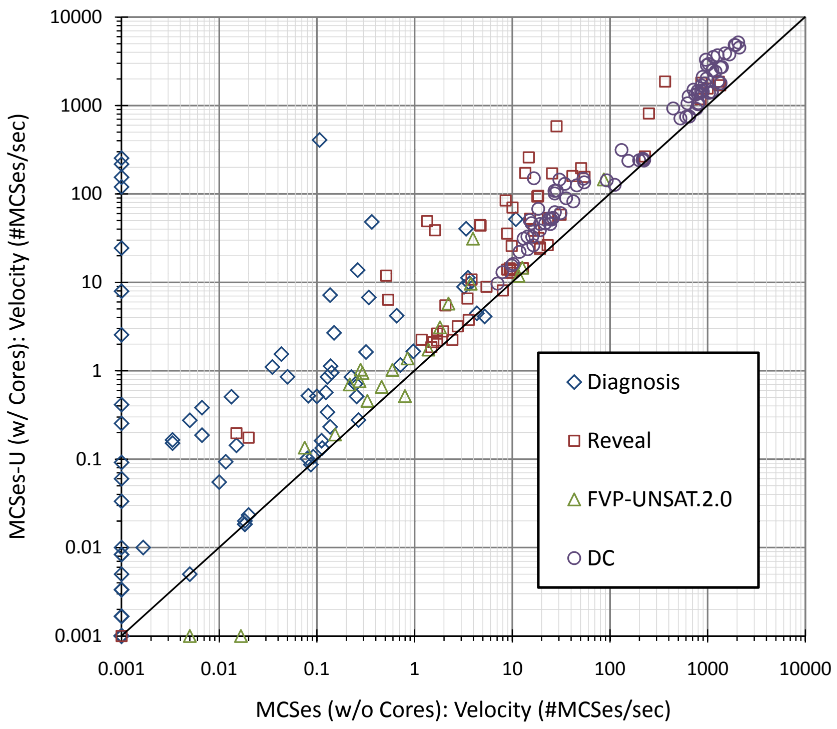

Figure 3.9 Comparing the performance of MCSes and MCSes-U on industrial benchmarks. (600 second timeout, 0 velocity mapped to 0.001.)

instances. Points above the diagonal are instances where MCSes-U finds MCSes more quickly. MCSes-U outperforms MCSes in nearly all cases. With the Diagnosis instances in particular, we see several benchmarks for which MCSes finds no MCSes within the timeout, while MCSes-U outputs up to several hundred per second.

An interesting situation is displayed in Figure 3.10, which compares the runtime of the MCSes algorithm solving Max-SAT (stopping after the first MCS is found) against three Max-SAT algorithms that use unsatisfiable cores: MCSes-U in the same Max-SAT mode, MSU1.1, and MSU1.2. This set of results is for the FVP-UNSAT.2.0 benchmarks. For these instances, we see that all of the algorithms that use cores take about two orders of magnitude longer than the vanilla MCSes algorithm. The time taken to identify an unsatisfiable core far outweighs the time needed to find a single minimum correction set (Max-SAT solution) in these instances. In these instances, the number of Max-SAT solutions is fairly large. For example, for each of the three 2pipe* instances in this set, approximately

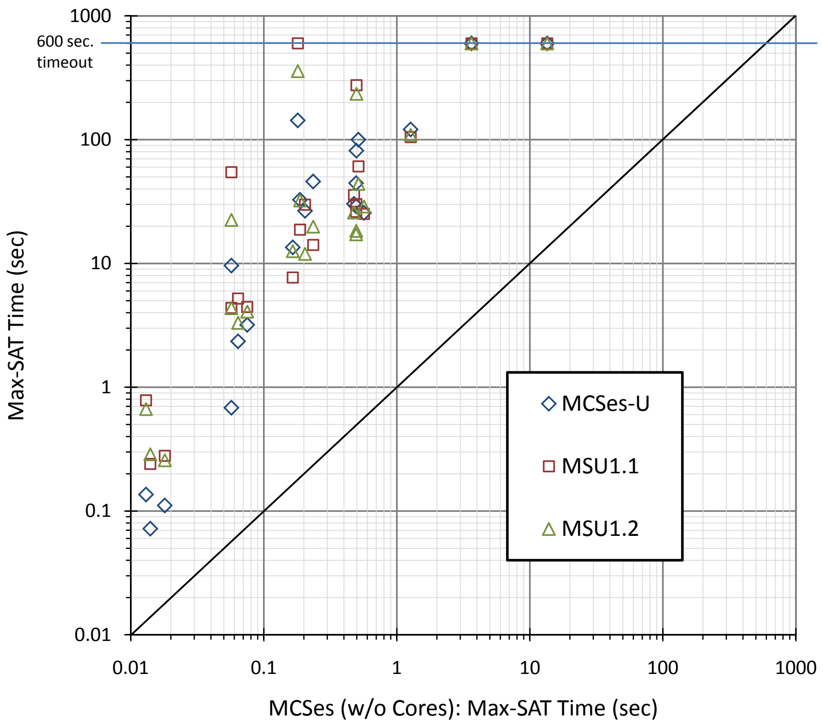

Figure 3.10 Comparing the performance of MCSes solving Max-SAT against MCSes-U MSU1.1, and MSU1.2 on industrial benchmarks. (FVP-UNSAT.2.0 benchmarks.)

one quarter of the clauses are single-clause MCSes; removing any one of them makes the instance satisfiable. Therefore, solving Max-SAT for these instances is fairly simple, as there are so many solutions, and they will be found in the first iteration of MCSes. In these instances, the time taken to solve the plain instance (in order to extract a core) far outweighs that taken to identify a single clause whose removal yields satisfiability.

It would seem that extracting cores is most useful for Max-SAT in instances where the cores are small (to provide the most benefit in limiting the search space) *and* the number of Max-SAT solutions is small. Otherwise, as we have seen, the time to find a core may outweigh that needed to find a solution, even with no core guidance. However, when finding MCSes, the overhead of finding cores appears to be amortized over the large number of results and outweighed itself by the increase in velocity gained from limiting the search space. Therefore, core extraction appears to be a safe addition to MCS algorithms in most industrial instances, which tend to have both small MUSes and small MCSes, relative to their formula sizes.

Looking forward, there are further ideas from the Max-SAT domain that can be applied to MCS algorithms. Notably, adding a single AtMost constraint per identified core, as done in the MSU1.* algorithms, may be applicable to MCSes-U. For Max-SAT, the MSU1.* approach has shown better performance than the approach used in MSU3 and MSU4 of creating a single monolithic AtMost constraint over all extracted cores, and it may be beneficial for MCSes-U as well. As with the proofs in this section, determining and proving the correct application of the concept to the generalized problem of finding MCSes may require non-trivial work. There is also potential in investigating the combination and interplay of the core-guidance technique with autarky pruning, another method for reducing the search space of the MCS search.

Further, the results here motivate applying MCSes-U in circuit debugging / diagnosis, as MCSes was applied in (77). While MCSes was used as an approximating preprocessor for an exact search in that work, the improved performance of MCSes-U may make it suitable for solving problems directly. A comparison to the algorithm in (81) could be instructive as well; though it is algorithmically very similar to MCSes-U, any substantial performance differences would indicate important implementation details that would aid in engineering future implementations. Further, (81) is restricted to only find minimum-cardinality solutions, and the more complete view of examining the set of *all* MCSes in such instances, which MCSes-U enables, could be beneficial.

3.6 Exploiting Symmetry

We investigated exploiting symmetries in CAMUS in two distinct ways. First, we looked at using symmetries to accelerate the search for MCSes. Secondly, we explored how symmetries can, in some cases, be used to provide concise encodings or descriptions of the sets of MCSes and MUSes, which can be exponentially large in the size of the original constraint system.

"Symmetry" here refers to a structural symmetry, an automorphism or a structure-preserving permutation of "pieces" of some "object." We can speak of symmetries of a constraint system as permutations of constraints, variables, or values that produce the same system. The precise definition of this depends on the type of constraint system. For Boolean CNF, for example, researchers have studied symmetries in terms of mapping clauses to clauses and literals (variables or their negations) to other literals with the added requirement that a literal and its negation must both map to literals of the same variable (note that $\{\{x \rightarrow \neg y\}, \{\neg x \rightarrow y\}\}$ is a valid permutation). Symmetries of Boolean CNF formulas have been used to speed SAT search, especially for unsatisfiable instances, by breaking symmetries in the formula, which lets a solver avoid searching redundant branches of the search tree (2).

Example 4.

$$\varphi = (a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee c) \wedge (\neg c)$$

This formula has a single non-trivial symmetry that permutes b with $\neg b$ and the first clause with the second. From this symmetry, we know that any assignment with b assigned TRUE is equivalent to the same assignment with b assigned FALSE; therefore, we can avoid searching redundant assignments by adding a new clause, (b), to restrict the space to just those assignments with b assigned TRUE. This effectively cuts the size of the search space in half.

Symmetries can be applied to MCSes and MUSes as well. Formally, in the language of group theory, we have a *group action* where clause symmetries form a group that can act on the set of MCSes (equivalently MUSes throughout this paragraph) such that applying a symmetry to an MCS produces another MCS. Given this group action, an *orbit* of a particular MCS is the set of MCSes produced by applying every symmetry to it. Every MCS thus appears in exactly one orbit, and the orbits form an *orbit partition* of the collection of

MCSes. Given the symmetry group and one representative MCS from every orbit, all other MCSes can be created by applying the symmetries to the orbit representatives.

