University of Cincinnati

Date: 11/9/2012

I, Sean A Weaver , hereby submit this original work as part of the requirements for the degree of Doctor of Philosophy in Computer Science & Engineering.

It is entitled:

Satisfiability Advancements Enabled by State Machines

Student's name: **Sean A Weaver**

This work and its defense approved by:

Committee chair: John Franco, PhD

Committee member: Michal Kouril, PhD

Committee member: Victor Marek, PhD

Committee member: Raj Bhatnagar, PhD

Committee member: John Schlipf, PhD

2957

Satisfiability Advancements Enabled by State Machines

A dissertation submitted to the

Graduate School
of the University of Cincinnati

in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in Computer Science

in the School of Computing Sciences and Informatics
of the College of Enginnering and Applied Science
December 2012

by

Sean Andrew Weaver
M.S. in Computer Science, University of Cincinnati
December 2004.

Dissertation Advisor and Committee Chair: John Franco, Ph.D.

Abstract

This dissertation focuses on research for state-based Satisfiability (SAT) [64–66], a variant of SAT that uses state machines (SMURFs) to represent constraints. Using this constraint representation allows for compact representations of SAT problem instances that retain more ungarbled user-domain information than other more common representations such as Conjunctive Normal Form (CNF). State-base SAT also supports earlier inference deduction during search, the use of powerful search heuristics, and the integration of special purpose constraints and solvers.

SBSAT, a state-based SAT research platform [144], was used and enhanced for both researching the new techniques presented here and gathering experimental data. Since the power of state-based SAT is diminished on problems naturally represented in CNF, the benchmarks used to collect results focus on domains with rich constraints such as verification and model checking.

Acknowledgments

I am very appreciative of the benefits received though my work with John Franco, whom I think of as my academic father. No man aside from my own father has had such a strong influence on my upbringing, he transitioned me from one family to another. He helped refine my interactions with both the scientific community and friends and family by pushing me into the mindset of a free and confident person, enabling me to be a creative and independent thinker and doer. For the eleven years I worked with him, he acted as a sounding board for my ideas, spurring on my creativity. He also provided me with a shelter where my new ideas (born of inexperience) were not rejected, but iteratively refined (sometimes through *aggressive ignorance* in an attempt to find real substance). I cannot thank him enough for the support and opportunities he's given me.

The same can be said for Victor Marek, though he led me through the transition from academic to professional life. The amazing opportunities I've had since graduating in 2005 are because of his determination to see me succeed. Every conversation I've ever had with him has been pure joy (even the occasional request to *polish* the Polish), and I expect many many more.

I would never have had a single success with computers if not for Michal Kouril. He has taught me everything I know about expressing ideas and developing them in elegant and user-friendly ways. Michal's determination and drive - that kind of obsessiveness that's needed to solve truly hard challenges - has been an inspiration to me and I look back on our time working together with great envy. He's discovered three van der Waerden numbers (so far) and I got a front row seat and the honor of sharing a white board with him.

I extend my gratitude and appreciation to the rest of my dissertation committee, Raj Bhatnagar and John Schlipf. They provided me with a great amount of support during my 14 years as a student at UC. My career basically constitutes supporting Artificial Intelligence constructs in a domain-specific functional programming language (go Cryptol). A perfect intersection of the knowledge and appreciation I gained directly from them.

My family attended my dissertation defense. I was elated to finally be able to describe my work

to my them. During my defense, when I saw my Mom, Dad, Sister, and Niece nod, I thought "my family understands me", and being understood by people I care for (or watching them strive to understand) is a wonderful feeling. This memory I quietly slip to the bottom of the stack so that it will be there when all others have faded away. A good friend of mine once said, "It's a keeper!" As well, my family being entertained for an hour with mathematics stands impressive on its own, silently speaking volumes of their love and support. It's a keeper.

I am so proud of Amanda Brennan for taking on so many new challenges and I am so grateful to her for making every effort to create a better life for us both. Her happiness and growth gives my life direction and stability. She is my rock and I am so in love with her.

I will forever honor the friendship of and show deference to my colleague, Marijn Heule. My admiration of him is beyond words. It seems like fate that as our advisors are good old friends, so shall we grow to be also.

I want to thank Denis Bueno for putting up with my incessant nagging about funcsat and for being, in general, a perfect sponge to pour ideas into. In the short time I've known him he has shown a true and refreshing desire for knowledge and a quick mastery of new ideas. He expresses admirable excellence and confidence in his professional and scholarly pursuits, and is never afraid of looking in the eye of or asking questions to those giants whose shoulders (or feet) we stand on.

I would not be in the good position I am today without the loving support of my extended family and friends. Thank you everyone!

Finally, a word of caution: this dissertation pales in comparison to the pinnacle of all human achievement, Q.A.F. by F.S.T. [135]. Since 1997 (If it weren't for all the software changes needed, we'd have renamed that year 0 by now.), all legitimate scholarly works have referenced it, making it the raison d'être for mankind. Here I humbly pay my debt for the many insights gleaned from this nigh holy work and its monarchical author.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter mainly serves as an introduction to state-based Satisfiability. Though, first, details of some of the technologies that support state-based Satisfiability are presented. The structure of this dissertation and some general notation are also given.

1.1 Satisfiability

Satisfiability (SAT) is a formal methods technology recognized as an essential element of solving finite-domain constraint satisfaction problems, i.e. given a set of constraints, find a way to either satisfy them or determine that they cannot be satisfied. Such problems are found in an increasing number of domains such as verification, bioinformatics, cryptography, and planning, to name a few. The recent popularity of SAT is due to a number of reasons, the first of which is its ease of use. *SAT solvers* (computerized tools that solve SAT problems) are, for the most part, totally automated. Traditional constraint solvers (such as theorem provers) tend to require immense human interaction. This is in sharp contrast to SAT solvers, which can be used successfully by a novice who doesn't need to understand anything about how the solver does what it does, i.e. users are expected to treat the SAT solver as a black box. SAT solvers are also plenteous; scores of open source SAT solvers are freely available for download from numerous academic and industrial web-sites. The number and quality of solvers are driven both by yearly competitions and the fact that the main solving algorithms are rather simple to implement. Also, discovering ways of speeding up SAT

solvers can feel very rewarding and involves the kind of thought that really engages the mind of a skilled programmer.

Yet another reason for the rise in popularity of SAT solvers is that they are customizable and pluggable, making for easy integration with other software tools such as package managers, symbolic simulators, SMT solvers, bounded model checkers, equivalence checkers, and so on. Even some established computer languages such as Java [102] and Haskell have well developed interfaces to native SAT solvers[1].

Finally, SAT solvers are revered for their power. Using a SAT solver can sometimes feel like magic as they quickly solve problems that seem impossible. For example, imagine an implementation of the Advanced Encryption Standard (AES) [50][2] that is *incorrect* in such a way that only one key causes the algorithm to be wrong, and perhaps in a malicious way such as: this one malicious key causes AES to realize the identity function, stripping away all security. It seems fantastic that there are SAT solvers capable of finding this one incorrect key (out of 2^{256} possible for AES-256) in a matter of seconds! Because of their ease of use, flexibility, and power, it is often the case that solving a combinatorial problem with a SAT solver turns out to be better (in terms of both development time and solving time) than creating a special purpose solver. And, in the cases where a special purpose solver *is needed* (perhaps for efficiency), SAT is often a useful tool for prototyping ideas; developers can treat SAT as a general purpose decision procedure and test various search techniques and heuristic strategies in order to hone special purpose solvers.

A formal definition of SAT, as well as some examples and the theory behind it, are presented in Section 2.1 on page 16.

1.2 Successes and Pitfalls

At present, Satisfiability (SAT) has replaced Binary Decision Diagrams (BDDs) for being the preferred problem solving method for several areas of formal verification [123] such as equivalence checking [98–100, 122, 151] and model checking [27, 60, 112]. This change is due to recent research in

[1] Cryptol, a domain specific language for cryptography [104], also has an interface to SAT solvers
[2] AES is a cryptographic block cipher with a 128/192/256-bit key and 128-bit plaintext

SAT that has produced solvers much more efficient than BDDs on many practical problems. Also, there has been little compelling research of late providing enhancements to BDDs.

The move from BDDs to SAT happened even though BDDs have many strengths over SAT. As Boolean constraints, BDDs support more powerful conflict and implication detection than Conjunctive Normal Form (CNF) [51, 119] and naturally capture more domain specific information than CNF[3]. In terms of proof complexity, BDD-based refutations are exponentially more powerful than resolution [9], and hence exponentially more powerful than current SAT techniques (which are based on resolution [118]). Also, BDD-based refutations can simulate in polynomial time (p-simulate) [47] resolution, Gaussian calculus, and other proof systems [9], whereas resolution, though powerful for some problem classes, cannot p-simulate BDDs [75][4].

The main limitation with traditional BDD methods is that the pairwise conjunction of two BDDs may need to create an exponential number of intermediate BDD nodes to produce the resulting BDD (even under the best possible variable orderings [36, 147]). In this respect, the conjoining of BDDs is similar to the early Davis-Putnam (DP) algorithm [55] in that both operations may cause exponential memory blow-up; BDD conjunction can create an exponential number of new BDD nodes and DP's resolution step can create an exponential number of new resolvents. In the case of DP, this exponential memory blow-up was made linear by refining DP into a backtracking search algorithm (named Davis-Putnam-Logemann-Loveland (DPLL) [54]).

The recent successes of SAT solvers are due in part to a collection of techniques that occurred within the framework of DPLL, and thus are specialized to work on CNF. These techniques directly contribute to the increased solving power and efficiency of modern SAT solvers. One such technique, conflict clause learning, helps to create a dynamic search space that is DAG-like instead of tree-like as was the case for early DPLL-based algorithms [17]. When this technique is coupled with restarts [83, 94], the resulting algorithm (known as Conflict-Driven Clause Learning (CDCL) [110,

[3] A proof system is deemed more powerful than another if it can simulate the other efficiently, and the latter cannot efficiently simulate the former [16]. This is distinctly different than the feasibility of a method to arrive at a solution for a given problem.

[4] For example, Pigeon Hole [77] problems can be solved in a polynomial number of steps using extended resolution [46] or BDDs [44] and k-XOR (parity) formulas can always be solved in a polynomial number of steps using Gaussian elimination, which BDDs can p-simulate; both of these problems can require an exponential number of resolution steps.

111, 125]) is as strong as general resolution, which is exponentially more powerful than DPLL [17]. This and other such techniques have allowed SAT solvers to be used practically in a wide range of fields such as verification [29,97,138], bioinformatics [107], cryptography [116,129], and planning [49, 149] as well as on many hard combinatorial problems such as van der Waerden numbers [59,95,96]. Also, the recent integration of SAT-based techniques with theory solvers (named Satisfiability Modulo Theories (SMT) [126]) is enabling push-button solutions to industrial-strength problems that previously required expert human interaction to solve (for example, see [30, 76, 85]).

However, SAT has its limitations, many of which (ones believed to be stalling further progress [142]) center around the syntactically restrictive CNF representation. In general, constructing a SAT instance involves translating a problem from some more expressive domain into CNF. This translation can garble domain specific information crucial for efficiently determining a solution to the original problem. For example, CNF contains no explicit information about control flow, and hence, control flow is garbled when encoding a combinational circuit into CNF. Also, the forced 2-level logic of CNF causes CNF-based search heuristics to be limited to purely syntactic considerations.

Some research efforts have pursued a combination of SAT and BDD-based methods (see [6, 51, 64–66, 68, 89, 91, 139]) with hopes that taking the best of both worlds will result in new techniques that overcome both the syntactic limitations of CNF and the memory limitations of BDDs. The intuition here is that applying a generalization of DPLL to collections of BDDs (instead of collections of clauses) effectively removes the exponential memory blow-up of BDDs (as DPLL did for DP) and also opens the doorway for other SAT techniques to tap into the power of BDDs. This enables the development of, for example, more advanced search heuristics and stronger inference and conflict detection mechanisms. The challenges here involve generalizing CNF-based methods, such as Boolean Constraint Propagation (BCP) [148], to BDDs while maintaining efficient search[5]; the SAT community has put significant effort into making CNF-based search methods efficient.

[5] See [87] for such a generalization.

1.3 State-based Satisfiability

Most relevant to this dissertation is the research of [64–66] where collections of BDDs are transformed into collections of state machines and then a DPLL-like search is performed. This area of research is called *state-based SAT*. The limitations of BDDs (that is, variable orderings) and CNF (2-level logic and very limited expression per constraint — that is, clause) disappear with state-based SAT. State machines can be used to efficiently represent clauses, BDDs, and many more (even non-Boolean) functions[6].

State-based SAT enables 1) the use of expressive constraints that can be used to exploit user-domain information, 2) the use of heuristics that capitalize on global semantic relationships between constraints, and 3) the integration of constraint-specific solving techniques, all while supporting (and sometimes strengthening) the recent SAT solving techniques that make SAT solvers so powerful. Briefly, the state-based SAT solving process entails creating a collection of state machines, each of which represent the complete search information for a segment of the input representation, and performing a DPLL-like search. Since complete search information has been memoized for each constraint, inferences are discovered earlier and advanced search heuristics can be applied, potentially reducing the size of the search tree. The ideas and techniques presented in this dissertation are intended to significantly advance the current state of SAT solving by 1) discovering new state-based SAT implementations of domain specific constraints and heuristics, 2) adapting current SAT techniques to the more expressive domain of state-based SAT, 3) enhancing SAT techniques by exploiting the expressiveness of state machines, 4) adapting other constraint solving techniques to state-based SAT, 5) and supporting the development of new techniques for speeding up search that are enabled by state-based SAT[7]. Finally, the state-based SAT paradigm is evaluated and results demonstrating the effectiveness of each of the proposed enhancements are given.

[6] For example, cardinality constraints are trivial to represent as efficient and arc-consistent state machines (see Section 4.1.4 on page 53), but very complex to efficiently represent in CNF [13, 20, 63, 109].

[7] The aim of this dissertation is very similar to that of [58], except that where their focus is on Pseudo-Boolean (PB) constraints, the focus here is on Boolean constraints encoded as SMURFS.

1.4 Structure of the Dissertation

This dissertation is laid out as follows: Chapter 2 on page 16 gives further background information on the technologies that support state-based SAT such as BDDs, CDCL solvers, and SMURFs. The chapter also provides background on the tools supporting this dissertation such as SBSAT, the main research platform used to test and benchmark all the new techniques presented in this dissertation.

Translation to CNF from a domain with rich constraints can cause a loss or blurring of structure, structure important to activate the benefits of state-based SAT. Some domain specific structure can be recovered by searching for common patterns of clauses that correspond to known, and commonly occurring higher level structures. Any corresponding clauses can be discarded in favor of the higher level structures. Techniques along these lines work well when the user domain is known ahead of time. A way to recover structure when the user-domain is not known is to cluster clauses into components. Here, clustering is guided by a heuristic with the aim of approximating higher level structures while remaining functionally equivalent to the original CNF. Chapter 3 on page 30 focuses on these two topics, namely, pattern matching and clustering in support of structure recovery. The chapter provides new preprocessing methods that recover structure from low-level input, transforming it into a set of higher level constraints amenable to state-based SAT. Also, experimental results for each of the described methods are presented.

Chapter 4 on page 48 provides methods for transforming preprocessed user input into an implicit conjunction of SMURFs, the main data structure of state-based SAT [65,66]. SMURFs are capable of representing generic Boolean constraints and support efficient inference propagation and heuristic computation during backtracking search. The chapter provides new precomputation methods and data structures that support and enhance state-based SAT. Specifically, special arc-consistent [71] SMURF data structures are introduced (i.e. produce the same inferences as a general SMURF while providing special compact representations for common Boolean functions), along with techniques that increase sharing among the SMURF collection and relax prohibitively expensive precomputation. Experimental results are presented that weigh the benefits of each precomputation technique both individually and cooperatively.

Preprocessing and precomputation, though both interesting in their own right, are really means

to an end - supporting state-based SAT search. Chapter 5 on page 66 provides details on how state-based SAT supports recent SAT techniques, such as conflict clause learning, and shows how it can be used in tandem with both CDCL solvers and special purpose solvers. Specifically, part of the work of this dissertation involved developing parts of SBSAT into a library and plugging it into the CDCL solver funcsat. In this embodiment, funcsat is driving the search and managing learned conflict clauses while SBSAT provides inferences and witness clauses during BCP. A library for performing Gaussian elimination has also been developed and plugged into SBSAT. In this embodiment SBSAT is driving the search and passing factored `XOR` constraints to the Gaussian elimination solver as they are discovered. Experimental results are presented for both approaches. Finally, Chapter 6 on page 93 provides a summary of the contributions of this dissertation as well as possible future directions.

1.4.1 Notation

This dissertation follows common practices with regard to LaTeX typefaces and scientific terms. The typeface "Sans serif" is used for denoting tools such as SBSAT and funcsat. The typeface "Small caps" is used for denoting algorithms or methods such as Restrict and BddMininf. Finally, the typeface "`Typewriter`" is used for denoting logical operations, Boolean gates, and constants such as `True` and `False`.

Chapter 2

Supporting Technologies

This section provides background on technologies used to support state-based SAT.

2.1 Satisfiability

Satisfiability is a Knowledge Representation mechanism that encodes finite-domain Constratint Satifaction Problems (CSP) [8,57] as Boolean functions. Given a finite-domain CSP $\mathcal{P}$, the Boolean formula $\varphi_{\mathcal{P}}$ encoding $\mathcal{P}$ has the property that there is a one-to-one and onto correspondence between solutions to $\mathcal{P}$ and satisfying assignments for $\varphi_{\mathcal{P}}$. Specifically, the *Satisfiability* (SAT) problem is: given a Boolean function $\varphi_{\mathcal{P}}$ with variables $\vec{x}$, find a mapping of $\vec{x}$ to $\{\texttt{False}, \texttt{True}\}$ that causes $\varphi_{\mathcal{P}}$ to evaluate to `True`. Such a mapping is called a solution or *satisfying assignment* for $\varphi_{\mathcal{P}}$. If a solution exists then $\varphi_{\mathcal{P}}$ is *satisfiable*. If no solution exists then $\varphi_{\mathcal{P}}$ is *unsatisfiable*.

Satisfiability is not the only mechanism that can be used to represent finite-domain CSPs. Other logic-based formalisms expressing CSPs include Reduced OrderedBinary Decision diagrams [35], Answer-Set Programming [14], and, central to this dissertation, SMURFs [65,66]. Traditionally, $\varphi_{\mathcal{P}}$ is expressed as a conjunction of clauses where a *clause* is a disjunction of literals and a *literal* is either a Boolean variable or it's negation, i.e. x or $\overline{x}$. A formula expressed in this form is called a *Conjunctive Normal Form* (CNF) formula. Here is an example CNF formula:

$$(x_1 \vee x_3) \wedge (\overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3}).$$

Assigning value `True` (`False`) to a variable x is denoted by $x \mapsto$ `True` ($x \mapsto$ `False`), respectively. One of the (many) solutions to this formula is $\{x_1 \mapsto \texttt{True}, x_2 \mapsto \texttt{False}, x_3 \mapsto \texttt{False}\}$. Though a solution may be hard to discover[1], checking whether or not a solution satisfies a given CNF formula is linear in the number of clauses of the formula. CNF formulas have become popular because every Boolean function can be re-expressed in CNF with at most a polynomial increase in the size of the resulting CNF formula. Also, every CNF formula can be solved (given enough resources) by repeatedly applying the resolution rule until either no more resolutions are possible (in which case the CNF formula is satisfiable) or the empty clause is generated (in which case the CNF formula is unsatisfiable). The Davis-Putnam (DP) algorithm [55] is one such instance of the repeated application of the resolution rule, a (conceptually) very simple algorithm, though very inefficient on practical problems due to the exponential number of intermediate resolvents that can be generated. As mentioned in Chapter 1 on page 9, DP was quickly reworked into a backtracking search algorithm, named Davis-Putnam-Logemann-Loveland (DPLL), which removed the need to store the intermediate resolvents [54]. Unfortunately, DPLL is a weaker algorithm than DP, that is, DP is as strong as regular resolution while DPLL is only as strong as tree resolution (both of which are exponentially weaker than general resolution) [17].

Recent SAT research has focused on enhancing the DPLL algorithm with the aim of regaining the power of DP while maintaining the memory efficiency of DPLL. Learned resolvents[2] are used to avoid re-exploration of structure sharing sub-trees, resulting in a relaxation of the structure of the search tree. By combining dynamic heuristics with systematic restarts, learned resolvents and new heuristic information are used to transform the search tree into a search DAG in an attempt to avoid heavy tailed behavior [72]. The resulting algorithm, named Conflict-Driven Clause Learning (CDCL) [110, 111, 125], is as strong as general resolution [17], as fast (or faster) than DPLL[3], and does not suffer from the memory problems of DP.