3.6.1 Boosting MCSes

We investigated exploiting symmetries to boost MCSes, the first phase of CAMUS, because, in practice, MCSes is the main performance bottleneck of CAMUS. Using symmetries to boost the search for MCSes can be approached in several ways. In general, applying symmetry to MCSes computed by search can produce new, as-yet-unfound MCSes. The process of applying symmetries is computationally inexpensive compared to search, and we have applied it in order to reduce the time spent in those expensive tasks.

First, in any case, we must find the relevant symmetries. Using a tool like Saucy (23), a state-of-the-art tool for extracting symmetry information from graphs, we can find the symmetries of a CNF formula by converting it to a graph such that symmetries of the graph correspond to symmetries of the formula. Projecting these symmetries onto the clauses produces symmetries of the clauses alone (each valid given some permutation of the literals that is unimportant to this application). Now, given an MCS of those clauses, applying any of these symmetries will produce another valid MCS.

One simple way to employ these symmetries is during the search for MCSes. Whenever an MCS is found, immediately generate its orbit under the symmetries (all symmetric MCSes). Add blocking clauses for the generated MCSes to prune them from the search tree preemptively. Another idea is to add symmetry breaking predicates (SBPs) to the formula as in (2) and apply the symmetries after the search completes. We investigated both of these options.

MCSes-Symm(φ)

```
⋆1.  Generators ← GetSymm($\varphi$)
 2.  $k \leftarrow 1$          ◁ iteration counter
 3.  MCSes ← $\emptyset$          ◁ growing set of results
 4.  while (InstrumentAll($\varphi$) + Blocking(MCSes)) is satisfiable
 5.      $\varphi_k \leftarrow$ InstrumentAll($\varphi$) + AtMost($\{y_1 \ldots y_n\}, k$)
         ▽ find all models of $\varphi_k$ corresponding to MCSes of size $k$
 6.      while ($\varphi_k$ + Blocking(MCSes)) is satisfiable
 7.          newMCS ← ModelToMCS($\varphi_k$+)
 8.          MCSes ← MCSes + newMCS
             ▽ generate all symmetric MCSes
⋆9.          MCSes ← MCSes + ApplyGenerators(Generators, newMCS)
10.      $k \leftarrow k + 1$
11.  return MCSes
```

Figure 3.11 The MCSes-Symm algorithm finds all MCSes of an unsatisfiable formula φ using symmetries. [A $\star$ indicates a difference from MCSes.]

Applying Symmetries During Search

Figure 3.11 contains pseudocode for the MCSes-Symm algorithm, which implements the idea of generating results by applying symmetries to each MCS as it is found. The changes from the base MCSes algorithm are minimal, and they are indicated by $\star$ symbols. The symmetry generators are found at the beginning of the algorithm, and they are applied to every MCS found by the standard search, adding the new MCSes thus generated to the results set immediately.

Clearly, any performance gains of this modification will depend on an instance having significant applicable symmetries. However, the technique has negligible overhead in any situation where the symmetries are not applicable. Saucy scales extremely well, taking far less time to find the symmetries of an instance than the search needed for finding MCSes. Furthermore, an efficient implementation of the **ApplyGenerators** function can use hash tables to waste a minimal amount of time looking for generators applicable to any given MCS.

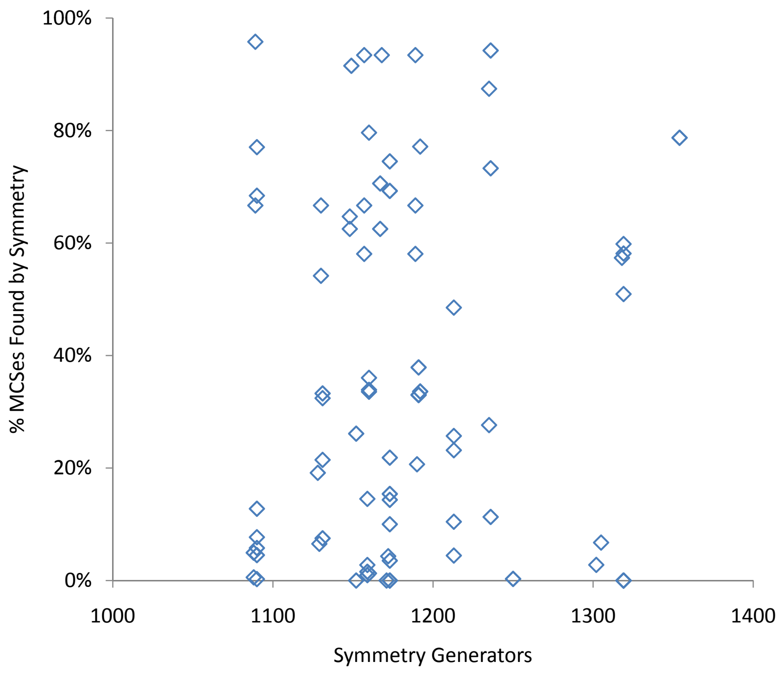

Figure 3.12 Percentage of MCSes found by applying symmetry generators vs number of symmetry generators (DC benchmarks)

Figure 3.12 shows the percentage of MCSes found by applying symmetry generators (line 9 of MCSes-Symm) versus the number of symmetry generators for the DC benchmarks (Appendix B.1). First, we can see that, in these instances, the number of MCSes found by applying symmetry information ranges from 0% to nearly all, with a fairly even distribution of percentages. Furthermore, we see that the number of symmetry generators does not have an effect on the percentage of symmetric MCSes. Similarly, the Reveal family of benchmarks (Appendix B.2) contains instances with tens of thousands of symmetry generators yet no symmetric MCSes whatsoever. The symmetries of an instance's clauses do not necessarily apply to its MCSes.

Though the high percentages of MCSes that can be generated from symmetries is encouraging, the actual performance gains are mixed. Figure 3.13 shows the MCS velocities for the DC benchmarks both with (y-axis) and without using symmetries (x-axis). The diamonds indicate a velocity on the y-axis that was calculated with the runtime of the MCSes algorithm alone, while the values marked by the red squares include the time taken for finding

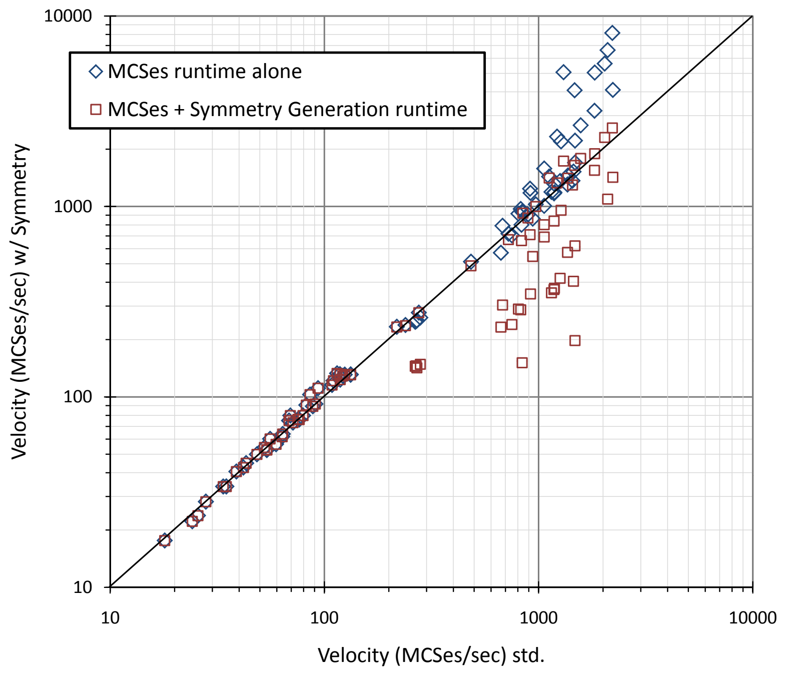

Figure 3.13 Velocities (MCSes per second) when using symmetry generators (y-axis) vs without (x-axis), 1) counting the runtime of MCSes alone in the velocity calculation and 2) including the runtime of the symmetry detection as well (DC benchmarks)

the symmetry generators in that calculation as well. Thought the MCSes algorithm itself sees either no change or reasonable performance gains from using symmetries, including the time taken to find symmetries often overwhelms those gains. It should be noted that this only occurs in those cases where the MCSes are found very quickly, as the time spent computing symmetries is consistently very short, but the gains from *using* symmetries were mainly in those instances with few MCSes found quickly as well.

Symmetry Breaking Predicates

Symmetry breaking predicates (SBPs) work well when solving satisfiability problem (2), but we have found that they are not directly applicable to finding MCSes. The idea is that SBPs are added to a formula in the form of additional constraints that permit only one solution out of a given set of symmetric solutions and block the remainder. Each SBP thus cuts out a portion of the search space that is symmetric to some other portion; this does not change whether an instance is satisfiable or not, so it can be used directly for solving satisfiability problems.

At first glance, it appears to be a reasonable approach to finding MCSes as well. Blocking symmetric portions of the search space should speed up search and reduce the number of results returned; blocked results can be generated by applying symmetries, after search, to those MCSes that were found. However, consider the following example.

Example 5. In this example, adding a symmetry breaking predicate to the formula prevents a symmetric MCS from being found, but it does not prevent a superset of that MCS from being a valid solution. This results in a spurious result (and in general can result in many spurious solutions).

$$\varphi = \quad 1.(a) \quad 2.(\neg a) \quad 3.(b) \quad 4.(\neg b) \quad 5.(a \vee b)$$

Clauses are numbered for easy reference, and we will refer to them by their number alone. The MCSes of this formula are:

$$\{\{1,4\},\{2,4\},\{2,3\},\{1,3,5\}\}$$

The only clause-symmetry of this formula maps clause 1 to 3 and 2 to 4 simultaneously. Following the construction in (2), a symmetry breaking predicate for this symmetry is

$$(y_1 > y_3) \wedge (y_1 = y_3 \vee y_2 \geq y_4)$$

where $>$ and $\geq$ are the numerical "greater than" and "greater than or equal to" operators applied to the $\{0,1\}$ values of the clause selector variables.
With this additional predicate, the MCSes algorithm will find these sets as MCSes:

$$\{\{1,4\},\{2,4\},\{\mathbf{1},2,3\},\{1,3,5\}\}$$

The SBP has blocked the solution $\{2,3\}$, because it is symmetrical to $\{1,4\}$. However, $\{1,2,3\}$: $(y_1 = 0, y_2 = 0, y_3 = 0)$ is not blocked by the SBP, and because we never found $\{2,3\}$, the algorithm never creates a blocking clause preventing $(y_2 = 0, y_3 = 0)$.

This example shows how SBPs *do* remove symmetric solutions from the search space, but this disrupts one of the invariants needed for the correctness of the MCSes algorithm: when finding MCSes of size k, all MCSes of size less than k must be blocked. This invariant prevents the search from finding non-minimal correction sets, because they will be subsumed by smaller MCSes that are blocked with added constraints. SBPs disrupt this invariant by skipping some solutions.

Furthermore, we cannot specially craft SBPs beforehand to avoid the subsumed, non-minimal correction sets as well, because we do not know which sets to avoid until we have found the MCSes that subsume them. In Example 5, we cannot know that the correction set $\{1,2,3\}$ should be blocked until we have discovered that $\{2,3\}$ is an MCS. In general, we are faced with the problem that we cannot break symmetries *statically*, before the search, because the decision to block certain solutions can only be made once earlier solutions (smaller MCSes) have been found. Some form of *dynamic* symmetry breaking, making decisions about pruning search subspaces during the search itself, may work. For example, GAPLex (75) could be used; it employs the GAP computational group theory system to perform symmetry breaking during search, dynamically pruning portions of a search tree by reasoning about the symmetry group of a problem at every node in the tree.

We can use the SBPs *if* we somehow block solutions symmetric to known MCSes as they are found. This becomes equivalent to adding static SBPs to MCSes-Symm, however, and the SBPs then serve little purpose other than to guide the search to specific solutions; the complete set of results will (and must) still be generated. We have run experiments that indicate that adding SBPs to MCSes-Symm does not aid performance, and it often slows the algorithm greatly. The SBPs add complexity and overhead to the constraint solving without providing any clear benefit.

3.6.2 Seeking Exponential Compression

Using symmetries to provide short, implicit encodings of the potentially exponential sets of MCSes or MUSes, holds the promise of tackling the intractability of computing exponentially large sets while still maintaining completeness, in a sense.

Example 6. Consider a scheduling application, in which meetings, or any other type of event, are assigned to rooms at certain times. Every event has constraints on these assignments, some restricting the locations of certain events, others constraining their times, and still others effecting certain combinations of locations and times, perhaps even linking multiple events. A particular instance in this application may have constraints that altogether end up producing a variant of the pigeonhole problem, in which m events must be assigned to n (where $n < m$) locations all at the same time.

If $n = m - 1$, then we have the standard pigeonhole problem, with one too many pigeons (events) for the holes (locations). This creates a single MUS, because removing any single constraint (that any particular location can have at most one event at a time or that any particular event must be assigned a location) eliminates the conflict. If $n \leq m - 2$, however, the instance is much more overconstrained and has a large number of MUSes. However, because all events are symmetric within the context of the pigeonhole problem and all locations are as well, the entire set can be represented much more simply.

For five locations ($n = 5$) and seven events ($m = 7$), the pigeonhole problem encoded in CNF has 112 constraints and 27,587 MUSes (found experimentally with CAMUS). However, these can all be generated from just 14 different MUSes by applying the symmetries among the constraints related to symmetric locations and symmetric events in order to generate the remaining 27,573 MUSes.

If a large collection of sets exhibits a great deal of symmetry, the entire collection can be represented by a small selection of its sets along with permutations that describe the collection's symmetry group. This representation is both compact, potentially an exponential compression of the entire collection, and intuitive. The following example illustrates this in a more formal manner.

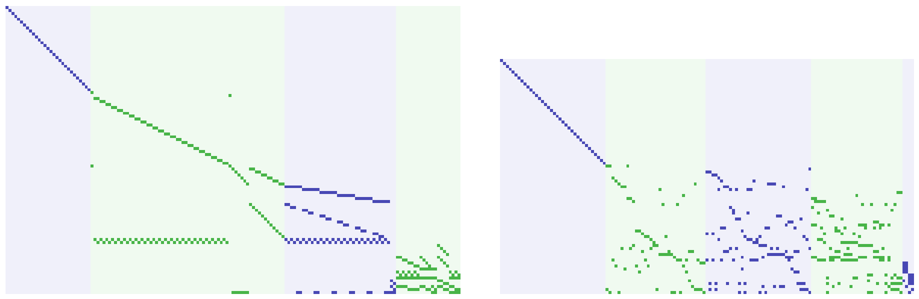

Figure 3.14 Visualizations of two sets of MCSes. The MCSes from an industrial benchmark (left) exhibit far more structure than those from a random 3-SAT instance (right). Both images were generated with the same algorithm, which uses ordering heuristics to draw out structure visually.

Example 7. A collection of sets of integers can be described as follows:

Every set contains one element of $\{1,2,3\}$, one element of $\{10,11,...,19\}$, and one element of $\{20,21,...29\}$.

This collection of 300 sets can be represented by a single set, $\{1,10,20\}$, and the following permutations: $\{1 \to 2 \to 3\}$, $\{10 \to 11 \to ... \to 19\}$, and $\{20 \to 21 \to ... \to 29\}$.

$\{1,10,20\}$ can be seen as an *orbit representative* of the entire collection of sets, which falls into a single orbit under the group action of the symmetry group defined by those permutations.

This "representatives plus permutations" representation of a collection is both tractable and human-readable, compared to a full enumeration of the collection. Formally, we are interested in computing orbit representatives for the collection of MCSes or MUSes and generators of their symmetry groups.

We hypothesized that many of the intractably large sets of MCSes and MUSes arising from real-world problems (as opposed to randomly-generated instances) contain symmetric structure sufficient to encode entire sets in exponentially smaller representations. We have done work on visualizing the sets of MCSes and MUSes of CNF formulas that supports this hypothesis; a great deal of structure is visible in industrial instances when compared to randomized formulas, as seen in the MCSes of such a pair of instances in Figure 3.14.

We investigated the prevalence of symmetry in the MCSes of real-world instances to determine the potential of using symmetries to combat the intractable MCS-set sizes. First, we explored the potential benefit. Using several instances for which the complete set of MCSes was available, we computed their orbits under the group actions of clause symmetries extracted from each instance's CNF formula. The following example illustrates this further, using the same formula as in Example 5.

Example 8.

$$\varphi = \quad 1.(a) \quad 2.(\neg a) \quad 3.(b) \quad 4.(\neg b) \quad 5.(a \vee b)$$

$$\mathsf{MCSes}(\varphi) : \{\{1,4\},\{2,4\},\{2,3\},\{1,3,5\}\}$$

The only clause-symmetry of this formula maps clause 1 to 3 and 2 to 4 simultaneously. This symmetry maps $\{1,4\}$ to $\{2,3\}$, thus these two MCSes are in the same orbit. The other two MCSes map to themselves under the symmetry.
Therefore, the set of MCSes can be represented by three orbit representatives and the symmetry:

$$\{\{1,4\},\{2,4\},\{1,3,5\}\} \qquad \{(1 \to 3)(2 \to 4)\}$$

The end goal would be an algorithm that computes the orbit representatives directly, avoiding enumerating all of the MCSes. Initially, looking at the number of MCS orbit representatives, even if generated from a complete enumeration, gives us an idea of how well such an algorithm could work.