[1] Satisfiability is the canonical NP-complete problem and hence the best known algorithms require $O(2^{|\vec{x}|})$ steps in the worst case.

[2] Namely, those generated via Unique Implication Points (UIPs) in the implication graph (see Section 2.2 on the following page).

[3] Recent efficient data structures, such as watched pointers, have greatly increased the speed at which inferences can be discovered and propagated during search.

2.2 CDCL Techniques

Early SAT solvers used refinements and enhancements to DPLL to solve SAT instances. Recently, these refinements and enhancements have matured, becoming standards in a parameterized algorithm far enough removed from DPLL that it is commonly referred to as Conflict-Driven Clause Learning (CDCL) [110, 111, 125]).

The reasons for the increased solving power of modern SAT solvers lie in the specialized CDCL techniques they employ, namely, conflict directed heuristics [117], watched pointers [69, 106, 117], conflict clause generation [111, 117], conflict clause minimization [18, 70], intelligent conflict clause memory management [10], and dynamic restarts [83, 94], among others. These techniques are tailored to support core CDCL operations such as BCP and heuristic computation and seem to work best when the SAT instance being solved has a small resolution proof, even instances with millions of variables and clauses. Lazy techniques, such as watched pointers and intelligent memory management (garbage collection), are used to efficiently update the state of the solver during search. Conflict directed heuristics and restarts are used to reorder the search DAG in an effort to focus search on unsatisfiable cores. And, conflict clause generation helps detect conflicts earlier by learning new resolvents that keep the search from re-exploring parts of the search space.

The above stated CDCL techniques are necessary components of any competitive state-based SAT solver. Chapter 5 on page 66 presents state-based SAT adaptations for many of these techniques, some of which have already been competitively adapted to state-based SAT (see [66]).

2.3 Binary Decision Diagrams

Another supporting technology is the Binary Decision Diagram (BDD). First introduced by Lee [103], Boute [31], and Akers [1], a BDD is a rooted directed acyclic graph that is commonly used as a compact representation of a Boolean function. Every node is labeled by the name of an input variable, except for the two leaf nodes labeled `T` (for `True`) and `F` (for `False`). Every non-leaf node has two out going edges, one labeled `T` and one labeled `F`. Given a particular ordering of variables, every path from the root to the `True` (`False`) node represents an assignment of values to

input variables that cause the Boolean function to evaluate to `True` (`False`). Reduction rules exist to transform BDDs into a canonical form, namely, Reduced Ordered Binary Decision Diagrams (ROBDDs) [35]. From now on the term BDD is used to mean ROBDD. The basic BDD operations are introduced and discussed in more depth in [33, 35, 131].

BDDs can be used both to preprocess and to solve SAT problems. One way to conceptualize this is to think of BDDs as buckets that hold clauses. When two BDDs are conjoined (i.e. buckets are merged) the BDD reduction rules work to turn the two buckets of clauses into a single canonical form that can be checked for satisfiability in constant time. Specifically, *BDD clustering* is the ordered pairwise conjoining (logical `AND`) of all BDDs in an implicit conjunction.

To determine the satisfiability of a given CNF formula using BDDs, first build one BDD for each clause. Next, choose an ordering under which pairs of BDDs will be conjoined. Finally, use the chosen ordering to perform BDD clustering until either the `False` BDD is created (in which case the formula is unsatisfiable), or until all BDDs have been conjoined, i.e. the *monolithic BDD* is created, in which case the formula is satisfiable and every path from root to leaf represents a partial solution. The drawback here is that the BDD conjunction operation can create an exponential number of new BDD nodes. Fortunately, there are many techniques that work in tandem with clustering to reduce the size of the BDDs. Those presented here include dynamic variable reordering, existential quantification, safe assignments, and the pairwise BDD methods Constrain, Restrict, and Strengthen.

Dynamic variable reordering [124] is crucial to BDD applications because the number of BDD nodes needed to represent a function can fluctuate by orders of magnitude depending on which ordering is used. It is often necessary to find the right variable ordering to efficiently solve hard industrial problems lest the number of BDDs nodes explode during clustering [3, 37, 121, 128, 131].

Existential quantification (see Definition 2.3 below) provides crucial support during clustering by removing variables in the attempt to reduce the size of the conjunction of BDDs. Specifically, existential quantification is used to safely[4] remove variables once they have become isolated in a single BDD. This operation is an essential component of *Early Quantification*, first proposed in [40]

[4] *Safe* here means that the operation does not change the satisfiability of the conjunction of BDDs.

and later applied to SAT in [74]. Since then, the method of early quantification has become an integral part of BDD-based solving (see Section 3.2 on page 38).

Definition Let x be a Boolean variable in the support of a Boolean function φ. Let $\varphi\,|_x$ ($\varphi\,|_{\overline{x}}$) denote the result of the assignment $x \mapsto \texttt{True}$ ($x \mapsto \texttt{False}$) to φ. Existentially quantifying x away from φ means replacing φ with $\varphi\,|_x \vee \varphi\,|_{\overline{x}}$.

In [143] it is shown that, if certain conditions are satisfied, variables may be safely existentially quantified away before being isolated in a single BDD, i.e. existential quantification can *sometimes* be distributed over conjunction. This technique has the added benefit of also being able to assign values to variables during quantification. This technique helps bypass the exponential blowup that can be incurred by conjoining BDDs and, at the same time, simplifies the problem through safe global variable assignments.

There exist a slew of pairwise BDD operations that aim to reduce the size of the BDDs and reveal information that may be exploited during solving (see [66, 80]). CONSTRAIN [48], RESTRICT [48], and STRENGTHEN [141] are just a few. Other such operations exist (see [127] for some others) but only these three will be described here.

Each of these operations take two BDDs, call them f and c, and produces a BDD g that can replace f without modifying the satisfiability of f. In general, this is done by considering the truth tables corresponding to the two BDDs f and c over the union of the support of both f and c, and hence the new BDD g will have support no larger than the union of the support of f and c. The new BDD g is produced by *sibling substitution*, by which is meant that rows of g's truth table which c maps to `True` g maps to the same value that f maps to, and on rows which c maps to `False` g maps to any value, independent of f. It should be clear that $f \wedge c$ and $g \wedge c$ are identical so g may replace f in a conjunction of BDDs without changing the solution space of the conjunction.

The are at least three reasons why this type of operation can be beneficial. The superficial reason is that g can be made smaller than f. A more important reason is that inferences can be discovered. The third reason is that whole BDDs can sometimes be removed from the conjunction.

CONSTRAIN, introduced in [48], is a generalized co-factoring operation that produces a BDD that may be either larger or smaller than f. More importantly, systematic use of this operation

can result in the elimination of BDDs from a conjunction. Unfortunately, by definition, the result of this operation depends on the underlying BDD variable ordering so it cannot be regarded as a logical operation.

The RESTRICT operation, similar to CONSTRAIN in that it is sensitive to BDD variable ordering, is guaranteed to produce a BDD that contains no more variables than f. Basically, RESTRICT prunes the subtrees from f that are either duplicated or in conflict with corresponding subtrees of c. This operation may reveal inferences that could be found by conjoining f and c, but without the combinatorial explosion of conjunction.

The STRENGTHEN operation, like RESTRICT, does not increase the number of variables, but also helps reveal inferences that are missed by RESTRICT due to its sensitivity to variable ordering. Given two BDDs, f and c, STRENGTHEN conjoins f with the *projection* of c onto the variables of f: that is, $f \wedge \exists_{\vec{x}} c$, where $\vec{x}$ is the set of variables appearing in c but not in f. Pseudo code for STRENGTHEN is shown in Algorithm 1.

Algorithm 1 STRENGTHEN(BDD f, BDD c)

```
1: x⃗ := {x : x ∈ SUPPORT(c), x ∉ SUPPORT(f)}
2: for each x ∈ x⃗ do
3:     c := ∃_x c
4: end for
5: return f ∧ c
```

Applying the STRENGTHEN operation to each pair of BDDs considered during clustering sometimes reveals additional inferences. STRENGTHEN provides a way to pass important information from one BDD to another without causing an increase in the number of variables of any BDDs because, before f is conjoined with c, all variables in c that don't occur in f are existentially quantified away. If an inference exists due to just two BDDs, then applying STRENGTHEN to them (i.e. pairwise) will *move* those inferences, even if originally spread across both BDDs, to one of the BDDs. Because STRENGTHEN shares information between BDDs, it can be thought of as strengthening the relationships between BDDs. Also, the added shared structure of these strengthened BDDs can be exploited by a search heuristics, as reported in [66].

2.4 SMURFs

Performing CDCL search on collections of non-clausal Boolean functions requires, foremost, a Boolean function representation; BDDs are used here. Also required are methods to propagate assignments and discover inferences based on a partial assignment, gather heuristic information, and backtrack. Performing search on BDDs has been attempted many times, see [6, 51, 68, 87, 89, 91, 139, 148]. However, performing basic backtracking search operations on BDDs, such as inference detection, can be expensive (see Figure 2.1 for an example). One way to perform efficient search is introduced in [64–66] where inference and heuristic information are precomputed from BDDs and stored as a collection of SMURFs (State Machines Used to Represent Functions), the central component of state-based SAT. SMURFs enable efficient search on collections of non-clausal Boolean functions.

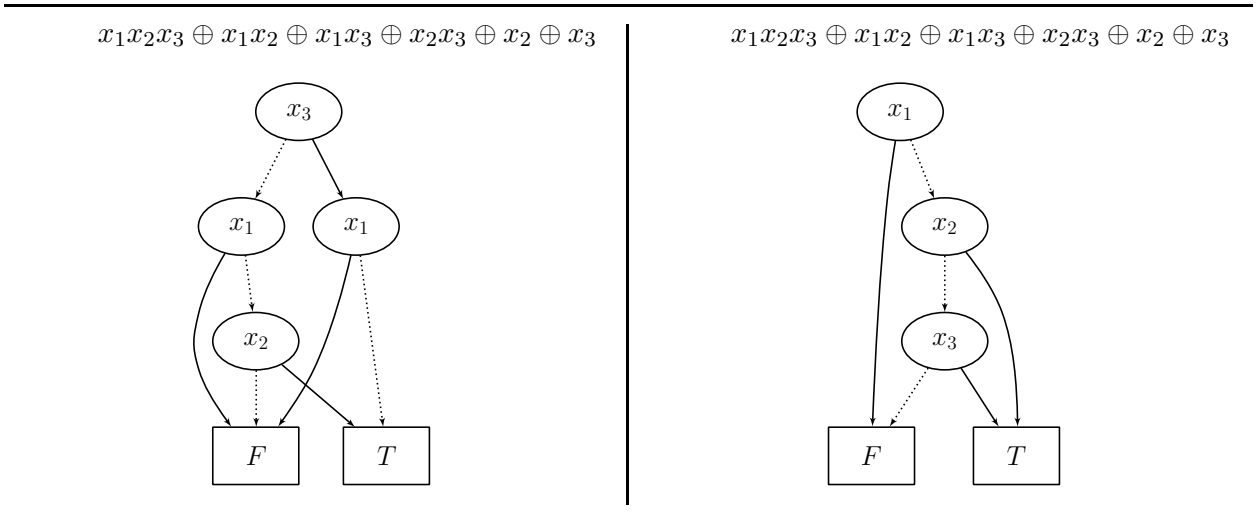

Figure 2.1 — Two BDDs for the Boolean function $x_1x_2x_3 \oplus x_1x_2 \oplus x_1x_3 \oplus x_2x_3 \oplus x_2 \oplus x_3$ built using different variable orderings. The function implies $x_1 \mapsto$ `False`. Discovering this fact is computationally expensive under some BDD variable orderings (e.g. the left BDD) and trivial under others (e.g. the right BDD).

A SMURF is an acyclic Mealy machine [113] where each state (or node) represents a function[5] and transitions (or edges) represent partial assignments to variables of the function. A SMURF state representing a Boolean function φ with n variables has $2n$ transitions, each labeled by a

[5]Though all SMURFs discussed here are purely Boolean SMURFs, SMURFs can represent more than just Boolean functions.

literal representing one of the possible assignments to φ. Following a transition labeled l from a state representing φ will result in the state representing $\varphi\,|_l$. Useful information, such as inference and heuristic computations, can be stored at both the states and transitions of a SMURF. See Figure 2.2 for a graphical depiction of a SMURF and its corresponding BDD.

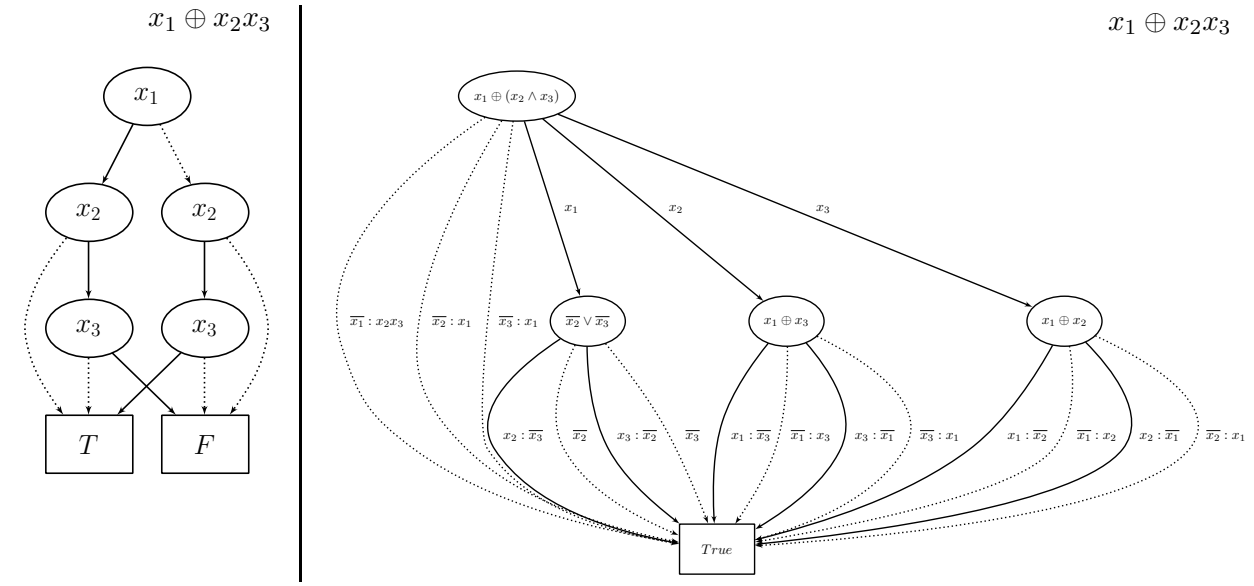

Figure 2.2 — Both the BDD and the SMURF representing the function $x_1 \oplus x_2x_3$. SMURF states are labeled by the Boolean function they represent. SMURF transitions are labeled by the assigned variable and list of inferences (if any exist).

Forward computation, such as applying an inference to a SMURF, is performed by traversing the appropriate transition. Precomputed inference and heuristic information can be read directly off the current state of the SMURFs. Backtrack on SMURFs is facilitated by either saving the current state of every SMURF at each decision level or maintaining a stack of state changes, components that CNF-based SAT has been able to massively improve upon through the use of lazy data structures. Once a collection of SMURFs has been built, something which can be quite computationally expensive, efficient search can proceed. Pseudo-code outlining how to build a SMURF from a BDD is shown in Algorithm 2.

Theorem 2.4.1 *A* SMURF *representing a Boolean function can have no more than* 3^n *states, each corresponding to a partial truth assignment on* n *variables [66].*

Algorithm 2 CreateSmurf(BDD b)
A *cube* is a BDD representing a conjunction of literals.
The CoFactor operation takes a BDD b and a literal l and produces $b|_l$.

1: **if** b.Smurf $\neq 0$ **then** ▷ If a Smurf already exists for b then just return it
2: **return** b.Smurf
3: **end if**
4: **for** each variable $x_i \in$ Support(b) **do**
5: **for** each $l_i \in \{x_i, \overline{x}_i\}$ **do** ▷ Create both the positive and negative transitions
6: BDD $b|_{l_i}$:= CoFactor(b, l_i)
7: BDD $I_{b|_{l_i}}$:= the cube of inferences of $b|_{l_i}$
8: BDD $b|_{l_i I_{b|_{l_i}}}$:= CoFactor($b|_{l_i}, I_{b|_{l_i}}$)
9: $S_b.transition(l_i).inferences := I_{b|_{l_i}}$
10: $S_b.transition(l_i).next$:= CreateSmurf($b|_{l_i I_{b|_{l_i}}}$)
11: **end for**
12: **end for**
13: b.Smurf $= S_b$
14: **return** S_b

Proof A Smurf's root has $2n$ outgoing transitions, each connecting the root to a unique Smurf state in the worst case. Smurf states at transition level one (after one transition from the root) have at most $2(n-1)$ transitions. Transitioning from the root on a set of literals, regardless of order, always results in the same Smurf state, e.g. transitioning on a literal x_1 then x_2 results in the same Smurf state as transitioning on literal x_2 then x_1. Hence, the worst-case number of unique states at level k of a Smurf is the number of sets of k literals drawn from n variables, namely, $\binom{n}{k}2^k$. Summing up the number of states at each transition level gives:

$$\sum_{k=0}^{n} \binom{n}{k} 2^k.$$

This is an upper bound on the number of states in a Smurf for a Boolean function, which by the Binomial theorem is 3^n. □

Even though this is a one-time cost, it can be prohibitively expensive. Some attempts have been made to overcome this potentially exponential number of states. Smurfs, as presented in [64–66], are compressed in a few ways:

1. The Smurf state representing a specific Boolean function is created only once, even though

that state may exist across several SMURFs.

2. All single literal inferences are memoized on SMURF transitions. Hence, SMURFs maintain *arc-consistency* [71], i.e. have maximal implicativity [119].

3. The functions $x_1 \vee .. \vee x_k$, $x_1 \oplus .. \oplus x_k$, and $x_1 \equiv (x_2 \wedge .. \wedge x_k)$ (allowing for the arbitrary negation of variables) can be represented using compact special counter-based SMURFs due to certain symmetries in these functions [66].

The SMURF data structure also supports the use of advanced heuristics. One such heuristic is the Locally Skewed, Globally Balanced (LSGB) heuristic [66]. This heuristic is modeled after the CNF-based Johnson (or Jeroslow-Wang) heuristic [88, 92] and generalized for SMURFs. LSGB attempts to direct search into early conflicts by causing inferences to happen near the top of the search tree. This heuristic makes heavy use of the SMURF data structure by performing a full look-ahead of each SMURF during precomputation and memoizing inference information at each SMURF state. The heuristic then has immediate access to the memoized inference information during search.

2.5 Supporting Tools

This section provides descriptions of the three main tools supporting this dissertation, namely SBSAT, funcsat, and the BDD Visualizer.

2.5.1 SBSAT

Recent interest in problems such as property checking, circuit verification, and circuit synthesis has led to problem descriptions in new non-clausal formats such as Trace [146] (a SSA language for hardware verification), AIGER [23] (a format describing And/Inverter Graphs), and a non-CNF DIMACS format [12] (sometimes referred to as EDIMACS that allows users to specify Boolean gates as constraints). Problems described in these formats are solved by various means, commonly involving the use of Boolean formula DAGs [136], Negation Normal Form (NNF) [86] formulas, or

BDDs (a few references are [51, 66, 73, 78, 91]) to represent clusters of connected components[6].

SBSAT [64–66, 144], a C-based implementation of a state-based SAT solver, allows the user to reason about certain non-clausal inputs via support for the Trace and AIGER formats as well as BDDs, CNF, and various other input formats. SBSAT initially represents each constraint as a BDD. After an advanced and modular preprocessing phase, SMURFs are constructed from the collection of BDDs. Finally, a CDCL-based search can be performed (though other solvers, such as a BDD-based variant on Stochastic Local Search (SLS), come prepackaged as well). SBSAT supports some current CDCL-based techniques such as conflict clause learning and restarts. However, other CDCL-based techniques, such as lazy updating of data structures, conflict clause minimization, among others, have not been implemented in SBSAT, and its performance can lag behind more modern solvers. This dissertation detail improvements that, when added to SBSAT, enable it to solve the same problems as modern SAT solvers. In addition, SBSAT has the potential to outperform modern SAT solvers because it is capable of performing search on more expressive constraint representations than CNF, enabling future use of more powerful proof systems than those which are resolution-based.

As discussed in Chapter 4 on page 48, SMURF precomputation can be very expensive, depending on the size of the input BDDs. In practice, it is not feasible to build SMURFs for BDDs with much more than twelve variables because such a SMURF could need roughly 500,000 nodes. These numbers appear to make state-based SAT infeasible, and certainly SBSAT would be a useless SAT solver if it could not handle constraints with more than twelve variables. SBSAT mitigates this huge deficiency by building special SMURFs that exploit common symmetries of constraint types that occur often in practice. By using a combination of general and special SMURFs, SBSAT can compactly represent large sets of BDDs; BDDs which cannot be represented as CNF without adding either a significant number of extra variables or clauses (or both), e.g. XOR constraints and cardinality constraints [13, 109]. However, these basic techniques alone are often not powerful enough to enable successful solving of real world problems[7].

[6] See Section 3.1 on page 30 for information about recovering (ungarbling) user-domain information from input formulas and Section 3.2 on page 38 for BDD-based clustering techniques in support of finding compact sets of constraints.

[7] Chapter 4 on page 48 describes SMURF-based symmetry detection and compression techniques which better compress and speed up computation on SMURFs and increase the scalability of state-based SAT search.

SBSAT is written in a highly structured way that makes it easy to implement and test new techniques. Because of this, many of the techniques proposed here were implemented in SBSAT and also many of the results were collected using SBSAT.

2.5.2 Funcsat

Funcsat [38], a C implementation of the Haskell-based funsat [39], is a conflict driven, clause learning SAT solver. It has many execution hooks which allow it to function as a client solver of another solver. It features cache-aware watched literals and conflict clause pool, any-UIP conflict clause generation, Glucose-style clause deletion heuristic [11], non-chronological backtracking, and rapid restarts. Many of these features have tunable parameters in order to better cooperate as a client solver. Funcsat also functions as an incremental solver, allowing unit assumption literals to be pushed and popped.

2.5.3 BDD Visualizer

The BDD Visualizer [140], a tool developed as part of this dissertation, generates visualizations of BDDs. A simple web-based interface to the visualizer is located at:
http://www.cs.uc.edu/~weaversa/BDD_Visualizer.html and is freely available for anyone to use. The page has, among other things, a text-box and a submit button. Upon clicking the submit button, two programs, SBSAT [144] and Graphviz [61], interpret the text written in the text-box and attempt to create graphs of BDDs corresponding to the text. If the text is not formatted correctly, an error message is displayed. As of this writing, the BDD Visualizer has been used to generate more than 3000 visualizations[8]. Other BDD visualization tools exist (see [115] for an overview), and each has some role to play. Some, such as BDD Scout [115] allow the user to interactively explore the generated visualization. Another, visBDD [114], has an audio component that speaks the steps of the ITE algorithm during BDD construction.

The BDD Visualizer is a simple web-based application that, with one button push, will produce visualizations in any Graphviz supported format, such as PDF or PNG. The BDD Visualizer supports

[8]This statistic does not include visualizations generated by the developer.

SBSAT's canonical form input language, making it easy to express and reorder BDDs. Figures 2.3 and 2.4 provide examples of how to use the visualizer. A derivative of this tool has been created that visualizes SMURFs instead of BDDs. This dissertation makes heavy use of both of these tools to help illustrate the various data structures and techniques presented within.

```
order(x4, x3, x2, x1)

; Create the first BDD
; it can be referred to using $1.
xor(and(x1, x2), and(x1, x3, x4), x4)

; Create the second BDD
; it can be referred to using $2.
xor(x1, x2, x3)

; Print the disjunction of the two BDDs.
print(or($1, $2))
```

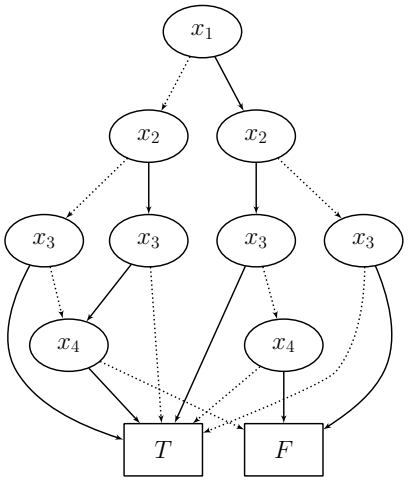

Figure 2.3 — An example demonstrating the use of the BDD Visualizer. Input to the visualizer is given on the left. The resulting BDD visualization is given on the right. `True` edges are denoted by solid lines and `False` edges by dotted lines.

```
order(x1, x2, x3, x4)

; Create the first BDD
; it can be referred to using $1.
xor(and(x1, x2), and(x1, x3, x4), x4)

; Create the second BDD
; it can be referred to using $2.
xor(x1, x2, x3)

; Print the disjunction of the two BDDs.
print(or($1, $2))
```

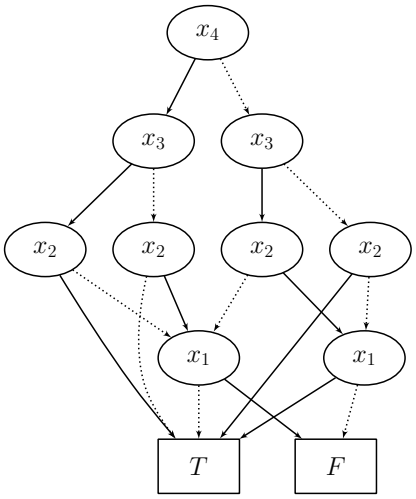

Figure 2.4 — The same function is visualized as in the previous figure, but under a different BDD variable ordering.

Chapter 3

Preprocessing

This chapter presents ideas and extensions needed to perform efficient state-based SAT search on low-level input languages such as CNF. Specifically, Section 3.1 details ideas and provides results related to CNF-based pattern matching and clustering and Section 3.2 on page 38 details ideas and results related to BDD clustering. The techniques discussed in this chapter are all in support of recovering user-domain information so that intelligent and efficient search can be performed on expressive SMURFs. The techniques presented here are used to preprocess SAT instances into a form that is much more amenable to state-based SAT search than, for example, CNF. The success of state-based SAT search is reliant upon these techniques because, when SMURFs are built directly from CNF, a state-based SAT solver can't perform any better than a purely CNF-based SAT solver. Reverse engineering structure and user-domain constraints, creating clusters from tightly connected components, and simplifying the clusters by inference, existential quantification, and pairwise BDD operations all act to massage low-level input into a form that is ripe for use with state-based SAT.

3.1 Structure Recovery

This section provides details for some of the current methods used to recover user-domain information from SAT instances in support of better state-based SAT search, and presents new methods that are complementary to current methods.

3.1.1 Overview

In practice, SAT instances are generated by encoding user-domain constraints (or components) into low-level input languages such as CNF and AIGER. The translation process entails high-level constraints being decomposed into either sets of clauses (in the case of CNF) or into `NAND` gates (in the case of AIGER). This is undesirable for two reasons:

1. User-domain information can become garbled, information that may be crucial for efficiently determining a solution to the original instance.

2. It is common for the encoding process to drastically increase the size of the formula either by an exponential blow up of the number of constraints, or by adding a polynomial number of variables. See, for example, the encoding of cardinality constraints into CNF [13, 20, 63, 109].

Recovering user-domain information is important because it can be used to support the development of advanced heuristics, faster inference propagation, and earlier discovery of conflicts, all of which state-based SAT supports. However, the time taken to perfectly recover the structure of a low-level SAT instance can sometimes outweigh the time taken to solve it. For this reason, it is often better to approximate the structure of the original formula. This can be done by identifying and clustering tightly coupled components that are loosely coupled with other components; this type of structure is very common, e.g. encodings of circuits into SAT. Knowing the circuit structure of a circuit-based SAT problem allows solvers to make use of recent synthesis and equivalence checking techniques such as *random simulation* as applied to *AIG structural hashing*, a technique that probabilistically detects stuck-at variables and equivalence relations [105] and can drastically reduce the solving time of an instance. Also, [67] shows how to use structural hashing [105] and local observability don't-cares [15, 151] to optimize a circuit for solving after reverse engineering circuit structure from CNF. Though circuit structure (typically stemming from verification problems) is used here to illustrate various clustering methods, clustering can be successfully used in other domains (see [44] for a simple example). As well, there are many different clustering techniques that have been developed which can be used for structure recovery. Utilizing the right one for processing a given instance can cause orders of magnitude reduction in both the number of

variables and constraints in the processed instance. In terms of state-based SAT, understanding these algorithms and applying them correctly is crucial to building compact state machines that make use of the domain specific information of the original problem.

One simple way to recover some user-domain constraints is to use pattern matching. *Pattern matching* involves reconstructing simple Boolean constraints by matching subsets of clauses against patterns noticed in their standard CNF representations [67, 120] (also see [41] for an And/Inverter Graph (AIG)-based reconstruction method). Recovering user-domain information can be simple if the CNF encoding processes are known. However, if Boolean constraints are not encoded into CNF in an expected way then pattern matching can fail. Fortunately, machines (more often than people) encode problems into SAT and hence SAT encodings of primitive operations, like logic gates, have become standardized. For example, instances representing questions about Boolean circuits commonly contain the following set of Boolean gates encoded into CNF [137]: `XOR`, `AND`, `OR`, `ITE`, and `MAJV`. The Boolean gate $x_0 = \texttt{OR}(x_1, x_2, x_3)$ is commonly encoded into the following set of clauses:

$$(\overline{x_0} \vee x_1 \vee x_2 \vee x_3) \wedge (x_0 \vee \overline{x_1}) \wedge (x_0 \vee \overline{x_2}) \wedge (x_0 \vee \overline{x_3}).$$

Using pattern matching to discover the Boolean `OR` gate encoded into this CNF formula involves searching for n two-literal clauses (with certain properties) and a corresponding n literal clause. However, this pattern matching technique will fail to find the `OR` gate when given the following CNF (also encoding $x_0 = \texttt{OR}(x_1, x_2, x_3)$, but which is hardly ever found in the wild):

$$(\overline{x_0} \vee x_1 \vee x_2 \vee x_3) \wedge (x_0 \vee \overline{x_1} \vee x_2 \vee x_3) \wedge (x_0 \vee \overline{x_2} \vee x_3) \wedge (x_0 \vee \overline{x_3}).$$

There are an overwhelming number of different CNF representations for a given Boolean gate-type. The reason that the first encoding is preferred is because it is minimal in both the number of literals and number of clauses needed to represent the function in CNF (not so for the second or any other encoding of the function). So, though it is not feasible to develop CNF pattern matching techniques for every encoding of every Boolean gate-type, due to minimality, this is not necessary. Only a handful of gate encodings need to be considered to be able to use pattern matching to

successfully recover the structure of the majority of practical SAT problems. However, if pattern matching fails, more complex clustering techniques can be used to search for sets of clauses that seem to have a strong relationship. Such clauses can be clustered into one constraint, i.e. by building a BDD from the set of clauses. As the results presented in Table 3.2 on page 37 show, clustering can be a remarkably good way of recovering structure that is missed by pattern matching.

3.1.2 Pattern Matching

In [120], pattern matching is implemented using a specialized stack-based pattern matching algorithm for each gate-type. This specialization makes it feasible to recover Boolean gates with a large number of fan-ins, but makes it time consuming to add new pattern matching algorithms for future gate encodings. In [67], Boolean gates are recovered from CNF by a single pattern matching algorithm and a user-definable gate-library. The reported results were gathered using a gate-library consisting of `AND` and `OR` gates up to a fan-in of ten inputs, `XOR` gates up to a fan-in of four inputs, and `MAJV` gates. The restrictions on gate fan-in exist to make the generic pattern matching algorithm perform efficiently on standard CNF benchmarks. However, unlike [120], it is relatively easy to add patterns for new gate-types and the pattern matching algorithm naturally covers all possible encodings for each gate-type in the library.

Gate recovery has been implemented in SBSAT similar to [120], i.e. specialized pattern matching algorithms were developed for each desired gate-type. SBSAT can pattern match against `AND`, `OR`, `XOR`, `MAJV`, and `ITE` gates. SBSAT also makes use of an idea from [67], namely, using key clauses to enhance pattern matching. A *key clause* is one that contains all of the variables of its corresponding constraint. It is typical for gate encodings to produce at least one clause containing all the input and output variables of the gate. This is important because once the key clause has been found, all other clauses also belonging to the gate have only these variables, reducing the number of clauses that need to be considered during pattern matching. For example, the gate $x_0 = \texttt{OR}(x_1, x_2, x_3)$ has key clause $\overline{x_0} \vee x_1 \vee x_2 \vee x_3$, hence, only clauses with the variables x_0, x_1, x_2, and x_3 need to be considered to know if the above `OR` gate exists in the CNF formula.

When performing CNF pattern matching, SBSAT first finds all key clauses in the CNF for a

particular gate-type and then iterates over the key clauses, searching for the set of clauses that correspond to each key clause for the desired gate-type. For every matching set of clauses found, a BDD is created for the Boolean gate and the set of clauses is marked for deletion. Marked clauses are deleted at certain stages of the recovery process. This approach is different than [67] where key clauses are not deleted until all gates have been matched. Though implementing different pattern matching algorithms for each gate-type is time consuming, this approach enables gate specific optimizations that overcome the fan-in restrictions of [67], and is very efficient on real world problems.

After all gate-types have been processed, SBSAT runs a simple and parameterized clustering algorithm on the remaining clauses with the goal of discovering constraints for which SBSAT does not have specific pattern matching algorithms. SBSAT first chooses the longest clause c_{key} that has not yet been considered (presuming this to be a key clause of an unknown user-domain constraint) and builds the set of clauses $C_{c_{key}}$ containing all clauses with smaller or equal length who's variables are a subset (or almost a subset) of c_{key}. Specifically,

$$C_{c_{key}} = \{c_i : c_i \in C, |c_{key}| \geq |c_{key} \cup \{c_i\}| - \delta\}. \tag{3.1}$$

Here, the $\cup$ operator is treating clauses as sets of positive variables rather than sets of literals. When δ is 0, all clauses in $C_{c_{key}}$ are subsets of the variables of c_{key}. If the set $C_{c_{key}}$ contains more than just the clause c_{key}, a new BDD is built by conjoining the clauses of $C_{c_{key}}$ and then marking them for deletion.

3.1.3 Experimental Results

Table 3.1 on page 36 show results of the newly introduced CNF reverse engineering methods (with δ from Equation 3.1 equal to 0) applied to the benchmark sets used in [120] and [67]. The benchmarks used come from a wide variety of industrial domains such as verification, equivalence checking, and bounded model checking and also include benchmarks from the crafted categories of previous SAT competitions [21]. Only `AND`, `OR`, and `XOR` gates are explicitly matched against as any `MAJV` and

`ITE` gates are efficiently recovered by the generic constraint recovery method since they each have a small fixed number of variables (four per constraint). The results show that the pattern matching techniques introduced in this chapter always find the same or more gates than prior work, and also take less time. The compression column given in Table 3.1 on the next page (labeled "Comp.") shows the ratio of the original number of clauses to the resulting number of BDDs. The reduction in the number of constraints is generally between three and four times, though sometimes much larger. Also, the runtime needed to find even a large number of gates is not prohibitive for either small or large instances. This means that pattern matching alone can efficiently reduce the number of constraints of real world problem instances, a major step on the path to creating expressive and compact SMURFs. For details on how each preprocessed constraint is transformed into a SMURF, see Chapter 4 on page 48. For experimental results showing the effects that preprocessing has on state-based SAT search, see Section 5.1.2 on page 76.

Pattern matching techniques can also be used to find other common Boolean formulas. Cardinality constraints are useful constraints commonly found in SAT instances. A *cardinality constraint* is a Boolean function of the form:

$$min \leq x_1 + .. + x_k \leq max$$

where min and max are positive integers, $x_1..x_k$ are Boolean variables, and the constraint is satisfied if the number of Boolean variables taking the value `True` is greater than or equal to min and less than or equal to max.

Cardinality constraints are much more expressive than other common Boolean constraints, and as such, developing a specialized CNF-based pattern matching algorithm is difficult. In contrast to XOR constraints, there is hardly any work on the recovery of cardinality constraints from CNF formulas. However, the key clause clustering method (described earlier, Equation (3.1) on the preceding page) can recover cardinality constraints from CNF input. This method was tested on the unsatisfiable SGEN [133] benchmarks (from the SAT-2009 competition [21]) that consist of sets of cardinality constraints encoded into CNF. The methods introduced in this chapter completely

Table 3.1 — Results of this chapter's CNF reverse engineering methods on benchmarks of [120] and [67].

Instance	Number Clauses	Number Variables	Number XORs	Number ∧/∨ Gate	Number Unknown	Number Unit	Clauses Leftover	Comp.	Time(s)
par8-1-c	254	64	56	15	0	0	0	3.57	0.01
par8-1	1149	350	240	80	0	43	80	2.59	0.01
par16-1-c	1264	317	270	61	0	0	31	3.49	0.01
par16-1	3310	1015	768	192	0	76	192	2.69	0.01
par32-1-c	5254	1315	1158	186	0	0	125	3.57	0.01
par32-1	10277	3176	2624	448	0	141	448	2.80	0.02
barrel5	5383	1407	1065	152	0	0	701	2.80	0.01
barrel6	8931	2306	1746	254	0	0	1153	2.83	0.02
barrel7	13765	3523	2667	394	0	0	1765	2.85	0.02
barrel8	20083	5106	3864	578	0	0	2561	2.86	0.03
barrel9	36606	8903	7164	812	0	0	4456	2.94	0.05
ssa2670-130	3321	1359	883	271	44	4	461	1.99	0.01
ucsc-bf1355-348	7271	2286	1190	468	440	4	2510	1.57	0.01
dubois100	800	300	200	0	0	0	0	4.00	0.01
dlx1_c	1614	287	0	216	67	0	2	5.66	0.01
1dlx_c_mc_ex_bp_f	3725	766	1	565	178	1	0	5.00	0.01
dlx2_cc_bug18	20208	2043	0	1	1905	0	4	10.58	0.21
2dlx_ca_mc_ex_bp_f	24640	3186	1	2466	782	1	0	7.58	0.03
2dlx_cc_mc_ex_bp_f	41704	4524	1	3610	1034	1	0	8.97	0.06
2dlx_cc...f2_bug019	48232	4824	1	3762	2563	1	0	7.62	0.07
9vliw_bp_mc	179492	19148	1	15369	2326	1	0	10.14	0.26
c499	1870	606	352	213	0	1	0	3.30	0.01
3bitadd_32	32316	4480	0	0	0	0	32316	1.00	0.14
x1.1_16	122	46	31	0	0	0	0	3.93	0.01
x2_128	1018	382	255	0	0	0	0	3.99	0.01
longmult12	18645	5974	1569	4351	0	47	0	3.12	0.03
longmult14	22389	7176	1805	5317	0	51	0	3.12	0.03
longmult15	24351	7807	1923	5830	0	53	0	3.11	0.04
ibm...3_02_3-k95	272059	73525	11836	56042	88	281	1053	3.92	0.37
ibm...23-k100	861175	207606	16398	186625	0	692	1170	4.20	1.19
hanoi6	39666	4968	0	1890	3402	1005	16677	1.72	0.07
Mat26	2464	744	256	480	0	0	0	3.34	0.01
Mat317	85050	24435	10935	13770	0	0	0	3.44	0.13
linvrinv8	6337	1920	896	960	0	0	1	3.41	0.01
linvrinv9	9154	2754	1296	1377	0	0	1	3.42	0.02
equilarge_l5	18519	4478	4167	856	2	0	248	3.51	0.03
pyh...unsat-40-4-01	31795	9638	3199	6358	0	83	0	3.29	0.05
clauses-2	272784	75527	5413	69152	192	1	0	3.64	0.44
clauses-4	1002957	267766	19600	246116	432	1	0	3.76	1.78
clauses-6	2623082	683995	47453	632996	768	1	0	3.85	5.05
clauses-8	5687554	1461771	94441	1361880	1200	1	0	3.90	11.63
clauses-10	8901946	2270929	147877	2115466	1552	1	0	3.93	18.89

Table 3.2 — Results of this chapter's CNF reverse engineering methods on the SGEN benchmarks.