Empirical Investigation

The chart in Figure 3.15 shows results for this experiment. Instances along the horizontal axis are sorted by their number of MCSes (a subset of the DC benchmarks whose MCSes can be calculated in under 600 seconds was used), and the top line shows this number for

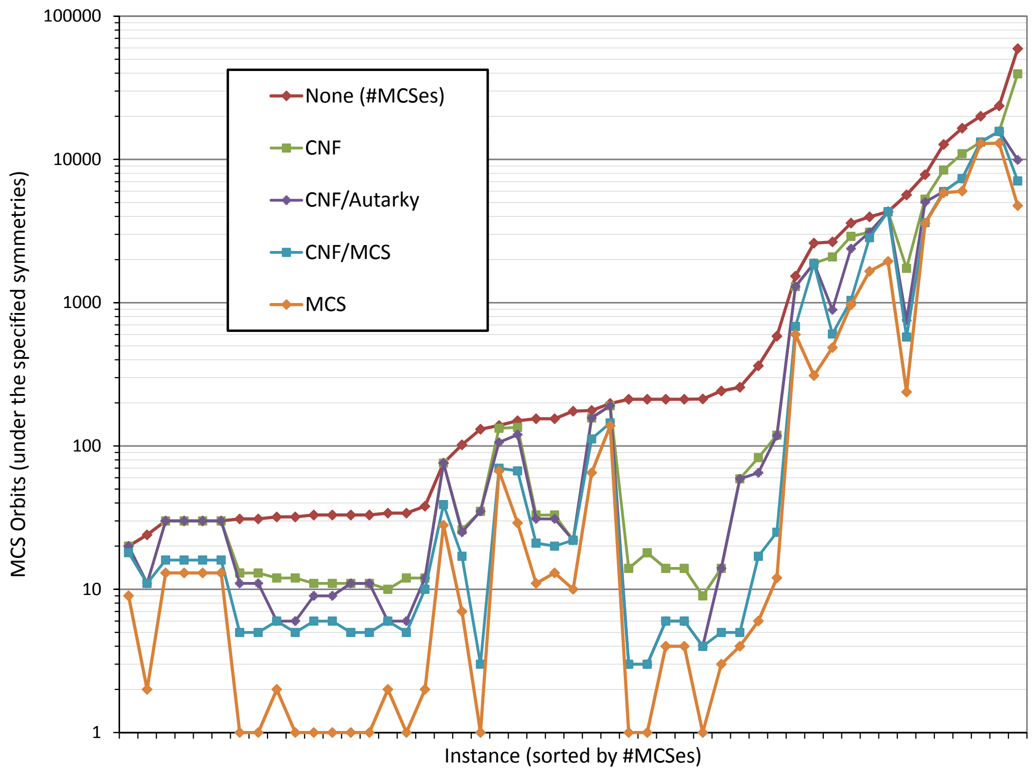

Figure 3.15 Number of MCS orbits for instances in the DC benchmark family, given: 1) no symmetries, 2) CNF symmetries, 3) CNF symmetries following autarky pruning, 4) CNF symmetries following pruning to MCS clauses, 5) MCS symmetries. (Subset of DC instances with all MCSes found in under 600 seconds.)

each. The second line down, labeled "CNF" in the legend, shows the number of MCS orbits computed, given the clause symmetries of the original CNF formula. While there are some cases in which the number of orbits is an order of magnitude lower than the number of MCSes, these cases are in the minority. Most instances range from no difference (the symmetries provide no "compression" at all) to differing by a factor of 2 or less.

The idea of *conditional symmetries* (similar to those described for CSPs in (38)) motivate looking further. Conditional symmetries are symmetries that emerge after some modification to the constraint system such as removing a constraint or propagating a partial assignment. Conditional symmetries may allow for further exploitation of problem structure in cases where the problem as given does not exhibit significant symmetry. Consider the following motivating example:

Example 9.

$$\varphi = (a \vee b) \wedge (\neg a \vee b) \wedge (\neg b) \wedge (a \vee c)$$

This formula has no symmetries. However, if the final clause, $(a \vee c)$, is removed, the first two clauses become symmetric to each other (with the permutation $\{a \rightarrow \neg a\}$). Symmetries of the original formula provide no benefit (there are none), but the symmetry of the first two clauses conditional on the removal of the last clause can be applied to an MCS such as $(a \vee b)$ to produce $(\neg a \vee b)$ without search, as the MCS does not contain $(a \vee c)$ and the conditional symmetry applies.

Following this idea, we looked at the number of MCS orbits generated by symmetries computed from the formula after removing extraneous clauses. The results of these experiments are also displayed in Figure 3.15. The third line from the top, labeled "CNF/Autarky," indicates the number of MCS orbits computed with symmetries found in the formula after pruning the maximum autarky (as described in Section 3.4), while the fourth line, labeled "CNF/MCS," shows the MCS orbits found with the symmetries of the formula after removing all clauses not in any MCS. Both show additional gains in reducing the number of representatives, with the the MCS clauses alone dominating the autarky pruning (as they should; the MCS clauses will always be a subset of the formula minus its maximum autarky).

Finally, we investigated the truly ideal case: generating MCS orbits under the symmetries of the MCSes themselves. To do this, we translated the MCSes into a graph structure with a node per clause, a node per MCS, and edges connecting clauses with the MCSes in which they occur. We again used Saucy to find the symmetries of the graph, which we mapped back to the clauses. This is the ideal case because it has shed the formula entirely, looking at the symmetry of the MCSes themselves, unencumbered by the structure of the original formula. The number of orbits found in this case are shown in the bottom line in Figure 3.15, labeled "MCS." Again, there are improvements over the other cases, though even in this scenario, the potential compression is often under a factor of 2, especially in the instances with larger sets of MCSes, for which the compression would be the most useful.

Observations

While they provide interesting information about the symmetries in these instances, the "MCS" case and the "CNF/MCS" case are unrealistic in practice. Each relies on having the symmetry group of the complete set of MCSes, which is of course unavailable until *after* a complete set of MCSes has been calculated. Any algorithm that makes practical use of symmetries will have to use only that symmetry information that can be gleaned from the formula without finding all MCSes first.

Given the definition of symmetry of an MCS that we are using, MCSes can only map to other MCSes of the same size. However, in a CNF formula, it is quite likely that groups of clauses with different sizes could have the same effect on the formula's infeasibility. For example $\{(a \vee b)\}$ could be swapped for $\{(a \vee c), (\neg c \vee b)\}$ in some MCSes, but this would never be captured given the symmetries we have defined which must have 1-to-1 permutations of clauses. A more general definition of symmetry that allows these more complex swaps could capture additional structure information and allow increased compression.

Those cases in Figure 3.15 that report 1 orbit for the "MCS" scenario all contain only single-clause MCSes (thus they must contain a single MUS), otherwise there would have to be further orbits to account for other MCS sizes. In fact, all single-clause MCSes will always be symmetric to one another in the "MCS" symmetry scenario, because they will never interact with other clauses in any other MCS.

Because the set of MCSes and MUSes are implicit encodings of each other (i.e., each MCS encodes "every MUS contains one of these," and each MUS encodes the same about the MCSes), either can in fact be an exponential compression of the other. One benchmark has 257 MCSes and exactly 2^{127} (about 10^{38}) MUSes, which can only be determined by considering the symmetries present in the MCSes. In cases like this, the MCSes are an exponential compression of the MUSes, and it may be that the MCSes themselves are as

Extension	Applicable To:
Relaxing Completeness	Any implementation
Constraint Grouping	Any implementation with selector variables
Smallest MUSes	Any implementation
Autarkies	Boolean SAT only
Exploiting Cores	Any implementation w/ an approximate core algorithm
Exploiting Symmetry	Any implementation

Table 3.4 Applicability of CAMUS extensions to constraint types beyond Boolean SAT

concise as any representation can be.

All of this work on using symmetries is complementary to that of O'Sullivan et al. (73), who developed an algorithm, based on Bailey and Stuckey's DAA (8), to generate a *Minimal Representative Set of Explanations*. In our terminology, they provide a minimal set of MCSes such that every constraint in any MCS is represented at least once and every constraint in the complement of any MCS is in the *complement* of at least one of the presented MCSes. While their algorithm does guarantee a linear number of explanations, in fact bounded by the number of constraints, it may still require enumerating all MCSes in the system, a potentially exponential set. This work is complementary in that it employs a different notion of "representative" (though it may be possible to combine the two concepts) and in that it aims to avoid the intractability of computing all MCSes.

3.7 Application to Other Constraint Types

As with the base algorithms of CAMUS, many of the extensions described in this chapter can be applied to constraint types beyond Boolean satisfiability as well. Table 3.4 outlines the potential for applying each of these extensions to constraint types beyond Boolean satisfiability.

The primary change to the basic algorithms in the extension of relaxing the completeness of CAMUS was to truncate MCSes as they are found to produce PCSes. The only extra interaction with the actual constraints is to remove constraints from the problem when they have been eliminated by the **Truncate** subroutine. This can be accomplished in exactly the same way blocking clauses are created, by adding new constraints on the selector variables for the removed constraints. Therefore, this relaxation can be accomplished with any implementation of CAMUS.

Constraint grouping is similarly wide in its applicability, but it is only immediately applicable to implementations that use selector variables. Grouping is accomplished for any constraint type by assigning the same selector variable to multiple constraints. Grouping can be achieved even in implementations without selector variables, like our implementation on top of YICES (cf. Section 2.6), however. In that implementation, constraints can be grouped by creating a single constraint that is a conjunction of the grouped constraints, as the SMT formalism supports logical AND combinations natively. This creates an object, treated by the solver as a single constraint, that accomplishes the same search space reduction as grouping constraints with selector variables.

The modification to CAMUS to find a smallest MUS of an instance is purely a modification to the second phase. Because the second phase is already independent of the type of constraints from which the MCSes were generated, this extension can be built into any implementation of CAMUS.

The extension of finding and pruning autarkies is the sole extension that is specific to Boolean Satisfiability. Autarkies have only been defined for SAT instances, and while similar definitions can be considered for other constraint types, the algorithm we developed for finding them will not translate easily.