Instance	Number Clauses	Number Variables	Number Cardinality	Clauses Leftover	Comp.	Time(s)
sgen1-unsat-61-100	132	61	30	0	4.40	0.017
sgen1-unsat-73-100	156	73	36	0	4.33	0.017
sgen1-unsat-85-100	180	85	42	0	4.28	0.017
sgen1-unsat-97-100	204	97	48	0	4.25	0.018
sgen1-unsat-103-100	220	105	52	0	4.23	0.018
sgen1-unsat-109-100	228	109	54	0	4.22	0.018
sgen1-unsat-115-100	244	117	58	0	4.20	0.018
sgen1-unsat-121-100	252	121	60	0	4.20	0.018
sgen1-unsat-127-100	268	129	64	0	4.18	0.018
sgen1-unsat-133-100	276	133	66	0	4.18	0.018
sgen1-unsat-139-100	292	141	70	0	4.17	0.018
sgen1-unsat-145-100	300	145	72	0	4.16	0.019
sgen1-unsat-151-100	316	153	76	0	4.15	0.019

Table 3.3 — Results of this chapter's CNF reverse engineering methods on the sliding window benchmarks.

Instance	Number Clauses	Number Variables	Number Unknown	Clauses Leftover	Comp.	Time(s)
slider_40_sat.cnf	840	39	40	0	21.00	0.02
slider_40_unsat.cnf	720	39	40	0	18.00	0.02
slider_60_sat.cnf	1260	59	60	0	21.00	0.02
slider_60_unsat.cnf	1980	59	60	0	33.00	0.03
slider_70_sat.cnf	1470	69	70	0	21.00	0.02
slider_70_unsat.cnf	2310	69	70	0	33.00	0.03
slider_80_sat.cnf	1680	79	80	0	21.00	0.02
slider_80_unsat.cnf	2640	79	80	0	33.00	0.04
slider_100_sat.cnf	2100	99	100	0	21.00	0.03
slider_100_unsat.cnf	3300	99	100	0	33.00	0.04
slider_120_sat.cnf	2520	119	120	0	21.00	0.03
slider_120_unsat.cnf	3960	119	120	0	33.00	0.05

and automatically recover the cardinality constraint representation exactly as it is described in the SGEN benchmark generation paper [133]. Results for SGEN cardinality constraint recovery are presented in Table 3.2.

The key clause clustering method can also recover more generic constraints from CNF input. Table 3.3 shows SBSAT completely and automatically recovering the original structure of the "sliding window problems" of [65,66]. These benchmarks are crafted problems that model key properties of Bounded Model Checking problems and consist of a handful of small random Boolean constraints with balanced truth tables.

3.2 BDD Clustering

The previous section showed how to efficiently reverse engineer some common Boolean constraints from CNF. The next preprocessing technique explored here is BDD clustering. This section presents improved BDD clustering techniques and methods that work in support of state-based SAT search by creating BDDs that can be transformed into expressive and compact SMURFs.

3.2.1 Overview

Though there has been a lot of research into different heuristics for BDD clustering , the generic processes of clustering BDDs is relatively simple. As earlier discussed in Section 2.3 on page 18, BDD clustering is the ordered pairwise conjoining (logical `AND`) of an implicit conjunction of BDDs. Also, variables can be existentially quantified away when they become isolated in a single BDD. When used in this manner, existential quantification (see Definition 2.3) does not change the satisfiability of the conjunction of BDDs. The process of interleaving existential quantification and BDD clustering is called Early Quantification [40, 74]. The first step in performing Early Quantification on a conjunction of BDDs φ is to create a quantification schedule (an ordering of the variables of φ). This schedule is used to guide clustering by specifying the order in which each variable will be quantified away. To follow a quantification schedule, first choose a variable x according to the schedule, conjoin all the BDDs that x occurs in, existentially quantify away x, and repeat. This type of clustering is typically referred to as either *Conjunction Scheduling* or *Quantification Scheduling* (mainly in the context of Image Computation). Constraints such as "if a BDD has more than 10 variables, do not consider it again during clustering" can also be levied during Conjunction Scheduling.

Many heuristics to choose good quantification schedules have been conceived over the years. Some such heuristics are DTREE [52, 53, 84], FINEGRAIN [89–91], FORCE [4], MINCE [2, 5], and VARSCORE [42]. Good heuristics are necessary to help avoid the exponential blowup in the number of BDD nodes while conjoining BDDs.

To give an example, one possible circuit-based SAT clustering heuristic is to cluster low-level gates, each represented as a BDD, generating a new set of complex gates (also BDDs), such that the

fanout of each gate-output equals one [86] (see Figure 3.1 on the following page for a step-by-step example). Clustering in this way produces a conjunction of BDDs where every non-input variable occurs in at most two BDDs, once as an input variable and once as an output variable.

BDD clustering can be further enhanced by interleaving specialized pairwise BDD operations between BDD conjunction steps. There are many different pairwise BDD simplification operations, most notably RESTRICT [48], CONSTRAIN [48], STRENGTHEN [141], LI-COMPACTION [80], among others (some are described in Section 2.3 on page 18). Their intent is to share partial information between pairs of BDDs, simplifying the conjunction of BDDs along the way. With state-based SAT in particular, the goal of clustering BDDs is to support search on SMURFS. Experience with SBSAT suggests that state-based SAT can be enhanced by preprocessing the instance into a small number of relatively compact and independent SMURFS, though this may not always be possible. However, there are a number of reasons clustering is essential for state-based SAT search, among them are the following:

1. Specialized heuristics can be developed that exploit the presence of compact and loosely coupled SMURFS.

2. Compact SMURFS can produce more useful conflict clauses (see Section 5.1.1 on page 68.

3. BCP is strengthened (inferences are discovered earlier).

4. BCP is more efficient. In fact, since clustering can result in a set of constraints that are drastically reduced in number and use less variables than the original CNF, BCP is sometimes much faster than using lazy data structures (such as watched pointers) on the original CNF. And, this speed up can be done without compromising on the ability to compute complex heuristics that require full information about the state of the instance under the current partial assignment.

3.2.2 Variable Elimination

Conjunction Scheduling can be performed using *Variable Elimination* (VE). VE originally of [56] and tailored to BDDs in [82,84] performs parameterized bucket-based BDD clustering according to

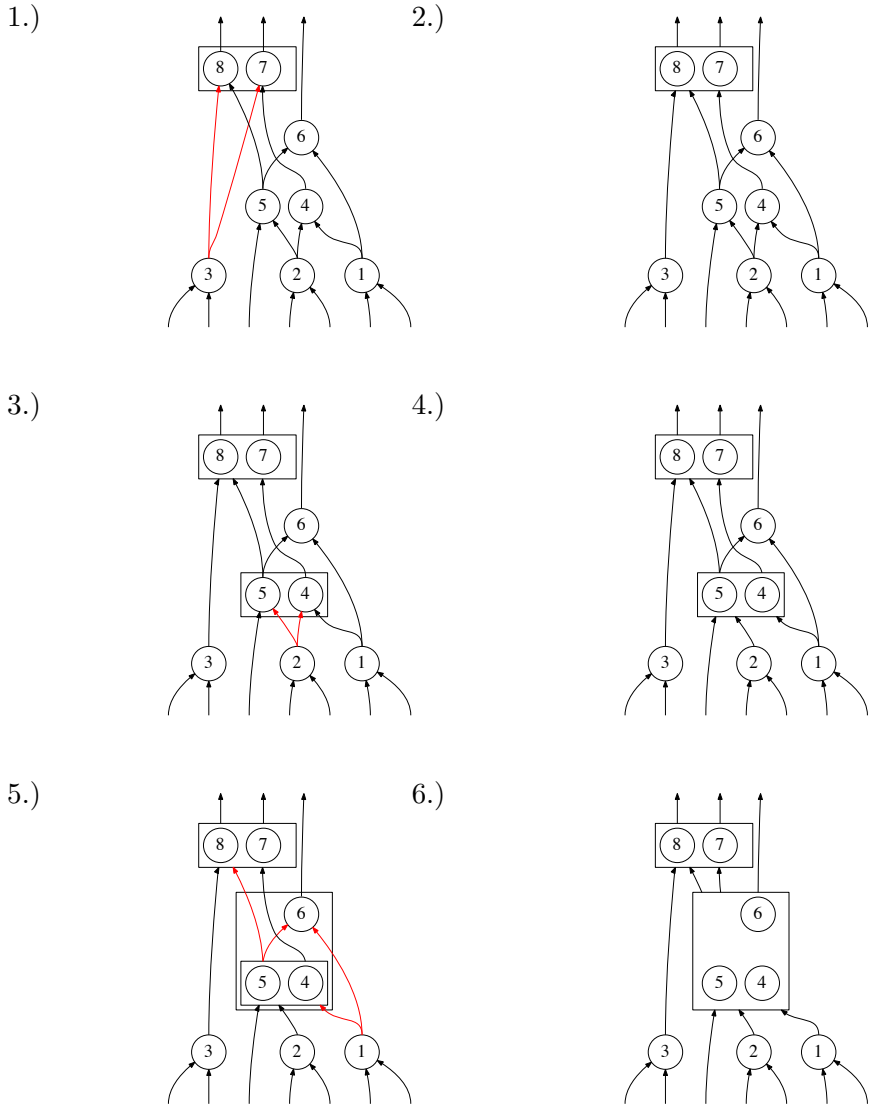

Figure 3.1 — An example illustrating one of the *many* possible clustering schedules for this specific And/Inverter Graph (AIG), a circuit constructed out of only `AND` and `NOT` gates. First, gates 7 and 8 are clustered, decreasing the output fanout for gate 3. Next, gates 4 and 5 are clustered, decreasing the output fanout for gate 2. Finally, node 6 is clustered into the $4, 5$ gate-cluster, decreasing the output fanout for both gate 1 and gate-cluster $4, 5$. Clustering stops at this point because every output has a fanout of one.

a given quantification schedule. In [93], VE was enhanced so that the order in which BDDs are clustered resembles a tree. This algorithm, named VE-TREE (Algorithm 3), was shown to outperform VE. To support state-based SAT (which can rely heavily on clustering as a preprocessing step) a new algorithm named VE-TREEITER, given in Algorithm 4, is introduced here that iteratively calls VE-TREE with an extra parameter: a limit on the number of new BDD nodes that can be created while conjoining BDDs. VE-TREEITER increases this limit with every call to VE-TREE. Using a small number of iterations allows the clustering process to proceed quickly, skipping over clustering BDDs that would cause the conjunction to explode, but still greatly reducing the number of both BDDs and variables by first removing the low-hanging fruit. Letting the number of iterations be infinite causes the clustering process to become complete, meaning the monolithic BDD will be built, solving the conjunction of BDDs. A parameterized version of VE-TREEITER has been implemented in SBSAT (and SBSAT is using the CUDD BDD package [130] to manage BDDs).

Heuristics to generate quantification schedules have also been implemented in SBSAT, namely STATIC, FORCE [4], VARSCORE [42], RANDOM, CUDD, OVERLAP, and OPTIMAL. The STATIC heuristic is the simplest heuristic (used to provide a baseline). It orders the variables according to their position in the original CNF input. The FORCE heuristic is "a fast and easy-to-implement variable-ordering heuristic" [4] that attempts to produce an ordering such that variables often occurring together in a constraint occur near each other in the ordering. This is done by iteratively refining the ordering such that the sum of the span of each BDD is reduced. Shown in Algorithm 5, the *span* of a BDD is the difference between the smallest and greatest variables in the BDD according to a given ordering. The VARSCORE heuristic chooses a variable ordering according to the sum of the number of nodes of the BDDs each variable occurs in, the less nodes the better. The RANDOM heuristic chooses a random ordering. The CUDD heuristic orders the variables to match CUDD's own internal BDD variable ordering. The OVERLAP heuristic orders the variables according to the number of BDDs each variable occurs in, less is better (this is similar to what was done in [143]). The OPTIMAL heuristic computes the total span [4] (the sum of the spans of every BDD) of each of the above heuristic orderings and uses the ordering with the lowest total span. Both FORCE and VARSCORE are the latest in a long line of BDD clustering heuristics. Surprisingly, they are both

Algorithm 3 VE-TREE(*BddManager*, CLUSTERHEURISTIC, *limit*)
CLUSTERBDDS places the conjunction of BDDs at positions b_0 and b_1 at b_0 and shifts remaining BDDs to fill the newly empty b_1 position

```
 1: order := ClusterHeuristic(BddManager)
 2: for k := 0 to NumberOfVariables(BddManager) do
 3:     v := order[k]
 4:     position := 0
 5:     while NumberOfBdds(BddManager, v) > 1 do
 6:         if (position + 1) ≥ NumberOfBdds(BddManager, v) then
 7:             position := 0
 8:         end if
 9:         b0 := BddPosition(BddManager, v, position)
10:         position := position + 1
11:         b1 := BddPosition(BddManager, v, position)
12:         exception := ClusterBdds(BddManager, b0, b1, limit)
13:         if exception = limit_reached then
14:             break
15:         else if exception = unsatisfiable then
16:             return unsatisfiable
17:         end if
18:     end while
19: end for
20: if NumberOfBdds(BddManager) ≤ 1 then
21:     return satisfiable
22: end if
23: return unknown
```

Algorithm 4 VE-TREEITER(*BddManager*, CLUSTERHEURISTIC, *iterations*)

```
1: limit := 0
2: for i := 0 to iterations do
3:     returnValue :=Ve-Tree(BddManager, ClusterHeuristic, limit)
4:     if returnValue ≠ unknown then
5:         return returnValue
6:     end if
7:     limit := 1 + (limit * 2)
8: end for
9: return unknown
```

very easy to implement compared to their predecessors.

Algorithm 5 SPAN(b, *order*)

```
1:  min := ∞
2:  max := 0
3:  if IsConstant(b) then
4:      return 0
5:  end if
6:  for each variable v_i in Support(b) do
7:      if order_{v_i} < min then
8:          min := order_{v_i}
9:      end if
10:     if order_{v_i} > max then
11:         max := order_{v_i}
12:     end if
13: end for
14: return max − min
```

3.2.3 Experimental Results

Table 3.4 on the next page shows results of VE-TREEITER (as implemented in SBSAT) solving benchmarks used in [93]. Specifically, SBSAT is configured to run 4 different ways. The structure recovery methods described in Section 3.1 on page 30 either are or are not used to reverse engineer Boolean constraints from the CNF. Also, VE-TREEITER is run with a high number of iterations and with either the STATIC heuristic or the FORCE heuristic. The results of two other solvers, Lingeling [25] (a recent competitive CDCL solver) and EBDDRES [26, 93] (a BDD-based solver configured to perform VE-TREE with the FORCE heuristic[1]) are also shown. Limits of 600 seconds and 2 GiB of RAM were used.

Many of the benchmarks used in Table 3.4 on the next page are known to be exponentially hard for resolution (this is hinted at in the results of Lingeling). Hence, they are good candidates for showing the usefulness of solvers that naturally make use of stronger proof systems, such as BDD-based solvers [9]. However, just because a solver *can* make use of a stronger proof system to solve a problem doesn't mean it will. For example, both EBDDRES and SBSAT (when configured to perform VE-TREEITER without recovering the garbled circuit structure) exhibit exponentially

[1] To increase performance, the trace generation feature of EBDDRES was disabled when collecting these results.

Table 3.4 — Results of this chapter's BDD clustering methods solving benchmarks of [93].

						SBSAT running VE-TREEITER							
						Without Gate Recovery				With Gate Recovery			
			Lingeling	EBDDRES		STATIC		FORCE		STATIC		FORCE	
Instance	Clauses	Vars	Time	Time	Mem	Time	Mem	Time	Mem	Time	Mem	Time	Mem
add4	60	157	0.00	0.00	1	0.03	27	0.03	27	0.03	27	0.03	27
add8	149	400	0.00	0.00	1	0.03	28	0.03	28	0.03	28	0.03	28
add16	330	895	0.02	0.01	2	0.04	29	0.04	29	0.04	28	0.04	28
add32	695	1894	0.05	0.05	7	0.05	31	0.07	32	0.04	30	0.05	30
add64	1428	3901	0.17	1.33	93	0.10	38	0.13	41	0.07	34	0.09	34
add128	2282	6586	0.24	40.34	-	1.14	206	3.11	371	0.52	107	0.40	101
fpga108	120	448	0.00	0.68	56	70.72	1248	5.27	220	0.72	59	0.36	55
fpga109	135	549	0.00	1.04	84	49.83	1228	84.78	792	0.92	67	0.50	60
fpga1211	198	968	0.00	13.85	896	-	1640	-	1732	8.90	423	7.85	441
mutcb3	8	19	0.00	0.00	1	0.03	27	0.03	27	0.03	27	0.03	27
mutcb4	20	56	0.00	0.00	1	0.03	27	0.03	27	0.03	27	0.03	27
mutcb5	36	107	0.00	0.00	1	0.03	27	0.03	27	0.03	27	0.03	27
mutcb6	56	172	0.00	0.00	1	0.03	27	0.03	27	0.04	27	0.03	27
mutcb7	80	251	0.01	0.00	1	0.03	28	0.03	28	0.04	27	0.03	27
mutcb8	108	344	0.01	0.00	1	0.03	28	0.03	28	0.04	28	0.04	28
mutcb9	140	451	0.09	0.01	2	0.05	30	0.06	30	0.04	28	0.04	28
mutcb10	176	572	0.17	0.02	3	0.08	32	0.05	30	0.05	29	0.04	28
mutcb11	216	707	0.90	0.04	7	0.06	32	0.09	33	0.06	31	0.08	31
mutcb12	260	856	2.10	0.09	13	0.15	37	0.08	31	0.11	33	0.07	32
mutcb13	308	1019	33.93	0.16	19	0.21	55	0.13	33	0.13	34	0.10	32
mutcb14	360	1196	71.97	0.33	37	0.22	57	0.31	57	0.14	36	0.13	34
mutcb15	416	1387	-	0.58	53	0.82	95	0.55	60	0.14	53	0.22	55
mutcb16	476	1592	-	1.25	106	0.32	62	0.16	53	0.22	57	0.11	34
mutcb17	540	1811	-	1.94	148	0.30	63	0.17	53	0.22	58	0.30	53
ph2	6	9	0.00	0.00	1	0.03	27	0.03	27	0.03	27	0.03	27
ph3	12	22	0.00	0.00	1	0.03	27	0.03	27	0.03	27	0.03	27
ph4	20	45	0.00	0.00	1	0.03	27	0.03	27	0.04	27	0.03	27
ph5	30	81	0.00	0.00	1	0.03	27	0.04	27	0.03	27	0.03	27
ph6	42	133	0.01	0.00	1	0.04	27	0.04	28	0.04	27	0.03	27
ph7	56	204	0.13	0.02	4	0.07	29	0.07	30	0.04	27	0.04	27
ph8	72	297	0.36	0.10	14	0.05	29	0.12	31	0.04	28	0.04	28
ph9	90	415	1.65	0.54	56	0.10	31	0.36	34	0.04	28	0.04	28
ph10	110	561	-	1.88	132	4.04	108	0.94	54	0.05	29	0.05	29
ph11	132	738	-	6.59	448	11.32	186	4.01	197	0.07	30	0.06	30
ph12	156	949	-	22.75	1344	336.92	610	18.33	198	0.10	31	0.07	31
ph13	182	1197	-	39.59	-	-	1141	93.97	566	0.13	31	0.12	31
ph14	210	1485	-	40.92	-	-	1564	-	1355	0.21	32	0.18	32
ph15	240	1816	-	43.06	-	-	1548	-	1938	0.82	33	0.14	33
ph16	272	2193	-	42.37	-	-	1673	-	1111	0.49	34	0.18	34
urq35	46	470	523.96	0.00	1	0.04	28	0.07	30	0.03	27	0.03	27
urq45	74	694	-	0.01	1	3.47	191	0.13	32	0.03	27	0.03	27
urq55	121	1210	-	0.01	1	585.41	-	0.24	37	0.03	27	0.03	27
urq65	180	1756	-	0.02	2	105.61	-	0.58	56	0.04	28	0.04	28
urq75	240	2194	-	0.03	3	254.62	-	60.85	750	0.04	28	0.04	28
urq85	327	3252	-	0.05	6	139.02	-	201.36	-	0.04	28	0.04	28

increasing solving times on some of these benchmark families, even though polynomial sized BDD-based proofs exist for those benchmark families. One way to coax the BDDs into these shorter proofs seems to be to make use of the user-domain information of the original instances. Such information is gained here by reverse engineering Boolean constraints from the CNF. Once this is done, VE-TREEITER naturally finds the short BDD proofs. This is evidenced in the final four columns of Table 3.4 on the preceding page. These four columns show that exploiting user-domain information can provide orders of magnitude improvement in solving time. Though, recovering user-domain information is not the only important process. As can be seen in Table 3.5 on the next page, both the clustering heuristic and inference discovery can have a large impact.

Table 3.5 on the following page shows results of this chapter's BDD clustering methods preprocessing benchmarks of [120] and [67]. Specifically, SBSAT recovers Boolean constraints from the CNF and runs only one iteration of VE-TREEITER with either the FORCE heuristic, the VARSCORE heuristic, or the VARSCORE heuristic combined with automatic inference and equivalence discovery and application. The inference and equivalence discovery and application that have been integrated into SBSAT are similar to what has been done in [64–66].

Finally, Table 3.6 on page 47 shows results of SBSAT using the clustering methods presented in the chapter to solve the unsatisfiable SGEN [133] benchmarks from the SAT-2009 competition [21]. Specifically, SBSAT was configured to recover Boolean constraints from the CNF and execute VE-TREEITER with no limit on the number of iterations. VE-TREEITER used the OVERLAP heuristic and the GROUP_SIFT variable reordering method of CUDD. Also shown are results of EBDDRES, PicoSAT, and Glucose. Both PicoSAT [24] and Glucose [11] are modern CDCL solvers. Limits of 15000 seconds and 16 gigabytes of RAM were used.

3.3 Summary

Pattern matching methods can be used to successfully reverse engineer some common Boolean constraints from CNF, generating a conjunction of BDDs that more closely resembles the constraints as they would be represented in the user-domain. Those BDDs can then be clustered using algorithms such as VE-TREEITER configured to use one of the latest BDD clustering heuristics such as FORCE

Table 3.5 — Results of SBSAT's BDD clustering methods preprocessing benchmarks of [120] and [67].

			Force			VarScore			VarScore + Infs		
Instance	Clauses	Vars	Time	BDDs	Vars	Time	BDDs	Vars	Time	BDDs	Vars
par8-1-c	254	64	0.01	0	0	0.01	0	0	0.02	0	0
par8-1	1149	350	0.04	14	65	0.02	11	47	0.03	0	0
par16-1-c	1264	317	0.02	24	74	0.04	11	37	0.04	11	37
par16-1	3310	1015	0.06	37	161	0.06	38	151	0.05	9	32
par32-1-c	5254	1315	0.08	56	178	0.06	49	165	0.06	49	164
par32-1	10277	3176	0.18	134	422	0.14	203	545	0.12	78	176
barrel5	5383	1407	0.14	250	507	0.06	252	343	0.07	253	343
barrel6	8931	2306	0.19	492	805	0.12	408	527	0.11	408	526
barrel7	13765	3523	0.29	652	950	0.18	695	758	0.19	716	758
barrel8	20083	5106	0.98	1004	1440	0.24	1031	1050	0.26	1037	1049
barrel9	36606	8903	1.25	2118	2049	0.62	2184	1667	0.56	2193	1670
ssa2670-130	3321	1359	0.11	42	248	0.08	15	76	0.04	9	68
ucsc-bf1355-348	7271	2286	0.29	147	564	0.20	313	553	0.10	157	381
dubois100	800	300	0.02	0	0	0.01	0	0	0.01	0	0
dlx1_c	1614	287	0.04	34	161	0.02	30	110	0.02	30	110
1dlx_c_mc...	3725	766	0.05	122	437	0.04	93	298	0.05	90	279
dlx2_cc_bug18	20208	2043	0.51	682	1421	0.34	684	1220	0.33	684	1220
2dlx_ca_mc...	24640	3186	0.24	933	1984	0.14	872	1632	0.16	873	1620
2dlx_cc_mc...	41704	4524	0.31	1665	3141	0.26	1572	2653	0.26	1554	2598
2dlx_cc...bug019	48232	4824	0.55	2409	3382	0.36	2345	2957	0.35	2362	2913
9vliw_bp_mc	179492	19148	2.19	6603	13163	1.10	6576	11115	1.19	6597	11113
c499	1870	606	0.04	42	181	0.02	18	105	0.03	16	102
3bitadd_32	32316	4480	1.42	14281	4480	0.74	16842	4480	0.67	16837	4480
x1.1_16	122	46	0.02	0	0	0.02	0	0	0.01	0	0
x2_128	1018	382	0.03	14	122	0.01	9	120	0.01	9	120
longmult12	18645	5974	0.61	274	1624	0.23	250	926	0.24	209	853
longmult14	22389	7176	0.69	335	1776	0.26	319	1214	0.28	279	1091
longmult15	24351	7807	1.08	359	1769	0.33	342	1341	0.38	323	1206
ibm...3-k95	272059	73525	12.24	5459	27534	4.72	4692	14296	4.94	4241	14031
ibm...23-k100	861175	207606	40.84	15271	59290	23.63	11547	35897	21.12	11399	33838
hanoi6	39666	4968	2.40	6738	3809	1.74	5199	2946	1.56	4989	2680
Mat26	2464	744	0.10	90	326	0.06	64	138	0.06	61	132
Mat317	85050	24435	6.27	3333	5969	3.52	2845	3630	3.75	2742	3631
linvrinv8	6337	1920	0.28	226	753	0.08	184	293	0.08	184	293
linvrinv9	9154	2754	0.44	268	980	0.11	270	395	0.12	270	395
equilarge_l5	18519	4478	0.37	292	540	0.15	338	499	0.12	341	506
pyhala...4-01	31795	9638	3.99	657	2396	2.29	453	1819	1.91	475	1678
clauses-2	272784	75527	30.40	12745	26860	14.13	9296	16917	7.37	7448	7369
clauses-4	1002957	267766	122.63	44850	100635	62.78	44307	83260	46.18	46237	32166
clauses-6	2623082	683995	481.81	169476	321958	310.97	163142	259036	206.94	164495	99148
clauses-8	5687554	1461771	923.84	283988	691547	873.81	428687	660173	894.57	216914	243802
clauses-10	8901946	2270929	2515.96	453311	1082341	2402.37	547964	1041582	3390.55	361123	400521

Table 3.6 — Results of the clustering methods presented in this chapter solving the unsatisfiable SGEN benchmarks.

Instance	Number Clauses	Number Variables	PicoSAT Time	Glucose Time	EBDDRES Time	SBSAT Time
sgen1-unsat-61-100	132	61	0.80	2.31	0.16	0.05
sgen1-unsat-73-100	156	73	5.70	24.44	1.63	0.08
sgen1-unsat-85-100	180	85	68.60	1009.82	6.12	0.14
sgen1-unsat-97-100	204	97	1971.50	4774.21	222.89	0.32
sgen1-unsat-103-100	220	105	10221.40	-	-	1.08
sgen1-unsat-109-100	228	109	-	-	247.32	2.72
sgen1-unsat-115-100	244	117	-	-	-	31.84
sgen1-unsat-121-100	252	121	-	-	132.23	3.14
sgen1-unsat-127-100	268	129	-	-	-	436.31
sgen1-unsat-133-100	276	133	-	-	-	3478.60
sgen1-unsat-139-100	292	141	-	-	-	5829.93
sgen1-unsat-145-100	300	145	-	-	-	201.64
sgen1-unsat-151-100	316	153	-	-	-	7732.34

or VARSCORE. The clustering process can be assisted by interleaving pairwise BDD reductions methods such as RESTRICT and inference discovery and application.

The processes introduced in this chapter can be used to solve families of problems that are intractable for resolution-based SAT solvers. However, the focus of this dissertation is in using this process as a preprocessing phase with the goal of creating expressive and compact SMURFs that can be exploited by heuristic and inference routines during state-based SAT search. Of course, it is difficult to really test how much these preprocessing methods assist state-based SAT search without a state-based SAT solver. In support of this, the state-based SAT search of SBSAT was enhanced. Chapter 4 on the following page discusses enhancements to the SMURF data structure and Chapter 5 on page 66 discusses state-based SAT solving. Both chapters present results showing that the preprocessing techniques presented in this chapter perform well in transforming CNF input into a collection of BDDs that are ripe for use with state-based SAT.

Chapter 4

State Machine Precomputation

The central component of state-based SAT is the SMURF (State Machine Used to Represent a Function). This chapter explores the limits of SMURF precomputation and compression. *Precomputation* makes information readily available during search. This is different than *preprocessing* (earlier discussed in Chapter 3 on page 30) which massages and streamlines constraints in support of precomputation. The methods presented in this section aim to make state-based SAT search more efficient through the use of special SMURFs, subgraph isomorphism detection, and lazy SMURF computation. All of these methods help state-based SAT solvers scale to industrial sized instances and enable the use of special purpose solvers, goals that were not previously being met.

Section 4.1 on the next page presents previously known special SMURF representations as well as introduces new special SMURF representations that exploit many different properties of Boolean functions. Special SMURFs are important because they are essentially compressed SMURF representations of common Boolean functions that have very large uncompressed (or *general*) SMURF representations. Instances whose general SMURFs could not even previously be built (because the general SMURFs are too memory intensive for today's computers) can now be solved by using special SMURFs. Section 4.2 on page 58 briefly discusses how general SMURFs and special SMURFs can be used together to see even more compression. Section 4.3 on page 58 introduces SMURF *normalization*, another kind of SMURF compression technique that decreases the size of the entire collection of SMURFs by increasing the amount of state sharing between SMURFs. A technique for relaxing

SMURF precomputation is introduced in Section 4.4 on page 61, namely constructing SMURFs on demand. Finally, results are presented showing the effectiveness of the new methods presented in this chapter.

4.1 Special SMURFs

This section presents variations on the SMURF data structure that support domain specific operations and efficient construction and search on SMURFs[1].

It is possible to compress the SMURF representation of certain Boolean functions while maintaining arc-consistency [71, 119] (i.e. maximal implicativity) and efficient heuristic computation. Compressed, or *special* SMURFs [66] (templates for certain types of constraints) can better aid search operations, such as inference detection and heuristic computation, than general SMURFs. Special SMURFs also help state-based SAT support specialized theory solvers because special SMURFs can be constructed to support constraints specific to theory solvers (see Section 5.2 on page 81 for more results on this topic).

It is often the case that a special SMURF will use orders of magnitude less space than a general SMURF. In fact, some special SMURFs are even more compact than their corresponding BDDs. There are many commonly used Boolean functions where a compact SMURF representation is unknown, but is likely to exist. Finding such functions and building special SMURF representations for them would enable state-based SAT solvers to solve problems that could not be solved otherwise, e.g. because the number of SMURF states needed exceeds the memory limit of today's computers. The special SMURFs shown in Figure 4.1 on page 51, Figure 4.3 on page 56, Figure 4.4 on page 57, Figure 4.5 on page 59, and Figure 4.6 on page 60 are graphical representations of simple Boolean functions encoded as both general and special SMURFs. These figures help to illustrate both the great complexity of general SMURFs and the benefits of special SMURFs. The SMURF visualizations used in this chapter were produced automatically by SBSAT and the BDD Visualizer. As such, the special SMURF visualizations provided here are generalizations of the underlying SMURF data structure. For example, `OR` SMURF visualizations display a subset of the variables as indices, whereas

[1] Background on the SMURF and its definition is given in a previous section, Section 2.4 on page 22

negation of variables is not displayed.

As reported in [66], one exploitable property of some Boolean functions that supports special SMURFs involves a type of symmetry that can be modeled with counters. For example, disjunction is associative, meaning that the functionality of a clause does not depend on the order in which its literals occur. A clause exhibits a type of symmetry similar to a symmetric Boolean function [145] but on a consistent permutation of *literals* (not variables) [108]. This symmetry allows a clause to be modeled as a state machine that, during search, maintains a count of the number of unset literals in the clause. If any literal is assigned the value `True` then the state machine transitions to `True`, otherwise the count is decremented. If the count becomes one, the lone unset literal is inferred. Figure 4.1 on the following page shows two examples where this type of symmetry exploitation considerably decreases the size of SMURFs. The next sections introduce both previously known and new special SMURFs.

4.1.1 `OR` SMURFs

The first special SMURF presented here represents a single clause of n literals (see the lower left of Figure 4.1 on the next page). This special SMURF is called the `OR` SMURF and was briefly introduced in [65, 66]. It represents an n literal clause and uses only $n+1$ SMURF states, whereas the general SMURF needs $2^n - n$ states[2]. This special SMURF becomes necessary when solving CNF problems with long clauses (say, clauses with more than 12 literals).

In support of efficient BCP, `OR` SMURFs have recently been refined to use 3 separate state types, namely, counter states, inference states, and the terminal state (the `True` state). A *counter state* is a state that cannot infer any literal and only acts to keep one or more counts of important state variables. For a clause, one count is kept, namely, a count of the current number of unset literals of the clause. There are two transitions out of the `OR` counter state. The first transitions to the `True` state and is followed when a literal in this special SMURF is inferred to `True`. The second transition is taken when a literal in this special SMURF is inferred to `False` and leads to either the

[2]It may seem odd to those readers familiar with CNF that a simple clause has an exponential representation. This is also the case in other domains, for example, clauses also have an exponential representation in Algebraic Normal Form (ANF), so called, Zhegalkin polynomials. Whereas those familiar with ANF may think it odd that XOR constraints (which are linear in ANF) have an exponential representation in CNF.

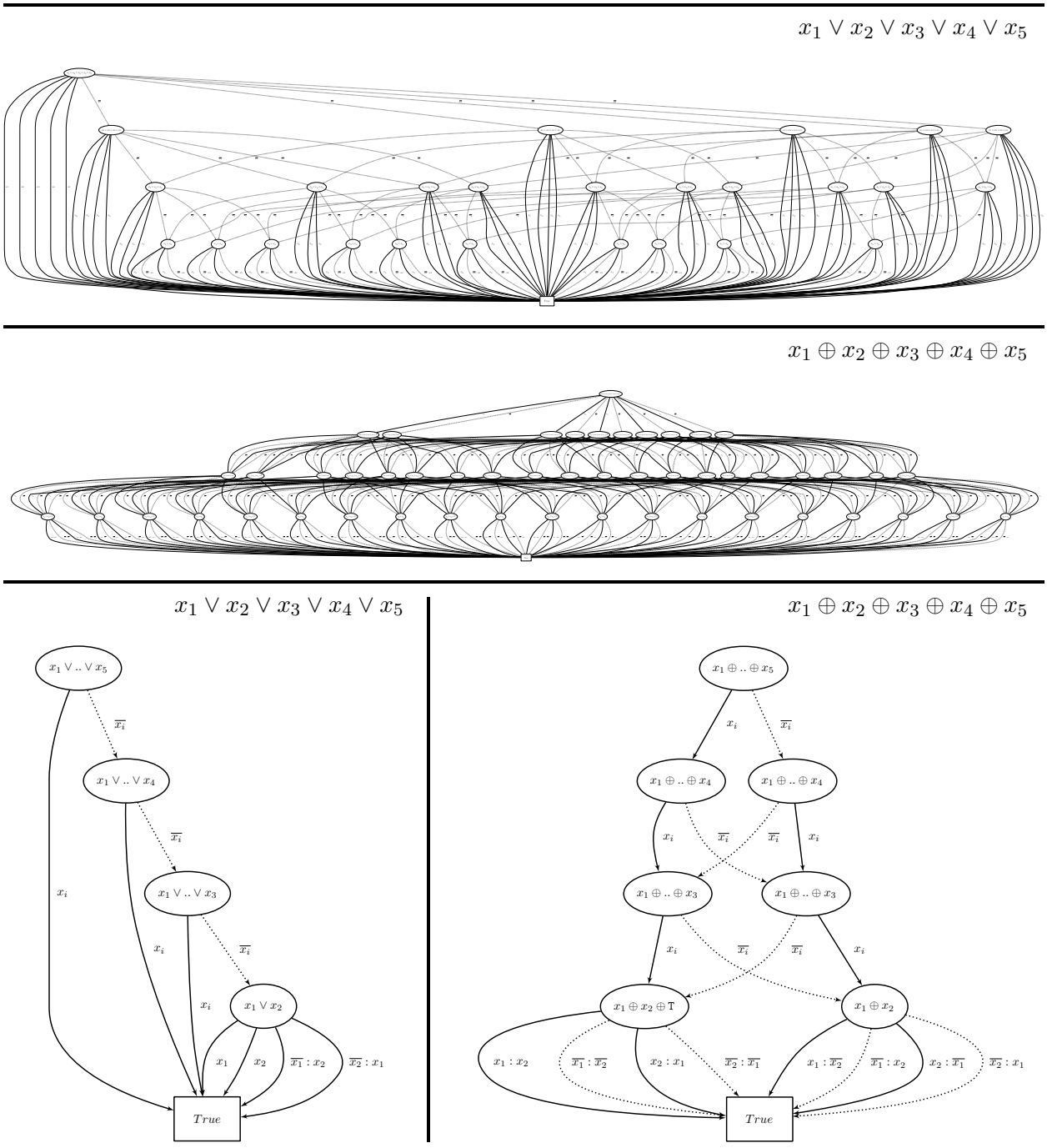

Figure 4.1 — This figure illustrates that a high level of SMURF compression is achievable by exploiting symmetries in some Boolean functions. The first two state machines are general SMURFs representing the Boolean functions $x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5$ and $x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5$, respectively. Also shown are the same functions represented by special SMURFs.

next counter state or the inference state. An *inference state* is one that can infer literals, and in the case of an `OR` SMURF there is only one inference state. When transitioning from an inference state, it may be necessary to perform a search to determine which literals need to be inferred. For `OR` SMURFs, this is a linear search though the list of literals of the associated clause to find the one remaining unset literal to infer.

This hints at the following: when inferences are memoized on general SMURF transitions, no computation is needed to deduce which inference to infer. With special SMURFs, some precomputation is foregone to save memory at the cost of more computation during search. This is a game of trading time for space. By relaxing precomputation, some amount of computation will be added during each node of the (potentially exponential) search tree, and vice versa.

This trade-off manifests itself in many SAT problem instances. One such example is the slider_100_unsat.cnf benchmark of [66]. The preprocessed form of this benchmark can make use of both general and special SMURFs. When solved using `SBSAT` with special SMURFs, total solver time is noticeably increased versus solving without using special SMURFs (27.6s versus 21.4s, with identical search trees). Likewise, solving the dlx1_c.cnf benchmark of [138] with special SMURFs takes noticeably less time than without (0.1s versus 6.0s). In the case of the first benchmark, precomputation both with and without special SMURFs takes very little time, but BCP is a little slower when using special SMURFs. In the case of the second benchmark, precomputation without special SMURFs takes the bulk of the solver time.

4.1.2 `XOR` SMURFs

The `XOR` SMURF is similar to the `OR` SMURF except that the `XOR` SMURF is composed of dual rails of counter states. An `XOR` constraint is either of the form $x_1 \oplus ... \oplus x_k$ or $x_1 \oplus ... \oplus x_k \oplus$ `T`, depending on whether or not it is negated. One rail of the `XOR` SMURF represents the negated form and the other represents the non-negated form. Setting a variable to `False` transitions the `XOR` SMURF from one rail to the other. Setting a variable to `True` keeps the SMURF on its current rail. Eventually, the SMURF will transition into one of two different inference states representing either the negated or non-negated form of the `XOR` constraint. See Figure 4.1 on the preceding page for a visualization

of this special SMURF.

4.1.3 `AND/OR` Gate SMURF

Both `AND` gates and `OR` gates can be modeled using a single special SMURF type because their functions are considered under arbitrary negation, i.e

$$x_0 \equiv (x_1 \wedge x_2) \quad \equiv \quad x_0 \oplus (\overline{x_1} \vee \overline{x_2})$$

Like the previous two special SMURFs, the `AND/OR` gate's counter states keep track of the number of unset literals, though only in the tail of the constraint (x_1 and x_2 in the function above). They also have two extra transitions which correspond to setting the head variable (x_0 in the function above). But, what's really fascinating about this special SMURF is that the counter states have transitions into other special SMURFs; both the previous special `OR` and `XOR` SMURFs. A visualization of the `AND/OR` gate SMURF is given in Figure 4.2 on the next page.

4.1.4 Cardinality SMURFs

Similar types of symmetries exist in Boolean functions other than those using `OR`s and `XOR`s. Introduced here are two such special SMURFs, namely cardinality constraints and negated cardinality constraints. Notoriously difficult to efficiently encode in CNF [13, 20, 63, 109], these special SMURFs have at most n^2 number of states where n is the number of variables of the constraint.

A *cardinality constraint* is a Boolean function of the form: $min \leq x_1 + .. + x_n \leq max$ where min and max are positive integers, $x_1..x_n$ are Boolean variables, and the constraint is satisfied if the number of Boolean variables taking the value `True` is greater than or equal to min and less than or equal to max. A *negated cardinality constraint* is the negation of a cardinality constraint. Figure 4.3 on page 56 and Figure 4.4 on page 57 show general and special SMURF representations of cardinality constraints and negated cardinality constraints.