Exploiting cores to boost the search for MCSes relies heavily on the ability to produce unsatisfiable cores of an instance *quickly*. In the case of Boolean SAT, this can be done by one of the many approximate-core resolution-proof (e.g., (72) and (87)). While similar algorithms exist for other constraint types (e.g., SMT (21)), developing such an algorithm for a new domain is not a simple task. Therefore, this extension is practically limited to those constraint types for which efficient core extractors exist.

Finally, symmetries exist in both constraint systems and their MCSes. Symmetries of MCSes can be found and used after the MCSes are found in any constraint type, because as usual the MCSes can be analyzed without any consideration for the underlying constraints. Symmetries of the constraints themselves can be more difficult to find, but the approach we used, of creating a graph representation of the constraints and finding graph automorphisms, should work for any constraint type. A specific graph representation will need to be developed for each type of constraint, and certain constraint types may naturally yield more symmetries than others, but the concepts are equally valid in any case.

Chapter 4
Applications and Conclusion

This chapter briefly outlines two applications in which, through collaborations with other researchers, CAMUS has successfully been applied to practical applications, and the dissertation concludes with a summary of the work and directions for future research. CAMUS has been used in two distinct logic circuit verification tasks that demonstrate its real-world potential. The first application, a system called Reveal (3), is a model-checking system developed to operate directly on high-level Verilog circuit descriptions (as opposed to lower level logic gate circuits). The second task is a circuit diagnosis system that employs the first phase of CAMUS, along with the constraint grouping technique (Section 3.2), to locate potential causes within a circuit of an observed erroneous output (77).

4.1 Reveal

In Reveal, MUSes are used as a tool for *generalization*; if one considers every constraint in an unsatisfiable system to be part of a description of the infeasibility, an MUS is a more general description than the entire set. Reveal verifies properties of circuits using counterexample-guided abstraction refinement (CEGAR) (59). In the CEGAR framework, inconsistent constraint systems are created that represent invalid solutions to a problem; each invalid solution is removed from the solution space by refining the problem using information about the inconsistency. Generalizing that information via an MUS eliminates a larger class of invalid solutions, leading to faster convergence of the entire process.

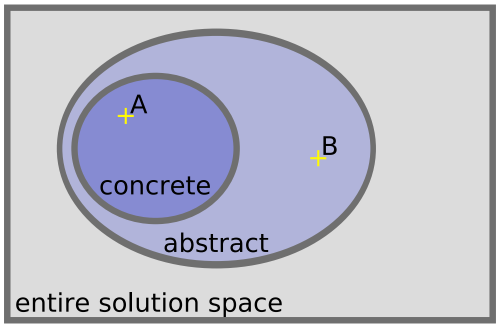

Figure 4.1 Two possible solutions for *Abs*: A lies within *Conc*, while B does not.

To avoid delving into the details of the circuit verification performed by REVEAL, this section will instead describe the process it uses in general terms to illustrate the application of CAMUS. In CEGAR, and abstraction refinement in general, a complex problem is tackled by first making an abstraction *Abs* of the problem that removes certain details. The abstraction can be much simpler than the fully-specified concrete problem *Conc*, as long as it has the property that any solution to *Conc* is also a solution to *Abs*; the other direction may not hold, however, as *Abs* is allowed to be an over-approximation of the problem with solutions that are not valid for *Conc*. The idea is essentially the same as “relaxation” methods used in operations research.

If *Abs* has no solution, then *Conc* cannot have one either, and the process is done. Any solution found for *Abs*, however, may be *spurious*, so it must be checked against *Conc*.

To check a solution to *Abs*, a constraint system is formed from the conjunction or intersection of *Conc* with the solution S: $Conc \wedge S$. If $Conc \wedge S$ is satisfiable, the solution is valid and the process exits (solution A in Figure 4.1). If $Conc \wedge S$ is unsatisfiable (solution B in Figure 4.1), however, the solution is spurious, and *Abs* must be *refined* to remove this invalid solution, after which the process repeats with an updated abstraction.

So *Abs* needs to be refined by removing the spurious solution S from its solution space. Any addition of constraints to *Abs* that conflict with S will suffice, but they also must not remove any real solutions. For example, asserting the negation of the counterexample is a valid option; S is not a solution to $Abs \wedge \neg(S)$, and the only solution removed is the spurious

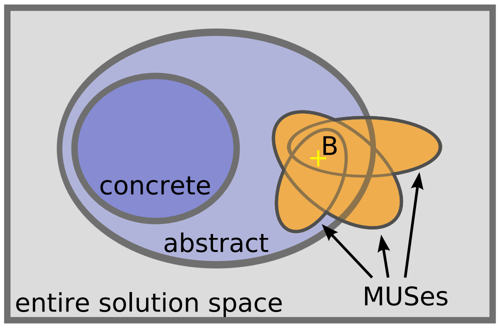

Figure 4.2 MUSes of *Conc* ∧ B, projected onto B, produce generalizations of B that cover more of the solution space.

solution itself. However, this will only eliminate that exact solution from the abstraction, and the next iteration of the process may find a very similar solution that differs only slightly.

By using MUSes, larger classes of solutions can be eliminated with fewer constraints added. An MUS of the system $Conc \wedge S$ projected onto the spurious solution (i.e., only considering the constraints in the MUS that came from S) is a generalization of S, a partially-specified spurious solution. By the minimal nature of the MUS, the specification is minimal (maximally general), and so it is an ideal candidate for updating the abstraction.

Furthermore, $Conc \wedge S$ may contain multiple MUSes, each indicating a distinct class of spurious solutions represented by that one counterexample. Refining with multiple MUSes does yield a larger model, as more constraints are added for each MUS used, but it eliminates more spurious behaviors at the same time. Experimental results have shown that refinement using multiple MUSes, as found by CAMUS, yields faster convergence of the refinement process both in terms of number of iterations and overall runtime (4). Generalization and refinement with multiple MUSes is illustrated in Figures 4.2 and 4.3.

Experimental results have shown that refinement using multiple MUSes, as found by CAMUS, yields faster convergence of the refinement process both in terms of number of iterations and overall runtime (4; 5). For example, in 9 out of 20 test cases, Reveal timed out, still working after reaching one- to two-thousand refinement iterations, when using a single MUS for refinement each time. Switching to using multiple MUSes per refinement

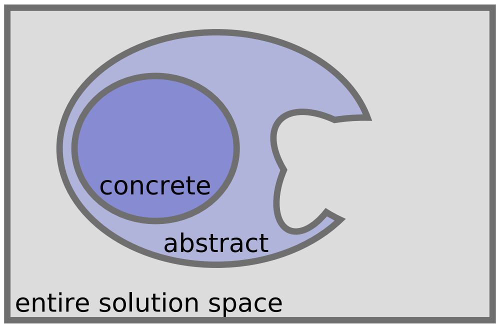

Figure 4.3 Refining the abstraction with all MUSes produces a maximal reduction in the solution space.

iteration reduced those all to between 7 and 93 iterations to completion, with corresponding runtime reductions. This shows the value of CAMUS over algorithms for finding a single MUS or US.

Initially, Reveal used the Boolean SAT implementation of CAMUS. We developed an implementation of CAMUS for SMT formulas using YICES (30) when Reveal was updated to use an SMT solver. This provided a large performance boost due to keeping everything at the higher level allowed by the more complex and expressive constraint types. The implementation was fairly straightforward, thanks to the generality of CAMUS and its lack of reliance on specific constraint solver abilities. YICES has a mode in which it can solve Max-SAT / Max-CSP, which, while not well-documented, appears to follow the same sliding objective approach that the MCSes algorithm uses. Using this mode and YICES' ability to apply weights to constraints (allowing for hardening previously soft constraints), the first phase of CAMUS was implemented with minimal trouble.

The second phase of CAMUS, which operates on MCSes, is entirely independent of the constraints from which the MCSes were generated. Every MCS is simply a set of values that do serve as constraint indices, but which are treated as nothing more than numbers by the hitting-set algorithm. Therefore, no changes whatsoever were required to the second phase of CAMUS for the port.

4.2 Circuit Error Diagnosis / Debugging

In this work, with Safarpour et al. (77), the goal was the same as that in the model-based diagnosis work begun in the 1980's: given a circuit producing the wrong output, identify minimal sets of locations in the circuit where errors could produce that output. Our main insight in this work was that the error diagnoses we sought to compute were in fact MCSes of the constraint system formed by joining the circuit model with the observed incorrect outputs. While experiments showed that computing the MCSes directly with the first phase of CAMUS was not efficient, we were able to use the ability of CAMUS to group constraints to reduce the complexity and find approximate solutions.

As described in Section 3.2, CAMUS can treat any subset of constraints as a single, higher-level constraint, and MCSes and MUSes can be found in terms of these high-level constraints. For example, by treating groups of 10 clauses each as single constraints, the number of constraints is reduced by a factor of 10 and the search space for CAMUS is reduced by an exponential factor.