Both types of cardinality SMURFs are composed of counter states that keep track of the number of unset variables and the number of variables that have been set to `True`. This is enough informa-

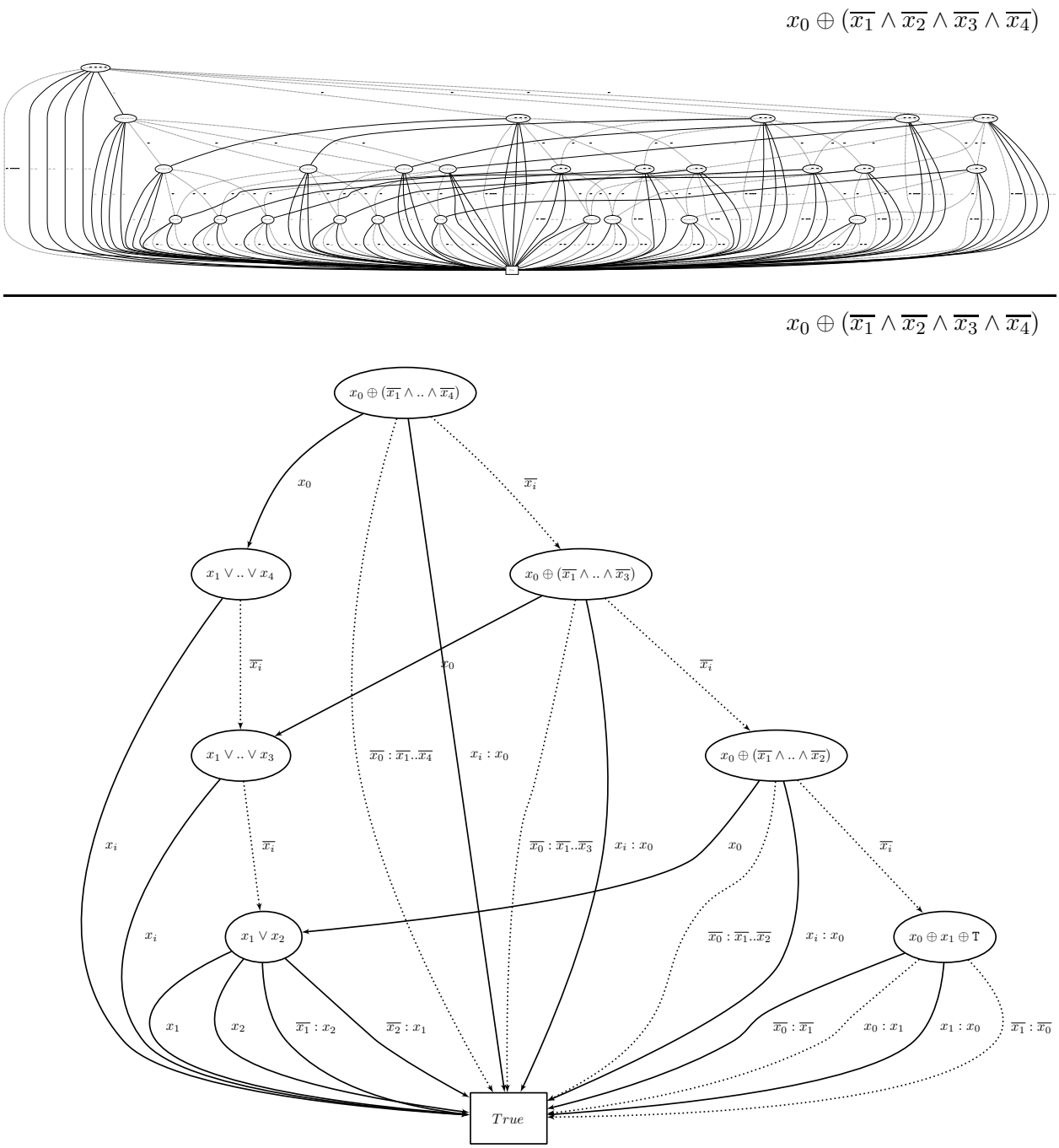

Figure 4.2 — Both the general SMURF and the special `AND` gate SMURF representing the function $x_0 \oplus (\overline{x_1} \wedge \overline{x_2} \wedge \overline{x_3} \wedge \overline{x_4})$.

tion, given the original *min* and *max* to determine whether or not any literals are inferred. The rules for determining inferences from these special SMURFs are given in Algorithm 6 and Algorithm 7.

Algorithm 6 TRANSITIONCARDINALITYSMURF(SMURF S_C, *variable*, *polarity*)

1: $min := \text{MAX}(0, S_C.min - S_C.numTrue)$
2: $max := S_C.max - S_C.numTrue$
3: **if** $polarity = \texttt{True} \wedge max = 1$ **then**
4: Infer remaining variables to `False`
5: **else if** $polarity = \texttt{False} \wedge S_C.numVariables - 1 = min$ **then**
6: Infer remaining variables to `True`
7: **end if**
8: **return** $polarity\ ?\ S_C.trueTransition : S_C.falseTransition$

Algorithm 7 TRANSITIONNEGATEDCARDINALITYSMURF(SMURF S_{NC}, *variable*, *polarity*)

1: $min := \text{MAX}(0, S_NC.min - S_NC.numTrue)$
2: $max := S_{NC}.max - S_{NC}.numTrue$
3: **if** $(polarity = \texttt{True} \wedge S_{NC}.numVariables \leq max \wedge min = 2) \vee$
4: $(polarity = \texttt{False} \wedge S_{NC}.numVariables = max + 1 \wedge min = 1)$ **then**
5: Infer remaining variables to `False`
6: **else if** $(polarity = \texttt{True} \wedge S_{NC}.numVariables = max + 1 \wedge min = 1) \vee$
7: $(polarity = \texttt{False} \wedge S_{NC}.numVariables = max + 2 \wedge min = 0)$ **then**
8: Infer remaining variables to `True`
9: **end if**
10: **return** $polarity\ ?\ S_{NC}.trueTransition : S_{NC}.falseTransition$

The sequence constraint [19,34] is a variant on the cardinality constraint that is satisfied if and only if a given length sequence of a list of variables all take the same value. Though not presented here, sequence constraints have a similar special SMURF encoding.

4.1.5 `XOR` Factors on SMURF Transitions

Introduced here is a technique that compresses SMURFs and supports the use of a special-purpose Gaussian elimination solver. Special SMURF transitions have been developed that memoize the parity (`XOR`) factors of Boolean functions fitting the form $(x_i \oplus .. \oplus x_j) \wedge F(x_1..x_n)$, where F is some non-linear Boolean function (see Figure 4.5 on page 59 for a visualization). These special transitions can be used to infer `XOR` functions, meaning that `XOR` functions can be factored out of SMURF states early, reducing the complexity of SMURFs with `XOR` components and supporting the

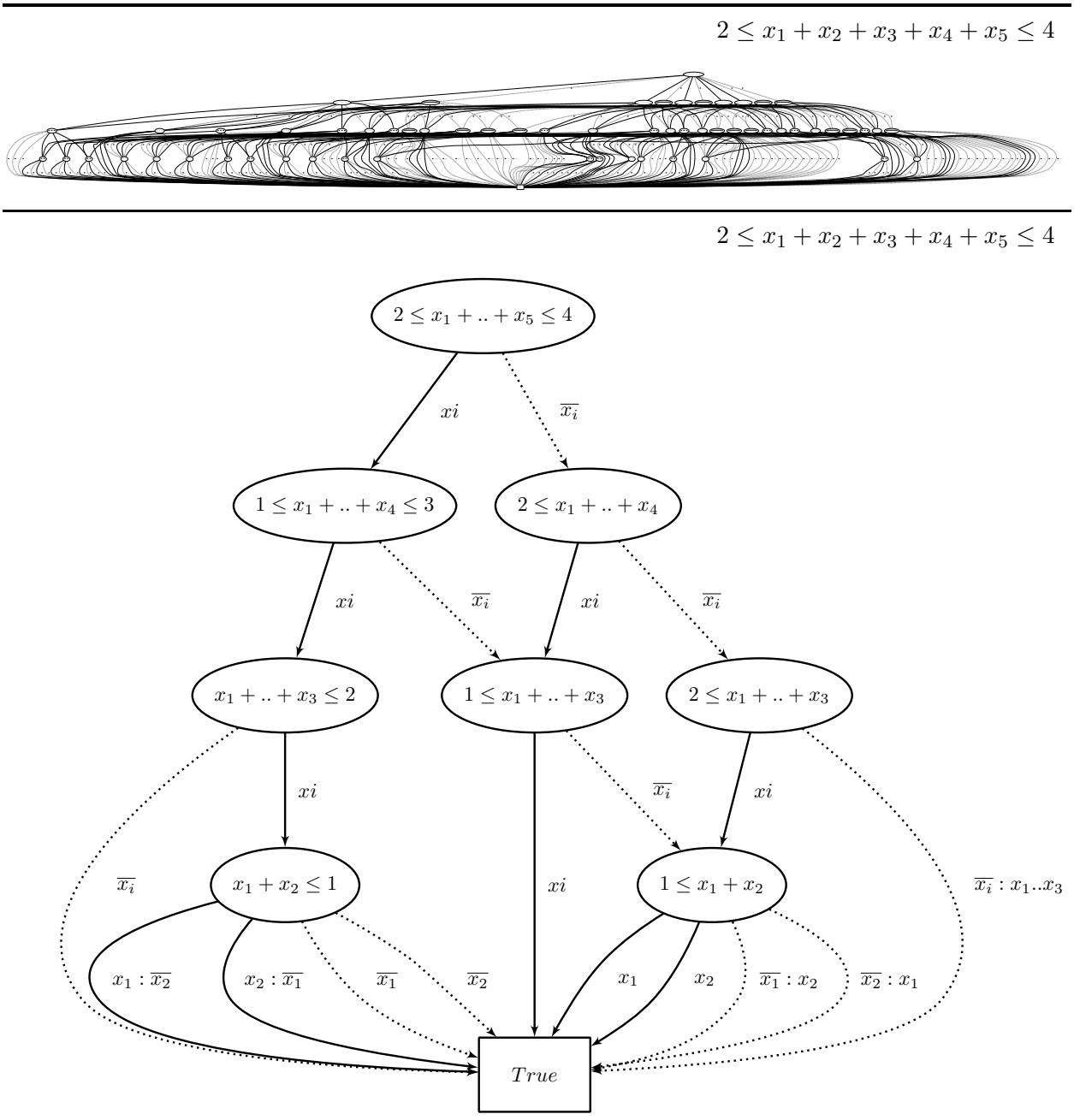

Figure 4.3 — Both the general SMURF and the special cardinality-based SMURF representing the function $2 \leq x_1 + x_2 + x_3 + x_4 + x_5 \leq 4$.

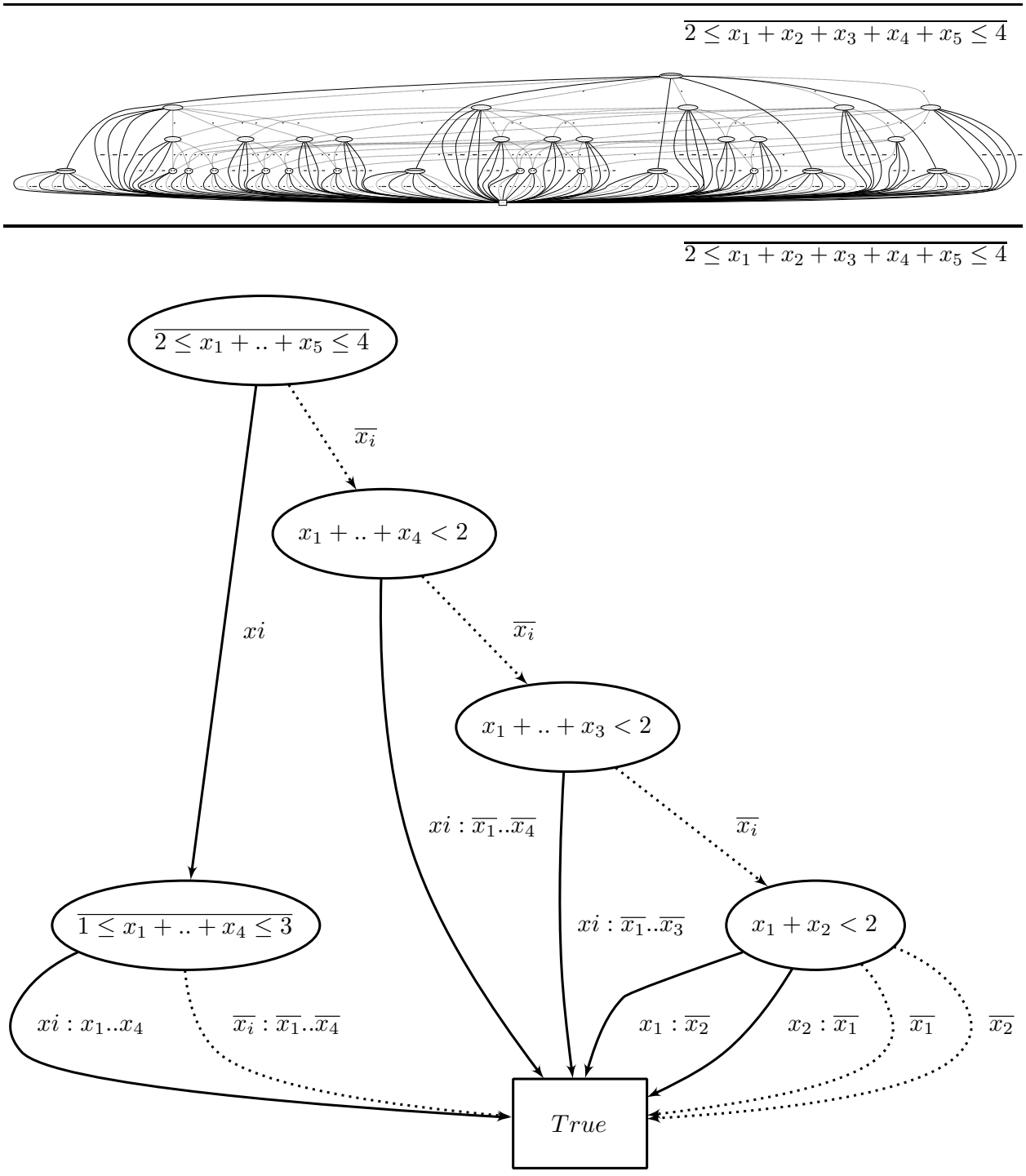

Figure 4.4 — Both the general SMURF and the special cardinality-based SMURF representing the function $\overline{2 \leq x_1 + x_2 + x_3 + x_4 + x_5 \leq 4}$.

use of a special-purpose Gaussian elimination solver. See Section 5.2 on page 81 for more on this special-purpose solver and how Boolean factors are discovered and safely factored out of Boolean functions.

4.2 General to Special SMURF Transitions

Another SMURF compression technique involves allowing general SMURFs to transition into special SMURFs. The special SMURFs from [66] can be made to act like the leaf states of a general SMURF, i.e. a general SMURF may transition into a special SMURF, but there are not currently any special SMURFs that transition into a general SMURF. For example, the function $x_1 \vee (1 \leq x_2 + x_3 + x_4 \leq 2)$ (visualized in Figure 4.6 on page 60) is more compact when built using special SMURFs as leaf nodes.

4.3 SMURF Normalization

SMURF normalization, another kind of SMURF compression technique, decreases the size of the entire collection of SMURFs by increasing the amount of state sharing between SMURFs. The idea is this: instead of building a SMURF for a BDD b with n variables, the SMURF is built for a BDD b' that contains variables $1, .., n$, is isomorphic to b, and, when its variables are interpreted as indices into the variable list of b, is functionally equivalent to b. Algorithm 8 shows how to normalize a BDD prior to building its SMURF. Normalizing every SMURF greatly increases SMURF state sharing across an entire collection of SMURFs because many SMURF states are now identical that before were only isomorphic. However, the variables of normalized SMURFs have to be interpreted as indices into the original variable list. This makes the implementation more complicated but, in terms of computational complexity, has a negligible effect. Even more sharing is possible if normalization is performed before the construction of every SMURF state. However, during search, it is cumbersome to interpret variables of this kind of SMURF.

Subgraph isomorphism in the context of BDDs has been studied a bit before. One approach to detecting some isomorphic BDD nodes is described in [7] where Differential BDDs are introduced. Also, ACL2's OBDD implementation [32] is geared toward isomorphic subgraph sharing because

Figure 4.5 — The SMURF representing the function $x_1 \oplus ((x_2 \oplus x_3) \wedge (x_3 \vee x_4))$ is shown here without and with `XOR` factors memoized on SMURF transitions. Though there are a lot of `XOR` factors shown here, the $\overline{x_1}$ transition contains the most interesting one because $\overline{x_1}$ is the only transition with an `XOR` factor that transitions into a general SMURF . This transition demonstrates the real power of factoring to remove factors early and simplify the SMURF.

Figure 4.6 — Shown here is both a general SMURF and a SMURF whose general SMURF states transition into special SMURF states. Both SMURFs represent the function $x_1 \vee (1 \leq x_2 + x_3 + x_4 \leq 2)$.

Algorithm 8 CreateNormalizedSmurf(BDD b)
The BddPermute operation remaps the variables in a BDD and is available in BDD packages such as CUDD.

1: b_{norm} :=BddPermute(b,Support(b), $[1, .., |\text{Support}(b)|]$)
2: **return** CreateSmurf(b_{norm})

the index of a BDD node is interpreted based on its depth from the root, rather than stored directly at the node.

4.4 Smurfs on Demand

One way to relax some of the precomputation that's done when building a Smurf is to construct it on demand, or *lazily*. At least for current applications, the only information Smurfs provide that is absolutely needed prior to beginning search is heuristic information. When using a common CDCL heuristic, such as VSIDS [117], Smurf precomputation does not directly contribute to the heuristic. In this case, Smurfs can be built lazily, i.e. Smurf nodes are built upon being transitioned into during search and may be deleted after the search has backtracked over them. This technique can be very profitable because Smurf precomputation is often expensive and many practical SAT instances only require a very small space to be searched. Hence it is likely that not every possible state of every individual Smurf needs to be explored to determine the satisfiability of an instance.

This technique is also helpful when preprocessing techniques inadvertently create large Smurfs. For example, the Smurfs for two of the benchmarks used in Table 4.1 on page 64 and Table 4.2 on page 65, `2dlx_ca_mc_ex_bp_f.cnf` and `2dlx_cc_mc_ex_bp_f.cnf` were not able to be built within the memory limit of 16 GiB. This is because CNF pattern matching may have reverse engineered some long `AND/OR` gates and early quantification existentially quantified away some of their variables. `AND/OR` gate constraints do not cause any memory problems during precomputation because a special `AND` gate Smurf can be used (see Section 4.1.3 on page 53). However, if existential quantification is turned on during preprocessing and one of the variables in the tail of such a constraint is quantified away, the constraint will no longer match any special Smurf type (see Figure 4.7 on the following page). If the constraint has a lot of variables, it will take significant time to precom-

pute the general SMURF, so much so that this time may dominate the precomputation process (as it did for the two benchmarks mentioned above). In cases like this, where preprocessing methods inadvertently work against each other, precomputation can be performed very quickly if SMURFs with too many variables are computed lazily and the rest are precomputed. When this technique was applied to the `2dlx_ca_mc_ex_bp_f.cnf` and `2dlx_cc_mc_ex_bp_f.cnf`, with a limit of twelve variables, precomputation took very little time.

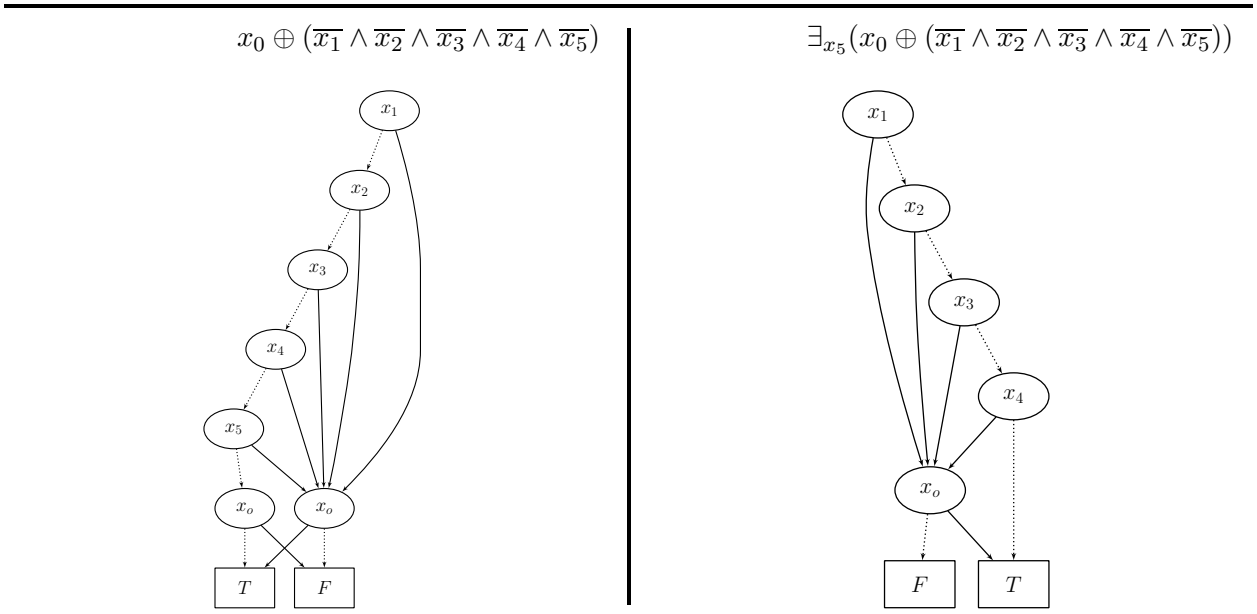

Figure 4.7 — This figure shows that preprocessing methods can sometimes be harmful to SMURF precomputation. Specifically, this figure shows that a function that would be precomputed using a special SMURF must instead be precomputed using a general SMURF because early quantification applied to the `AND` gate BDD on the left changes its function type to that of the BDD on the right that has no corresponding special SMURF.

4.5 Experimental Results

This section provides results demonstrating the power of the SMURF precomputation techniques introduced in this chapter. All of the new techniques and special SMURFs have been implemented in SBSAT, the tool used to collect the results shown in the two tables below. Both table show results on the benchmarks used in the previous chapter. The benchmarks come from a wide variety of industrial domains such as verification, equivalence checking, and bounded model checking and

also include benchmarks from the crafted categories of previous SAT competitions [21]. The first table, (Table 4.1 on the next page), shows the time spent creating SMURFs and the total number of SMURF states needed to fully represent each benchmark as a collection of SMURFs. Each benchmark was preprocessed by first recovering Boolean gates from the CNF and then running VE-TREEITER to completion with the VARSCORE heuristic and a limit of no more than ten variables per conjoined BDD. Also, existential quantification and inference propagation were used. The "-" symbol indicates that physical memory (16 GiB) was exhausted. Four different configurations are used. The first configuration uses only general SMURFs. The second allows special SMURFs to be transitioned into from general SMURFs (described in Section 4.2 on page 58). The third configuration uses SMURF normalization (described in Section 4.3 on page 58). The fourth configuration allows both general SMURF to special SMURF transitions and SMURF normalization. This first table shows that the techniques introduced in this chapter can give a large improvement on what was previously possible, along with providing confidence that these methods enable state-based SAT to scale to industrial problems.

The second table, Table 4.2 on page 65, shows the numbers of general and special SMURF states used to build the collection of SMURFs presented in Table 4.1 on the next page. The columns labeled "`CC`" and "`NCC`" give the number of special cardinality constraint and special negated cardinality constraint states, respectively (described in Section 4.1.4 on page 53). The column labeled "Inference" gives the number of inferences memoized on SMURF transitions. The column labeled "XFACT" gives the number of `XOR` factors memoized on SMURF transitions (described in Section 4.1.5 on page 55). This table demonstrates that every special SMURF type is used, and, given that each special SMURF type has an exponential general SMURF representation, necessary to represent most problems as a collection of SMURFs. Even with the support of special SMURFs, many general SMURF states are still being built. This hints that more special SMURFs types could be built, reducing the need for general SMURFs. This table also provides perspective, showing that any state-based SAT solver needs the techniques and special SMURFs introduced in this chapter to scale to industrial sized problems with millions of variables and constraints.

Table 4.1 — Statistics of SBSAT building general SMURFs, special SMURFs, normalized SMURFs, and normalized special SMURFs for benchmarks of [120] and [67].

			General		Special		Normalized		Norm.+Spec.	
Instance	Clauses	Vars	Time	States	Time	States	Time	States	Time	States
par8-1-c	254	64	0.00	0	0.00	0	0.00	0	0.00	0
par8-1	1149	350	0.00	0	0.00	0	0.00	0	0.00	0
par16-1-c	1264	317	0.84	77848	0.17	42309	0.08	18605	0.05	9851
par16-1	3310	1015	2.28	80841	0.11	6684	0.15	29522	0.08	5664
par32-1-c	5254	1315	10.00	301723	1.18	97366	0.32	74995	0.20	30588
par32-1	10277	3176	23.76	287511	1.83	39201	0.28	70629	0.17	19222
barrel5	5383	1407	-	-	7.68	256728	-	-	0.19	22037
barrel6	8931	2306	-	-	16.73	428125	-	-	0.17	22439
barrel7	13765	3523	-	-	35.61	652198	-	-	0.16	21019
barrel8	20083	5106	-	-	-	-	-	-	0.23	22779
barrel9	36606	8903	-	-	-	-	-	-	0.48	62356
ssa2670-130	3321	1359	0.97	44246	0.77	38688	0.17	40480	0.23	36131
ucsc-bf1355-348	7271	2286	14.26	308529	11.01	273624	0.52	150177	0.76	110445
dubois100	800	300	0.00	0	0.00	0	0.00	0	0.00	0
dlx1_c	1614	287	21.20	821878	0.52	73808	6.42	767668	0.41	60149
1dlx_c_mc...	3725	766	-	-	2.33	178278	173.98	14720645	0.74	137371
dlx2_cc_bug18	20208	2043	-	-	-	-	-	-	5.28	707682
2dlx_ca_mc...	24640	3186	-	-	-	-	-	-	-	-
2dlx_cc_mc...	41704	4524	-	-	-	-	-	-	-	-
2dlx_cc...bug019	48232	4824	-	-	-	-	-	-	10.12	928851
9vliw_bp_mc	179492	19148	-	-	-	-	-	-	63.12	3424878
c499	1870	606	-	-	0.46	56392	-	-	0.25	28773
3bitadd_32	32316	4480	-	-	-	-	-	-	0.14	8683
x1.1_16	122	46	0.00	0	0.00	0	0.00	0	0.00	0
x2_128	1018	382	0.95	49624	0.00	532	0.00	2046	0.00	124
longmult12	18645	5974	-	-	-	-	7.78	603607	1.30	137309
longmult14	22389	7176	-	-	-	-	5.77	607969	1.34	139745
longmult15	24351	7807	-	-	-	-	5.90	623923	3.13	155186
ibm...3-k95	272059	73525	-	-	-	-	2.41	380861	2.95	334358
ibm...23-k100	861175	207606	-	-	-	-	-	-	7.16	784307
hanoi6	39666	4968	-	-	-	-	6.06	1230676	9.54	1211366
Mat26	2464	744	5.10	162849	4.88	264263	0.45	108292	1.75	104338
Mat317	85050	24435	-	-	-	-	5.58	1029325	11.08	948782
linvrinv8	6337	1920	-	-	16.66	575573	-	-	1.38	133600
linvrinv9	9154	2754	-	-	24.33	868700	-	-	3.97	183057
equilarge_l5	18519	4478	-	-	3.32	201832	0.38	101568	0.27	32977
pyhala...4-01	31795	9638	-	-	-	-	11.40	2174858	30.72	1983185
clauses-2	272784	75527	-	-	-	-	-	-	5.12	485178
clauses-4	1002957	267766	-	-	-	-	-	-	8.93	588432
clauses-6	2623082	683995	-	-	-	-	-	-	16.45	667006
clauses-8	5687554	1461771	-	-	-	-	-	-	22.64	832715
clauses-10	8901946	2270929	-	-	-	-	-	-	24.19	911094

Table 4.2 — A table showing the number of each special SMURF type that was to build the collection of SMURFs for the benchmarks of [120] and [67].

Instance	Clauses	Vars	OR	XOR	∧/∨ Gate	CC	NCC	Inference	XFACT	General
par8-1-c	254	64	0	0	0	0	0	0	0	0
par8-1	1149	350	0	0	0	0	0	0	0	0
par16-1-c	1264	317	213	3168	192	635	0	2039	513	3092
par16-1	3310	1015	206	893	168	340	2	2023	618	1415
par32-1-c	5254	1315	386	4571	926	1564	152	8633	5296	9061
par32-1	10277	3176	735	1197	658	305	48	6975	3371	5934
barrel5	5383	1407	2862	230	3878	0	10	3793	142	11123
barrel6	8931	2306	2653	257	4308	0	12	4153	167	10890
barrel7	13765	3523	3293	203	4048	0	12	3009	114	10341
barrel8	20083	5106	3088	244	4080	0	12	4073	155	11128
barrel9	36606	8903	6522	143	11082	0	16	15665	147	28782
ssa2670-130	3321	1359	1870	230	2960	12	53	15354	1985	13668
ucsc-bf1355-348	7271	2286	19296	3719	723	16	42	29995	9196	47459
dubois100	800	300	0	0	0	0	0	0	0	0
dlx1_c	1614	287	5290	169	11829	30	0	20554	5065	17213
1dlx_c_mc...	3725	766	10009	296	19129	23	16	41618	11022	55259
dlx2_cc_bug18	20208	2043	129529	92	755	19	30	84708	7842	484708
2dlx_ca_mc...	24640	3186	-	-	-	-	-	-	-	-
2dlx_cc_mc...	41704	4524	-	-	-	-	-	-	-	-
2dlx_cc...bug019	48232	4824	55751	464	95664	72	638	219221	159651	397391
9vliw_bp_mc	179492	19148	79377	454	115637	48	127	1387076	51267	1790893
c499	1870	606	660	4820	3115	57	69	9430	4333	6290
3bitadd_32	32316	4480	1954	0	0	0	0	952	0	5778
x1.1_16	122	46	0	0	0	0	0	0	0	0
x2_128	1018	382	0	124	0	0	0	0	0	1
longmult12	18645	5974	1519	3338	5855	689	568	56414	20425	48502
longmult14	22389	7176	1436	3282	5366	691	523	55781	22540	50127
longmult15	24351	7807	1318	3182	5290	765	541	64307	26188	53596
ibm...3-k95	272059	73525	11846	1230	36911	164	198	99689	35413	148908
ibm...23-k100	861175	207606	31451	1597	95675	498	428	255110	79741	319808
hanoi6	39666	4968	18070	1241	54255	699	168	277402	85145	774387
Mat26	2464	744	175	9759	4546	0	0	20476	5364	64019
Mat317	85050	24435	180	14244	5634	0	134	154593	122609	651389
linvrinv8	6337	1920	224	12266	4710	0	14	11639	8568	96180
linvrinv9	9154	2754	250	13035	5276	0	30	14774	13266	136427
equilarge_l5	18519	4478	179	5216	824	1094	20	10331	7566	7748
pyhala...4-01	31795	9638	324	10718	5902	5848	1550	598670	708770	651404
clauses-2	272784	75527	29441	2630	36590	1982	208	162448	50199	201681
clauses-4	1002957	267766	21443	3663	25608	8185	600	212442	68312	248180
clauses-6	2623082	683995	20209	3528	22469	3977	467	239794	82852	293711
clauses-8	5687554	1461771	22172	8904	25102	5051	668	308526	109507	352786
clauses-10	8901946	2270929	19872	5254	26513	5947	562	345504	125105	382338

Chapter 5

State-based SAT Search

The previous chapters have laid the foundation for state-based SAT search by introducing methods for preprocessing low-level input and precomputing state-based SAT search data structures. Chapter 3 on page 30 introduced a BDD-based preprocessing methodology that reverse engineers common Boolean gates from CNF, generating a conjunction of BDDs which can be clustered using the Ve-TreeIter method working in tandem with a few of the latest BDD clustering heuristics such as Force and VarScore. Inference discovery and pairwise BDD operations are interleaved with clustering to further simplify the conjunction of BDDs. The goal of the preprocessing phase is to significantly reduce the number of variables and constraints of a given problem while eliciting any user-domain structure. Chapter 4 on page 48 showed how to transform the preprocessed conjunction of BDDs into a collection of highly compressed Smurfs using precomputation, special parameterized Smurfs, Smurf normalization, and Boolean factorization methods. This chapter introduces methods for performing intelligent and efficient search on Smurfs. Also, results showing the effects of these search techniques and techniques presented in previous chapters are given.

5.1 Support for CDCL Techniques

Recent CDCL techniques such as watched pointers [69, 106, 117], conflict clause minimization [18, 70], Unique Implication Points (UIPs) [111, 117], conflict clause memory management [10], and dynamic restarts [83, 94] have quickly become necessary components of competitive CDCL solvers.

Fortunately, each of these recent techniques can be supported by state-based SAT search. This is achieved by encapsulating the core components of state-based SAT search in a library providing a tailored set of search routines that can be used by extensible CDCL solvers.

The source code of many CDCL solvers is freely available. Some are well documented and extensible, meaning, they were developed with the methodology that they would be used as components (or libraries) in larger projects. This means that anyone developing a state-based SAT (or other higher domain solver) can easily make use of existing CDCL techniques by plugging into one (or more) of the many extensible CDCL solver frameworks. With state-based SAT (and SBSAT in particular) this task is relatively simple. Five hooks are needed, four of which communicate information about the CDCL solver state to the state-based SAT solver, and one which communicates SMURF inferences and witness clauses to the CDCL solver. Figure 5.1 provides a diagram showing the communication between the two types of solvers.

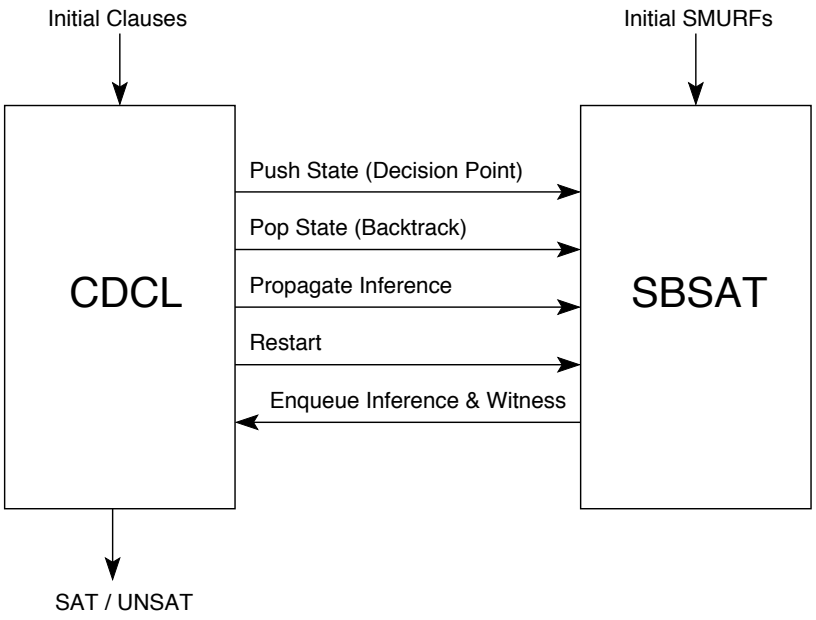

Figure 5.1 — Communication between an extensible CDCL solver and a state-based SAT solver.

The information passed from the CDCL solver to the state-based SAT solver is trivial, really just a series of commands denoting where the solver is in the search tree. The most complex pieces

of information are the witness clauses generated from SMURFs that are passed from the state-based SAT solver to the CDCL solver. Once the CDCL solver receives a witness clause as a reason for an inference it will automatically take care of conflict clause analysis, minimization, heuristic updates, and so on. The next section describes in detail the processes for generating witness clauses from SMURFs.

5.1.1 State-based SAT Learning

This section describes efficient methods of learning new information (such as conflict clauses) from SMURFs. The most successful industrial-strength SAT solvers learn new clauses during search. Whenever the solver reaches a conflict, resolution is used to produce a new *conflict clause* that is added to the formula to avoid that same conflict in the future. Recent SAT solver research has focused heavily on efficient generation and use of conflict clauses because, for certain problem domains such as verification, the use of conflict clauses has been shown to produce orders of magnitude improvement in solving time [18, 110, 111, 125].

With state-based SAT, many choices exist regarding learning. For example, resolving on a variable x in clauses c_i and c_j produces the resolvent c_r where $c_r \equiv \exists_x(c_i \wedge c_j)$. When a state-based SAT solver reaches a conflict, SMURF-based resolvents could be computed in the same way, adding new conflict SMURFs to the current collection, however, this approach is not explored here. State-based SAT supports standard CDCL conflict clause generation and all currently known CDCL conflict clause techniques. The main contributions of state-based SAT solving are the witness clauses generated by SMURFs. *Witness clauses* (sometimes called reason clauses) are those clauses that have become unit during BCP and hence are asserting a literal. During conflict analysis, witness clauses are resolved together at UIPs to generate conflict clauses [111, 117].

In the case of state-based SAT search, witness clauses are generated from SMURFs during BCP. Conflict clause analysis and learned clause maintenance can then work in the same manner as that of a standard CDCL solver. The approach taken here was to hook a state-based SAT solver into a CDCL solver, letting the CDCL solver manage all conflict clause analysis and maintenance (the communication needed is shown in Figure 5.1 on the previous page).

A SMURF witness clause for an inference i generated by a SMURF S's transition t must consist of i and at most the negation of the non-inferred literals on the path from the root of S to t that are consistent with the current partial assignment. There may be many such paths, and hence many different yet correct witness clauses for a particular inference. However, since inferences are handled in turn during BCP, there is always a *minimal witness clause*, i.e. one that contains i and consists of the negation of the non-inferred literals of the shortest path from the root of S to any state that directly implies i (not necessarily the current state) and is consistent with the *current* partial assignment. Consider the following Boolean function:

$$(\overline{x_1} \vee x_3) \wedge (\overline{x_2} \vee x_3)$$

Both assignments $x_1 \mapsto \texttt{True}$ and $x_2 \mapsto \texttt{True}$ infer x_3. Since SMURFs are arc-consistent with respect to the function they were generated from, and since BCP handles one inference at a time, only one of the witness clauses $(\overline{x_1} \vee x_3)$ or $(\overline{x_2} \vee x_3)$ can be minimal with respect to the current partial search assignment, i.e. when the witness clause for x_3 is generated, only one of $x_1 \mapsto \texttt{True}$ or $x_2 \mapsto \texttt{True}$ will have been removed from the inference queue, added to the current partial assignment, and propagated by BCP.

Recent SAT research has focused on generating new clauses and minimizing them during search [18, 70]. Witness clauses produced by SMURFs can provide added support to conflict clause generation and minimization. A greedy method for generating witness clauses from SMURFs is used in the tools presented in [65, 66], however, the witness clauses are not guaranteed to be minimal. To see how to generate minimal witness clauses from SMURFs, it will help to first explore previous research on generating minimal witness clauses from BDDs.

In [51], generating witness clauses from BDDs is briefly mentioned. They claim to use a greedy method that would likely produce the same clauses as the method used in [65, 66], though, since details are not provided, this is only a guess. In [78], an algorithm is given that generates the minimal witness clause M from a BDD G and an unminimized witness clause P. P is the conjunction of the literals of the witness clause except for the literal being inferred, l. Prior to executing the algorithm,

G is transformed into $\exists_{\vec{x}} G|_{\overline{l}}$, where $\vec{x}$ is the support of G not in P. A slightly enhanced version of their algorithm is presented here in Algorithm 9. The main difference is that the algorithm of [78] has the precondition that G only contain variables in P, requiring all variables in G but not in P to be existentially quantified away from G prior to executing the algorithm. Algorithm 9 relaxes this precondition by interleaving existential quantification between recursive calls at line 12.

Algorithm 9 BddMininf(G, P, M)
$G \wedge P \equiv$ `False`
P is the conjunction of literals representing an initial witness clause for literal l

```
 1: if G = False then return M
 2: G := Ite(v_g, G_{v_g}, G_{¬v_g})
 3: P := Ite(v_p, P_{v_p}, P_{¬v_p})
 4: if P_{v_p} = False then l := ¬v_p else l := v_p
 5: if v_g = v_p then
 6:     if IteConstant(P_l, G_{¬l}, False) = False then
 7:         return BddMininf(G_l, P_l, M ∪ l)
 8:     else
 9:         return BddMininf(∃_{v_g} G, P_l, M)
10:     end if
11: else if BddVarOrder(v_g) < BddVarOrder(v_p) then
12:     return BddMininf(∃_{v_g} G, P, M)
13: else
14:     return BddMininf(G, P_l, M)
15: end if
```

Existential quantification is an expensive BDD operation and the optimization presented here only goes so far in reducing its cost. It is possible to remove the existential quantification operation on line 12 by adding a precondition that the BDD variable ordering should put all variables in the support of G but not in P nearest the leaves of the BDDs. However, BDD reordering also is expensive and the existential quantification operation at line 9 cannot be removed by variable reordering without knowing ahead of time which variables will be in the minimized clause.