The MCSes returned by CAMUS when this sort of grouping is used are approximations, as each group in a "high-level" MCS may bring in clauses that are not part of an MCS of the lower-level clauses. Thus, we used these as seeds to Safarpour, et al.'s existing technique for circuit diagnosis, called debug, which could find the exact diagnoses within the high-level MCSes. The debug algorithm works by instrumenting each gate in the circuit model in order to permit it to be disabled, and it uses a CNF encoding of the instrumented circuit to find diagnoses; the seeds provided by CAMUS allow debug to limit the gates that it instruments to those indicated by the approximate MCS. In this way, CAMUS was essentially used to boost the existing debug algorithm.

The runtime depended on the size of the constraint groups created. If the groups were too small, the runtime of the boosting phase that found all MCSes of the grouped constraints dominated that taken by debug. On the other hand, if the groups are too large, the approximation is so great that the results of the first phase provide little benefit to the debug

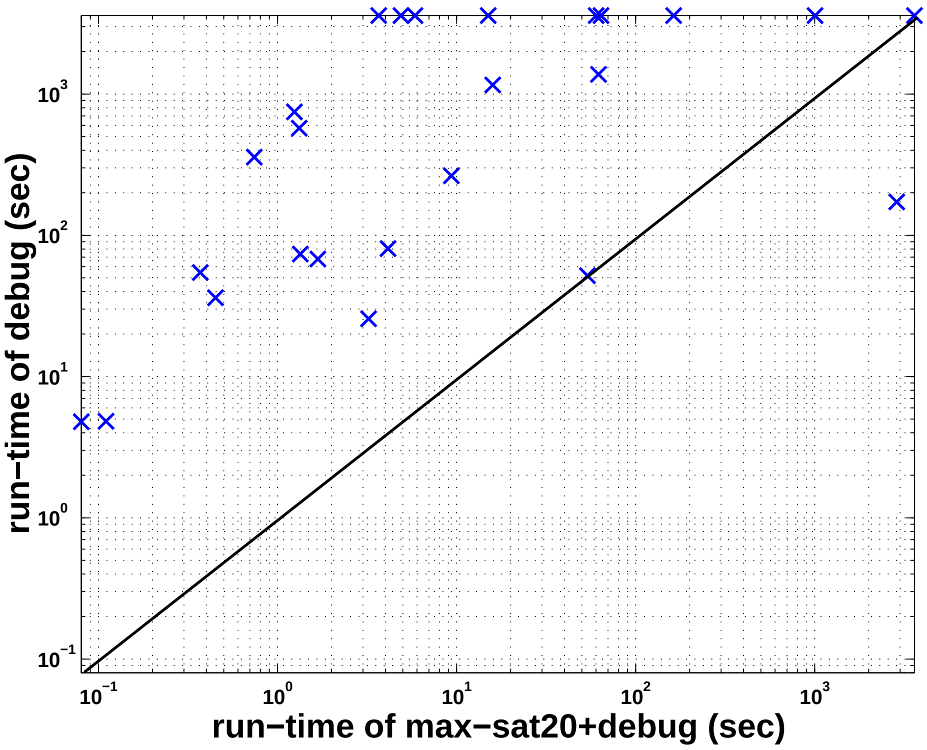

Figure 4.4 CAMUS+debug (max-sat20+debug) versus debug

algorithm, and runtime is little changed from running debug by itself. Experimentally, it was found that groups of 20 clauses each provided the best tradeoff between spending time in CAMUS and saving time in debug, and a hybrid boosted version of debug was created: max-sat20+debug.

Experimental results, shown in Figures 4.4 and 4.5, displayed speed increases in all but two out of 24 instances that either the new or the old technique solved within a timeout. The runtime changes ranged from one to over three orders of magnitude improvement. (One instance was solved in under 4 seconds by the boosted version of debug using CAMUS and timed out after 3600 seconds in the plain debug algorithm.) Further, the boosted system solved 24 out of 26 instances within the timeout, compared to 16 out of 26 for the previous algorithm.

The unsatisfiable core-guided variation of CAMUS was developed after this collaboration. The results in Section 3.5 on the "Diagnosis" benchmarks, from this very application,

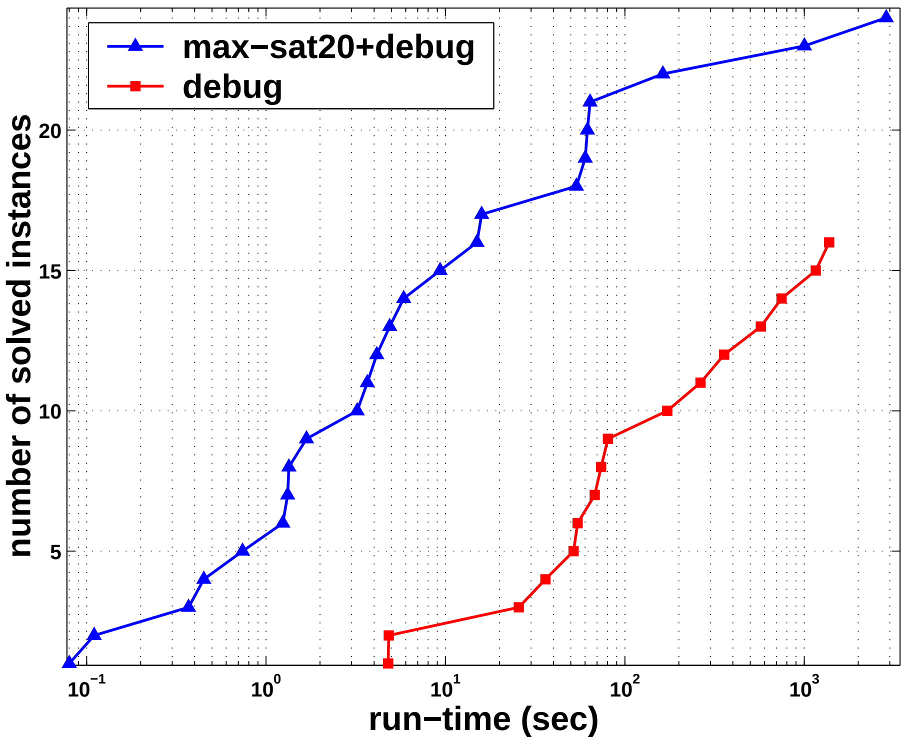

Figure 4.5 CAMUS+debug (max-sat20+debug) versus debug

show substantial improvements in the performance of CAMUS alone on these instances. It remains to be seen whether this makes the first phase of CAMUS, which produces the MCSes that are the error diagnoses sought in this application, competitive on its own with other systems developed specifically for circuit error diagnosis.

4.3 Conclusion

The study of analyzing infeasible constraint systems is still in its infancy, relative to the time and effort that has been put into solution-seeking constraint solvers. As the field grows and capabilities are advanced, new applications are identified; and as new applications are identified, the field is pulled in new directions. The work in this dissertation has formed part of this expanding wave, synthesizing existing work, developing new algorithms for new problems, and enabling practical applications.

It became clear in the early stages of this work that the field was fractured along lines separating large bodies of research. Constraint systems were used and studied in a variety of domains, with different focuses and different terminology in each, and though there was some cross-pollination between domains, it rarely, if ever, occurred in the field of infeasibility analysis. Even if an artificial intelligence researcher were well aware of the constraints-related work occurring in operations research, he or she would be unlikely to know about the few papers coming out of OR, spread across conferences and journals, that were relevant to infeasibility. In my publications, and in this dissertation especially, we have attempted to draw connections between work on infeasibility analysis in different domains where we find it, taking a small step towards consolidating the field.

This research itself occupies an important niche within infeasibility analysis, that of producing *complete* analyses of infeasibility. The majority of the work in the field has been related to producing a single view of the infeasibility of a constraint system, a glimpse of the structure underlying it. Many algorithms were developed to find single, approximate unsatisfiable subsets of constraints or single Max-SAT solutions. We became curious early on about how we could produce a complete view of the structure of infeasibility and what that would even look like.

The algorithms in CAMUS (Compute All Minimal Unsatisfiable Subsets) produce such complete views. These algorithms compute both the complete set of minimal corrections for an unsatisfiable system (MCSes) and the complete set of minimal conflicts (MUSes), exploiting a strong theoretical relationship between the two. And these algorithms immediately illuminated real-world constraint systems that in fact have impossibly-large structures for their infeasibility; computing an entire set of MCSes and/or MUSes was at times intractable. No algorithm could possibly produce the complete view within a lifetime.

This wall of intractability motivated looking for ways to avoid it; CAMUS has been extended in several ways, either pushing the wall back by improving the performance of the algorithms or glancing over the wall by modifying them to produce more than a single glimpse but less than the complete view. The performance increases were achieved by being smarter about where the algorithms look for solutions; we remove or ignore sections of the constraint system that can be shown to be redundant or uninvolved in any infeasibility, which reduces the search space and yields often impressive gains. Relaxing the original question enables us to avoid the intractability; extensions of CAMUS compute subsets or approximations of the complete result set. Throughout all of this, the work was kept as general as possible, often applicable to any type of constraint, to avoid becoming stuck in a single domain and to make the work useful to researchers in other areas.

This basic research and algorithm development was proved practical and needed by real-world applications. We found that two distinct digital logic circuit verification tasks benefited from the complete view of infeasibility provided by CAMUS and its extensions, and in fact it was critical for their performance and functionality. The Reveal equivalence checking system required the ability to compute all MUSes of a constraint system produced during its abstraction refinement flow. Using a single MUS from each constraint system proved to be insufficient, and without using all MUSes, runs would often fail to complete within the experimental timeout. A collaboration with a team of researchers at another university began with the observation that the circuit error diagnoses they were computing were in fact equivalent to the MCSes that CAMUS produced in its first phase. The ability of CAMUS to produce all MCSes (as opposed to the more common result of returning a single Max-SAT solution) was crucial for its application to their work, which resulted in orders of magnitude performance increases. Both of these collaborations were bidirectionally beneficial; the verification system gained impressive performance increases in one direction, and new benchmarks and test cases to spur further development of the algorithms flowed in the other.

Looking forward, the basic research should continue, examining further extensions and enhancements to the core algorithms. The work done so far on exploiting symmetries has unveiled some unforeseen barriers, but it has also provided some tantalizing results about potential for future work. Applying CAMUS to new applications in other domains holds great potential. We plan on using CAMUS to explore optimization problems, mostly arising in the operations research domain; these are satisfiable for any value of the objective function worse than the optimum, but what information could we gain by looking at a system made unsatisfiable by requiring an objective function value better than optimum? And the shift in computer architecture towards multi-core, parallel machines raises the question of parallelizing CAMUS. While work on parallelizing constraint solvers has met difficulties, the "all-solutions" nature of CAMUS makes it a different problem, and it may be better-suited to parallel processing.

Taking a broad view, this work has helped lay a foundation for a more rigorous study of infeasibility. Even in applications that do not require a complete view of the unsatisfiability of constraints, that view can help researchers better understand their constraint systems. Part of the coming work will likely be evangelism, finding others who can benefit from these tools and helping them exploit the work. We also intend to make these analyses more useful by way of visualizations and interactive tools that assist researchers in exploring their own constraint systems. Overall, we hope to continue working in and advancing this field, and we are excited by the possibilities that its relative youth provides.

Appendices

Appendix 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.

Appendix B
Catalog of Benchmarks

Table B.1 lists size characteristics of the benchmark families used throughout this dissertation. These are all CNF formulas, and for each family, the table lists the number of instances "#", and the minimum/median/maximum number of variables and clauses. The benchmark families, their origins, and any salient features are described below.

B.1 Automotive Product Configuration / Benz / Daimler-Chrysler / DC

This is a set of CNF benchmarks from an automotive product configuration domain (80). Each instance encodes a set of available configurations for a product, along with constraints enforcing a specific property to be checked. Analysis showed that the original formulas contained numerous duplicate clauses, which can yield a combinatorial explosion of MUSes; the duplicate clauses were removed 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).

These benchmarks 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, 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. This diversity of results exercises algorithms broadly.

Family	#	Variables			Clauses		
		min	med	max	min	med	max
Benz	84	1,513	1,561	1,891	4,013	5,399	9,957
Reveal	62	1,704	6,268	63,153	4,004	14,347	161,242
Miter	8	1,266	3,761	17,303	1,027	3,434	34,238
Dimacs	20	389	1,919	7,767	1,115	5,108	20,812
AIM	24	50	100	200	80	160	400
FVP-UNSAT-2.0	21	834	5,237	23,910	6,695	89,473	751,118
BMC:Barrel	8	50	1,407	8,903	159	5,383	36,606
BMC:Longmult	16	437	3,319	7,807	1,206	10,335	24,351
BMC:Queueinvar	10	116	886	2,435	399	5,622	20,671
Diagnosis	108	1,880	222,851	4,426,323	5,049	728,516	15,983,633
3SAT (parameterized on n and r)	-		n			$n * r$	

Table B.1 Benchmark Characteristics

Source: http://www-sr.informatik.uni-tuebingen.de/~sinz/DC/

B.2 Reveal

The Reveal benchmarks were generated by a hardware design verification system. The Reveal flow (Section 4.1) 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. For more details, see Section 4.1.

These 62 instances were generated in the abstraction refinement phase of a version of Reveal that used the CNF implementation of CAMUS. Instances were generated from three different designs.

B.3 Miter, Dimacs

These families contain CNF benchmarks generated from performing circuit verification tasks such as equivalence checking on industrial logic circuits. The benchmarks have been made available by João Marques-Silva.

Source: http://sat.inesc.pt/benchmarks/cnf/

B.4 AIM

These are the unsatisfiable instances from the AIM benchmarks, a set of small generated benchmarks often used in MUS papers. All instances have either one or two MUSes, and they are by far the smallest benchmarks used in this work.

Source: http://sat.inesc.pt/benchmarks/cnf/

B.5 FVP-UNSAT-2.0 / nPipe

The nPipe instances from Miroslav Velev's FVP-UNSAT-2.0 benchmarks were "generated in the formal verification of correct superscalar microprocessors."

Source: `http://www.miroslav-velev.com/sat_benchmarks.html`

B.6 BMC

The BMC:[] instances are formulas generated in a bounded model checking (BMC) system as described in (10).

Source:
`http://www.cs.cmu.edu/~modelcheck/bmc/bmc-benchmarks.html`

B.7 Diagnosis / Debugging

Diagnosis: These 108 instances, from the Max-SAT 2008 Evaluation (6), are generated in a process that diagnoses potential error locations in a physical circuit that is producing incorrect output(s) (77). In this application, the MCSes of each instance directly identify the candidate error locations. See Section 4.2 for further details. These instances tend to have small MUSes relative to their total number of clauses, which makes them well-suited to the core-guided extension to CAMUS described in Section 3.5. (The set used in the Max-SAT Evaluation has 112 instances, and II removed 4 that are satisfiable.)

Source: `http://www.maxsat.udl.cat/08/maxsat-industrial.tgz`

B.8 Random 3SAT

Randomized constraint systems were avoided in this work because they do not provide a useful or practical view of performance. Characteristics of randomized instances are much different than those of structured, man-made instances, and conclusions drawn from experiments on randomized instances may not hold for real-world use.

One set of experiments (Section 2.5.2), however, used an algorithm that did not scale well and could not solve most of the industrial benchmarks. For these experiments, we generated sets of small randomized 3SAT instances (CNF formulas with exactly three variables per clause).

The generated 3SAT instances were parameterized on n, the number of variables, and r, the ratio of constraints to variables. Given n and r, each of the $r * n$ clauses were generated by randomly selecting 3 literals (without replacement) given a uniform distribution over all n positive literals $\{x_1, \ldots, x_n\}$ and n negative literals $\{\neg x_1, \ldots, \neg x_n\}$. Duplicate clauses were discarded until $n * r$ clauses had been generated.

To produce a set of unsatisfiable instances with a given set of parameters, we simply generated instances, checked each for satisfiability using a standard SAT solver, and kept the unsatisfiable instances until the desired number had been collected.

Bibliography

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

[2] Fadi A. Aloul, Karem A. Sakallah, and Igor L. Markov. Efficient symmetry breaking for boolean satisfiability. *IEEE Transactions on Computers*, 55(5):549–558, May 2006.

[3] Zaher S. Andraus, Mark H. Liffiton, and Karem 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.

[4] Zaher S. Andraus, Mark H. Liffiton, and Karem A. Sakallah. CEGAR-based formal hardware verification: a case study. Technical Report CSE-TR-531-07, University of Michigan, 2007.

[5] Zaher S. Andraus, Mark H. Liffiton, and Karem A. Sakallah. Reveal: A formal verification tool for verilog designs. In *Proc. 15th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR-2008)*, pages 343–352, November 2008.

[6] Josep Argelich, Chu Min Li, Felip Manyà, and Jordi Planes. Max-SAT evaluation 2008. Website.
`http://www.maxsat.udl.es/08/`.

[7] James Bailey, Thomas Manoukian, and Kotagiri Ramamohanarao. A fast algorithm for computing hypergraph transversals and its application in mining emerging patterns. In *Proceedings of the 3rd IEEE International Conference on Data Mining (ICDM'03)*, page 485, 2003.

[8] James Bailey and Peter 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.

[9] Joaquín Bautista and Jordi Pereira. A GRASP algorithm to solve the unicost set covering problem. *Computers and Operations Research*, 34(10):3162–3173, 2007.

[10] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, volume 1579 of *LNCS*, pages 193–207, 1999.

[11] 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.

[12] S. Bistarelli, U. Montanari, F. Rossi, T. Schiex, G. Verfaillie, and H. Fargier. Semiring-based CSPs and valued CSPs: Frameworks, properties, and comparison. *Constraints*, 4(3):199–240, September 1999.

[13] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM (JACM)*, 44(2):201–236, 1997.

[14] Endre Boros, Khaled M. Elbassioni, Vladimir Gurvich, and Leonid Khachiyan. An efficient implementation of a quasi-polynomial algorithm for generating hypergraph transversals. In *Proceedings of the 11th European Symposioum on Algorithms (ESA 2003)*, volume 2832 of *LNCS*, pages 556–567. Springer, 2003.

[15] S. Bouveret, F. Heras, S. de Givry, J. Larrosa, M. Sanchez, , and T. Schiex. Toolbar. Website. http://carlit.toulouse.inra.fr/cgi-bin/awki.cgi/ToolBarIntro.

[16] Renato Bruni and Antonio 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.

[17] Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. 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)*, pages 78–92, 2002.

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

[19] B. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J.P. Warners. Radio link frequency assignment. *Constraints*, 4(1):79–89, 1999.

[20] John W. Chinneck and Erik W. Dravnieks. Locating minimal infeasible constraint sets in linear programs. *ORSA Journal on Computing*, 3(2):157–168, 1991.

[21] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. 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)*, volume 4501 of *LNCS*, pages 334–339, 2007.

[22] Amy M. Cohn and Cynthia Barnhart. Improving crew scheduling by incorporating key maintenance routing decisions. *Operations Research*, 51(3):387–396, 2003.

[23] Paul T. Darga, Mark H. Liffiton, Karem A. Sakallah, and Igor L. Markov. Exploiting structure in symmetry detection for CNF. In *Proceedings of the 41st Annual Conference on Design Automation (DAC'04)*, pages 530–534, 2004.

[24] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[25] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[26] Gennady Davydov, Inna Davydova, and Hans Kleine 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.

[27] Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[28] Maria J. Garca de la Banda, Peter J. Stuckey, and Jeremy Wazny. Finding all minimal unsatisfiable subsets. In *Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming (PPDP'03)*, pages 32–43, 2003.

[29] Nachum Dershowitz, Ziyad Hanna, and Alexander Nadel. A scalable algorithm for minimal unsatisfiable core extraction. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT-2006)*, volume 4121 of *LNCS*, pages 36–41, 2006.

[30] Bruno Dutertre and Leonardo Mendonça de Moura. A fast linear-arithmetic solver for DPLL(T). In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*, pages 81–94, 2006.

[31] 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.

[32] Thomas Eiter and Georg Gottlob. Identifying the minimal transversals of a hypergraph and related problems. *SIAM Journal on Computing*, 24(6):1278–1304, 1995.

[33] Thomas Eiter and Georg Gottlob. Hypergraph transversal computation and related problems in logic and AI. In *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA 2002)*, pages 549–564, 2002.

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

[35] Eugene C. Freuder and Richard J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70, December 1992.

[36] Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT-2006)*, volume 4121 of *LNCS*, pages 252–265, 2006.

[37] Rafael M. Gasca, Carmelo Del Valle, María Teresa Gómez López, and Rafael Ceballos. NMUS: Structural analysis for improving the derivation of all MUSes in overconstrained numeric CSPs. In *Current Topics in Artificial Intelligence, 12th Conference of*

the Spanish Association for Artificial Intelligence (CAEPIA 2007), volume 4788 of *LNCS*, pages 160–169, 2007.

[38] I.P. Gent, T. Kelsey, S. Linton, J. Pearson, and C.M. Roney-Dougal. Groupoids and conditional symmetry. In *Principles and Practice of Constraint Programming (CP 2007)*, volume 4741 of *LNCS*, pages 823–830, 2007.

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

[40] John Gleeson and Jennifer Ryan. Identifying minimally infeasible subsytems. *ORSA Journal on Computing*, 2(1):61–67, 1990.

[41] Evgueni Goldberg and Yakov 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.

[42] Éric Grégoire, Bertrand Mazure, and Cédric Piette. Local-search extraction of MUSes. *Constraints*, 12(3):325–344, 2007.

[43] Olivier Guieu and John W. Chinneck. Analyzing infeasible mixed-integer and integer linear programs. *INFORMS Journal on Computing*, 11(1):63–77, 1999.

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

[45] Benjamin Han and Shie-Jue 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.

[46] Federico Heras, Javier Larrosa, and Albert Oliveras. MiniMaxSat: An efficient weighted Max-SAT solver. *Journal of Artificial Intelligence Research*, 31:1–32, 2008.

[47] Christian Holzbaur, Francisco Menezes, and Pedro Barahona. Defeasibility in CLP(Q) through generalized slack variables. In *Proceedings of the 2nd International Conference on Principles and Practice of Constraint Programming (CP'96)*, pages 209–223, 1996.

[48] Aimin Hou. A theory of measurement in diagnosis from first principles. *Artificial Intelligence*, 65(2):281–328, 1994.

[49] Jinbo 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.

[50] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[51] Ulrich Junker. QuickXplain: Conflict detection for arbitrary constraint propagation algorithms. In *IJCAI'01 Workshop on Modelling and Solving problems with constraints (CONS-1)*, pages 75–82, 2001.

[52] Ulrich Junker. QuickXplain: Preferred explanations and relaxations for over-constrained problems. In *Proceedings of the 19th AAAI Conference on Artificial Intelligence (AAAI 2004)*, pages 167–172, 2004.

[53] Dimitris J. Kavvadias and Elias C. Stavropoulos. Evaluation of an algorithm for the transversal hypergraph problem. In *Proceedings of the 3rd Workshop on Algorithm Engineering (WAE'99)*, pages 72–84, 1999.

[54] Dimitris J. Kavvadias and Elias C. Stavropoulos. An efficient algorithm for the transversal hypergraph generation. *Journal of Graph Algorithms and Applications*, 9(2):239–264, 2005.

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

[56] Oliver Kullmann. Investigations on autark assignments. *Discrete Applied Mathematics*, 107(1-3):99–137, 2000.

[57] Oliver Kullmann. On the use of autarkies for satisfiability decision. In *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT-2001)*, volume 9 of *Electronic Notes in Discrete Mathematics*, pages 231–253, 2001.

[58] Oliver Kullmann, Inês Lynce, and João Marques-Silva. Categorisation of clauses in conjunctive normal forms: Minimally unsatisfiable sub-clause-sets and the lean kernel. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT-2006)*, volume 4121 of *LNCS*, pages 22–35, 2006.

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

[60] Mark H. Liffiton, Zaher S. Andraus, and Karem A. Sakallah. From Max-SAT to Min-UNSAT: Insights and applications. Technical Report CSE-TR-506-05, University of Michigan, February 2005.

[61] Mark H. Liffiton, Maher N. Mneimneh, Inês Lynce, Zaher S. Andraus, João P. Marques Silva, and Karem A. Sakallah. A branch and bound algorithm for extracting smallest minimal unsatisfiable subformulas. *Constraints*, 2008. In press.

[62] Mark H. Liffiton, Michael D. Moffitt, Martha E. Pollack, and Karem A. Sakallah. Identifying conflicts in overconstrained temporal problems. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI'05)*, pages 205–211, 2005.

[63] Mark H. Liffiton and Karem 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.

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

[65] Mark H. Liffiton and Karem A. Sakallah. Searching for autarkies to trim unsatisfiable clause sets. In *Proceedings of the 11th International Conference on Theory and Applications of Satisfiability Testing (SAT-2008)*, volume 4996 of *LNCS*, pages 182–195, May 2008.

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

[67] João Marques-Silva and Vasco Manquinho. Towards more effective unsatisfiability-based maximum satisfiability algorithms. In *Proceedings of the 11th International Conference on Theory and Applications of Satisfiability Testing (SAT-2008)*, volume 4996 of *LNCS*, pages 225–230, May 2008.

[68] João Marques-Silva and Jordi Planes. On using unsatisfiability for solving maximum satisfiability. *Computing Research Repository*, abs/0712.1097, December 2007.

[69] João Marques-Silva and Jordi Planes. Algorithms for maximum satisfiability using unsatisfiable cores. In *Proceedings of the Conference on Design, Automation, and Test in Europe (DATE'08)*, March 2008.

[70] Maher N. Mneimneh, Inês Lynce, Zaher S. Andraus, João P. Marques Silva, and Karem 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.

[71] Burkhard Monien and Ewald Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discrete Applied Mathematics*, 10(3):287–295, March 1985.

[72] Yoonna Oh, Maher N. Mneimneh, Zaher S. Andraus, Karem A. Sakallah, and Igor L. Markov. AMUSE: a minimally-unsatisfiable subformula extractor. In *Proceedings of the 41st Annual Conference on Design Automation (DAC'04)*, pages 518–523, 2004.

[73] Barry O'Sullivan, Alexandre Papadopoulos, Boi Faltings, and Pearl Pu. Representative explanations for over-constrained problems. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI 2007)*, 2007.

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

[75] Karen Petrie, Chris Jefferson, Tom Kelsey, and Steve Linton. GAPLex: Generalised static symmetry breaking. *Trends in Constraint Programming*, pages 329–376, 2007.

[76] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[77] Sean Safarpour, Mark H. Liffiton, Hratch Mangassarian, Andreas Veneris, and Karem A. Sakallah. Improved design debugging using maximum satisfiability. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD'07)*, pages 13–19, November 2007.

[78] Thomas Schiex, Hélène Fargier, and Gerard Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI'95)*, 1995.

[79] Carsten Sinz. 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)*, pages 827–831, 2005.

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

[81] Andre Sülflow, Görschwin Fey, Roderick Bloem, and Rolf Drechsler. Using unsatisfiable cores to debug multiple design errors. In *Proceedings of the 18th ACM Great Lakes symposium on VLSI, 2008*, pages 77–82, 2008.

[82] Stefan 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.

[83] Allen Van Gelder. Autarky pruning in propositional model elimination reduces failure redundancy. *Journal of Automated Reasoning*, 23(2):137–193, 1999.

[84] Allen Van Gelder. Extracting (easily) checkable proofs from a satisfiability solver that employs both preorder and postorder resolution. *Seventh International Symposium on AI and Mathematics*, 2002.

[85] J.N.M. van Loon. Irreducibly inconsistent systems of linear inequalities. *European Journal of Operational Research*, 8(3):283–288, November 1981.

[86] Jianmin Zhang, Sikun Li, and Shengyu 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.

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

[88] Lintao Zhang and Sharad 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.