One natural way to completely remove the need for existential quantification is to use Smurfs. Algorithm 10 shows an adaptation of Figure 9 that generates minimal witness clauses from Smurfs. Smurfs naturally represent all possible variable orderings and hence the "best" ordering can easily be chosen on the fly. Due to the fact that a Smurf node may be shared by two or more Smurfs, witness clauses cannot be memoized on Smurf transitions. This makes having an efficient witness

clause generation algorithm even more important because it will be called often during search.

Algorithm 10 SmurfMininf(S, P, l)
P is the set of literals representing an initial witness clause for literal l
Traverse traverses Smurf S according to the set of literals P. If a Smurf transition is taken that infers a literal opposite its value in P, `CONFLICT` is returned. Otherwise, the resulting Smurf node is returned.

```
1:  M := {l}
2:  S := Traverse(S, {¬l})
3:  if S = CONFLICT then return M
4:  for each literal P_i in P do
5:      P := P \ {P_i}
6:      if Traverse(S, ¬P) ≠ CONFLICT then
7:          M := M ∪ {P_i}
8:          S := Traverse(S, {¬P_i})
9:      end if
10: end for
11: return M
```

5.1.2 Experimental Results

The core routines of SBSAT have been compiled into a library and integrated into funcsat, a state of the art CDCL solver [38] (more about funcsat can be found in Section 2.5.2 on page 27). Presented here are experimental results showing the effectiveness of this integration. These results act to provide a baseline and set expectations for integrating state-based SAT with CDCL solvers in general. This integration also allows preprocessing and precomputation methods to be measured by benchmarking their ability to enhance CDCL search.

The first table, Table 5.1 on the next page shows the results of three solvers on the benchmarks used in [120] and [67]. The first solver is SBSAT configured to use the preprocessing techniques introduced in Chapter 3 on page 30. Specifically, CNF pattern matching was applied to reverse engineer common constraints from CNF input. Pattern matching is critical for the performance of both SBSAT and SBSAT + funcsat because it enables full use, or approaches full use, of special Smurfs. During BDD clustering, existential quantification and inference propagation were interleaved with Ve-TreeIter. Ve-TreeIter was configured to use the VarScore clustering heuristic and given a maximum limit of nine variables per conjoined BDD. After preprocessing,

Table 5.1 — Results of SBSAT, funcsat, and SBSAT + funcsat solving benchmarks of [120] and [67].

			SBSAT		funcsat		SBSAT + funcsat	
Instance	Clauses	Vars	Choices	Time	Choices	Time	Choices	Time
par8-1-c	254	64	0	0.01	21	0.04	0	0.00
par8-1	1149	350	0	0.01	25	0.05	0	0.01
par16-1-c	1264	317	5922	0.09	807	0.08	998	0.08
par16-1	3310	1015	107744	0.74	2087	0.27	3104	0.16
par32-1-c	5254	1315	-	-	-	-	-	-
par32-1	10277	3176	-	-	-	-	-	-
barrel5	5383	1407	6818	0.51	4825	0.89	4235	0.60
barrel6	8931	2306	23690	1.23	14607	5.46	12892	2.60
barrel7	13765	3523	62978	4.05	29948	15.75	41513	10.62
barrel8	20083	5106	133410	10.97	63290	53.27	72499	24.60
barrel9	36606	8903	-	-	223862	131.50	411089	149.98
ssa2670-130	3321	1359	52	0.11	259	0.06	45	0.09
ucsc-bf1355-348	7271	2286	0	0.14	63	0.08	0	0.15
dubois100	800	300	0	0.01	2936	0.06	0	0.01
dlx1_c	1614	287	121083569	261.00	681	0.06	744	0.08
1dlx_c_mc...	3725	766	-	-	1811	0.10	2052	0.14
dlx2_cc_bug18	20208	2043	268	4.16	6389	0.33	2102	1.03
2dlx_ca_mc...	24640	3186	-	-	27882	1.89	22939	1.78
2dlx_cc_mc...	41704	4524	-	-	36163	2.79	43900	4.98
2dlx_cc...bug019	48232	4824	-	-	18891	1.59	6611	1.16
9vliw_bp_mc	179492	19148	-	-	646385	43.50	601164	52.30
c499	1870	606	33820644	78.73	5152	0.18	405	0.07
3bitadd_32	32316	4480	-	-	2776	0.26	3821	0.68
x1.1_16	122	46	0	0.00	1228	0.06	0	0.00
x2_128	1018	382	-	-	-	-	-	-
longmult12	18645	5974	144518	8.85	108062	130.79	72934	22.39
longmult14	22389	7176	157078	11.86	99869	118.88	90779	32.48
longmult15	24351	7807	125220	11.30	110436	105.41	77959	27.93
ibm...3-k95	272059	73525	-	-	36067	4.76	49342	6.20
ibm...23-k100	861175	207606	-	-	3973539	4112.46	4385430	751.76
hanoi6	39666	4968	-	-	398980	184.77	413690	120.19
Mat26	2464	744	714557309	9201.62	1806502	1640.60	2047316	706.10
Mat317	85050	24435	-	-	-	-	-	-
linvrinv8	6337	1920	-	-	-	-	-	-
linvrinv9	9154	2754	-	-	-	-	-	-
equilarge_l5	18519	4478	-	-	-	-	-	-
pyhala...4-01	31795	9638	9314604	781.87	-	-	4020593	4057.72
clauses-2	272784	75527	-	-	9093	10.36	63930	10.69
clauses-4	1002957	267766	-	-	63631	183.09	248392	123.13
clauses-6	2623082	683995	-	-	1182887	5875.73	2689386	850.98
clauses-8	5687554	1461771	-	-	-	-	11007228	7656.73
clauses-10	8901946	2270929	-	-	-	-	12505	3016.64

search was performed using the LSGB heuristic without conflict clause learning. The second solver is funcsat configured with its default parameters. The third solver is SBSAT + funcsat configured to preprocess input in exactly the same manner as the first solver. The SBSAT component has been made into a library that provides inferences and witness clauses to funcsat. The funcsat component manages conflict clause computation, BCP, and heuristic computation. This third approach solves all the benchmarks that SBSAT and funcsat solved, along with some that neither SBSAT nor funcsat are able to solve independently. This provides evidence that using state-based SAT methods can enhance the performance of solvers and enables them to solve harder problems than without.

Table 5.2 on the next page demonstrates the benefits of SMURF-based witness clause generation and minimization (see Section 5.1.1 on page 68). This table shows that solver performance can be enhanced by generating minimal witness clauses from SMURFs. In the worst cases, neither total decisions nor decisions per second are significantly impacted by SMURF witness clause minimization and, with few exceptions, both total decisions and total runtime are reduced.

Effects of Precomputation on Solving Time

Table 5.3 on page 75 demonstrates the benefits of building SMURFs on demand versus fully precomputing them (see Section 4.4 on page 61). In both cases, CNF pattern matching was applied to reverse engineer common gates from the original CNF. Existential quantification and inference propagation were interleaved with VE-TREEITER. VE-TREEITER was configured to use the VARSCORE clustering heuristic and given a maximum limit of nine variables per conjoined BDD. The table shows that building SMURFs on demand can reduce runtime in cases where the space the solver needs to search is much less than the total searchable space. This can be seen both in the reduced runtime on most benchmarks and reduced number of SMURF states computed during search. When the space searched is relatively large compared to the total searchable space, lazy SMURF computation can slow the search down but not by a considerable amount. Also, since FUNCSAT's heuristic was used, the number of decisions stays the same whether or not SMURFs are precomputed. This demonstrates that the exact same search space was explored, as expected.

Table 5.2 — Results of SBSAT + funcsat solving benchmarks of [120] and [67] both with and without SMURF-based witness clause minimization.

			SBSAT + funcsat			
			No Minimization		Minimization	
Instance	Clauses	Vars	Choices	Time	Choices	Time
par8-1-c	254	64	0	0.01	0	0.00
par8-1	1149	350	0	0.01	0	0.01
par16-1-c	1264	317	3673	0.22	998	0.08
par16-1	3310	1015	2345	0.14	3104	0.16
par32-1-c	5254	1315	-	-	-	-
par32-1	10277	3176	-	-	-	-
barrel5	5383	1407	3186	0.58	4235	0.60
barrel6	8931	2306	12360	3.36	12892	2.60
barrel7	13765	3523	29948	15.75	41513	10.62
barrel8	20083	5106	40866	21.86	72499	24.60
barrel9	36606	8903	394673	217.61	411089	149.98
ssa2670-130	3321	1359	52	0.24	45	0.09
ucsc-bf1355-348	7271	2286	0	0.36	0	0.15
dubois100	800	300	0	0.01	0	0.01
dlx1_c	1614	287	1416	0.09	744	0.08
1dlx_c_mc...	3725	766	2956	0.16	2052	0.14
dlx2_cc_bug18	20208	2043	2370	1.06	2102	1.03
2dlx_ca_mc...	24640	3186	32099	2.82	22939	1.78
2dlx_cc_mc...	41704	4524	48293	7.19	43900	4.98
2dlx_cc...bug019	48232	4824	29176	5.00	6611	1.16
9vliw_bp_mc	179492	19148	759120	79.14	601164	52.30
c499	1870	606	405	0.08	405	0.07
3bitadd_32	32316	4480	158243	41.70	3821	0.68
x1.1_16	122	46	0	0.00	0	0.00
x2_128	1018	382	-	-	-	-
longmult12	18645	5974	79083	35.57	72934	22.39
longmult14	22389	7176	93716	39.86	90779	32.48
longmult15	24351	7807	77697	32.34	77959	27.93
ibm...3-k95	272059	73525	49241	7.28	49342	6.20
ibm...23-k100	861175	207606	13735678	5674.71	4385430	751.76
hanoi6	39666	4968	606669	250.73	413690	120.19
Mat26	2464	744	2553273	1163.67	2047316	706.10
Mat317	85050	24435	-	-	-	-
linvrinv8	6337	1920	-	-	-	-
linvrinv9	9154	2754	-	-	-	-
equilarge_l5	18519	4478	-	-	-	-
pyhala...4-01	31795	9638	5407158	9942.07	4020593	4057.72
clauses-2	272784	75527	64913	14.96	63930	10.69
clauses-4	1002957	267766	673060	190.47	248392	123.13
clauses-6	2623082	683995	1910619	1649.51	2689386	850.98
clauses-8	5687554	1461771	-	-	11007228	7656.73
clauses-10	8901946	2270929	12103	3542.00	12505	3016.64

Table 5.3 — Results of SBSAT + funcsat solving benchmarks of [120] and [67] both with and without lazy SMURF precomputation.

			SBSAT + funcsat					
			Precomputed SMURFs			On Demand SMURFs		
Instance	Clauses	Vars	Choices	States	Time	Choices	States	Time
par8-1-c	254	64	0	0	0.00	0	0	0.00
par8-1	1149	350	0	0	0.02	0	0	0.01
par16-1-c	1264	317	998	11774	0.11	998	1554	0.08
par16-1	3310	1015	3104	15494	0.21	3104	1600	0.16
par32-1-c	5254	1315	-	-	-	-	-	-
par32-1	10277	3176	-	-	-	-	-	-
barrel5	5383	1407	4235	20681	0.58	4235	6640	0.60
barrel6	8931	2306	12892	25907	2.44	12892	10260	2.60
barrel7	13765	3523	41513	32575	12.06	41513	13575	10.62
barrel8	20083	5106	72499	26331	26.60	72499	12015	24.60
barrel9	36606	8903	411089	92913	162.18	411089	63019	149.98
ssa2670-130	3321	1359	45	9658	0.13	45	390	0.09
ucsc-bf1355-348	7271	2286	0	0	0.18	0	0	0.15
dubois100	800	300	0	0	0.01	0	0	0.01
dlx1_c	1614	287	744	31372	0.12	744	3597	0.08
1dlx_c_mc...	3725	766	2052	73840	0.39	2052	7514	0.14
dlx2_cc_bug18	20208	2043	2102	645328	3.94	2102	30860	1.03
2dlx_ca_mc...	24640	3186	-	-	-	22939	42206	1.78
2dlx_cc_mc...	41704	4524	-	-	-	43900	71855	4.98
2dlx_cc...bug019	48232	4824	6611	731867	4.29	6611	96193	1.16
9vliw_bp_mc	179492	19148	601164	3028316	73.74	601164	147926	52.30
c499	1870	606	405	8563	0.09	405	745	0.07
3bitadd_32	32316	4480	3821	8653	0.74	3821	5132	0.68
x1.1_16	122	46	0	0	0.01	0	0	0.00
x2_128	1018	382	-	-	-	-	-	-
longmult12	18645	5974	72934	74431	27.90	72934	17232	22.39
longmult14	22389	7176	90779	75509	34.77	90779	16854	32.48
longmult15	24351	7807	77959	87341	29.34	77959	17032	27.93
ibm...3-k95	272059	73525	49342	172915	6.69	49342	5842	6.20
ibm...23-k100	861175	207606	4385430	388146	608.00	4385430	138582	751.76
hanoi6	39666	4968	413690	1025187	137.13	413690	484532	120.19
Mat26	2464	744	2047316	57920	733.37	2047316	52044	706.10
Mat317	85050	24435	-	-	-	-	-	-
linvrinv8	6337	1920	-	-	-	-	-	-
linvrinv9	9154	2754	-	-	-	-	-	-
equilarge_l5	18519	4478	-	-	-	-	-	-
pyhala...4-01	31795	9638	4020593	429719	4872.02	4020593	153526	4057.72
clauses-2	272784	75527	63930	291895	13.20	63930	37848	10.69
clauses-4	1002957	267766	248392	342277	128.74	248392	48858	123.13
clauses-6	2623082	683995	2689386	385633	820.24	2689386	56893	850.98
clauses-8	5687554	1461771	11007228	459393	8975.63	11007228	83098	7656.73
clauses-10	8901946	2270929	12505	563008	2719.96	12505	51693	3016.64

Effects of Preprocessing on Solving Time

Figure 5.2 on page 78, Figure 5.3 on page 79, and Figure 5.4 on page 80 demonstrate the effect that preprocessing has on the search processes. The four benchmarks used here were chosen because they exhibit a wide range of behaviors and are representative of many different families of SAT problems. Each of the benchmarks were run thirty-one times using the funcsat + SBSAT approach. For the first run, a limit of zero was placed on the maximum number of variables per BDD allowed to be considered during BDD clustering. This limit was incremented for each subsequent run. The first figure, Figure 5.2 on page 78, provides graphs for each benchmark that show the solving time versus the BDD clustering variable limit. Figure 5.3 on page 79 shows the number of propagations required to solve each benchmark versus the BDD clustering variable limit. On the whole, these two sets of graphs show that preprocessing can reduce the solving time of an instance, and, in every case, effectively decreases the total number of propagations needed to solve an instance.

Since, as is common knowledge, the bulk of SAT search is spent in BCP [24, 150], a decrease in the total number of propagations should decrease total time. This effect is seen near the front of the graphs of Figures 5.2 and 5.3. However, the trend reverses itself and solving time increases towards the back part of the graphs. The increase is due to larger and larger BDDs being created by BDD clustering as the limit grows. As the size of BDDs grow, the number of SMURF nodes needed to represent the BDDs also grows, and quickly becomes prohibitively expensive. This can be seen in Figure 5.4 on page 80 which shows the number of SMURF nodes created during search versus the BDD clustering variable limit.

Notice how the graphs of Figure 5.2 are similar to an overlay of the graphs of Figures 5.3 and 5.4. These figures demonstrate that as propagations go down, search time decreases, but as the number of SMURF nodes grows, search time increases. The cost of building SMURFs, even on demand, is prohibitively high for moderate to large size BDDs and can quickly counteract even large drops in the number of propagations.

Fortunately, there is a local minimum for total time in the middle of each graph. This demonstrates that the preprocessing methods introduced in Chapter 3 on page 30 can be effective at reducing solving times, but if overused, can harm the search. It is also possible that solving times

could be reduced if the SMURF generation component of SBSAT was optimized. At the moment, SBSAT is considered to be research grade and preprocessing will become more of a benefit as SMURF creation is further optimized by, for example, further increasing node sharing between SMURFs (SMURF normalization goes some way in doing this), adding more special SMURF types, increasing caching performance and memory management, and creating specialized co-factoring and inference discovery functions (two BDD functions called often during SMURF creation).

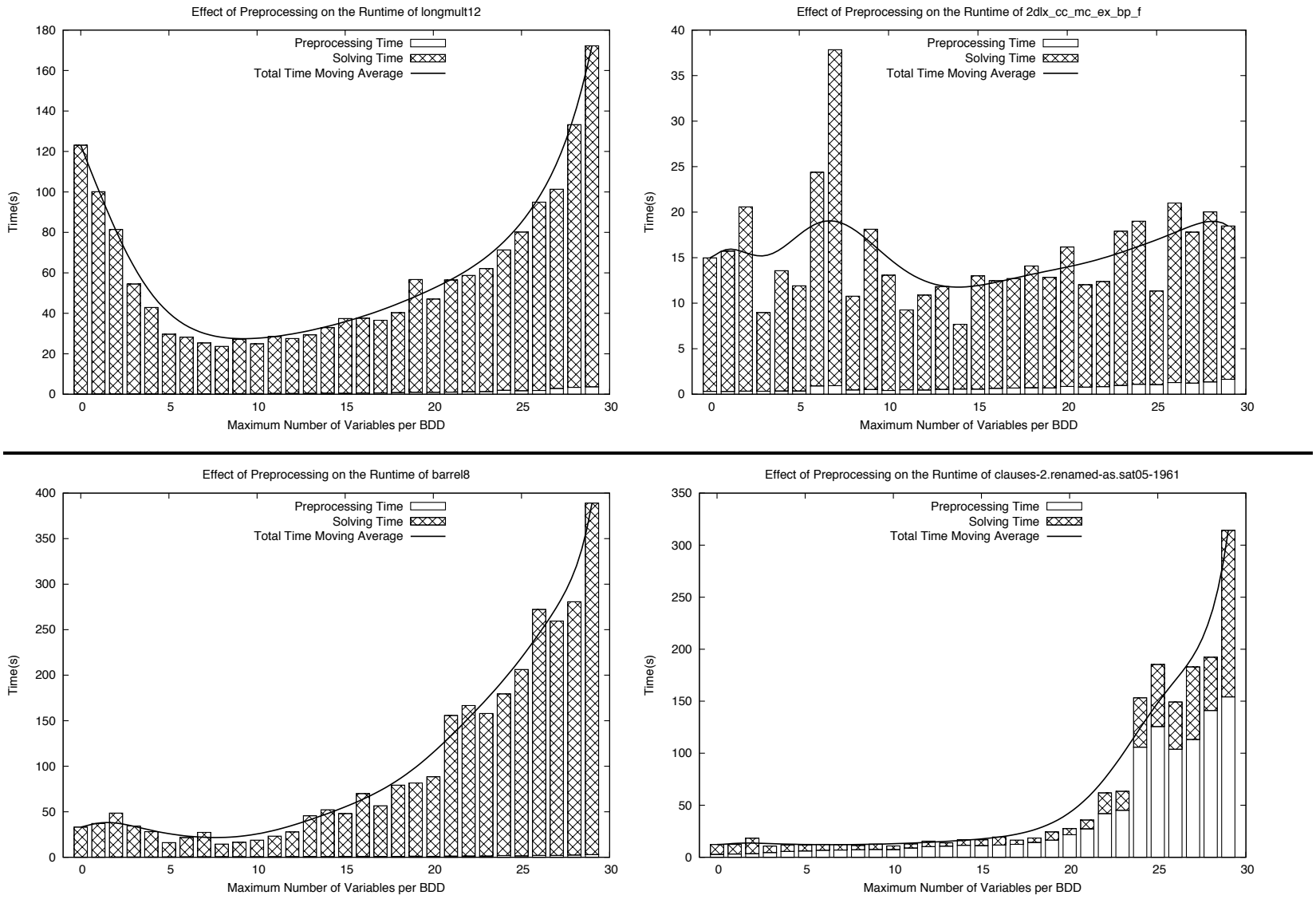

Figure 5.2 — These graphs show the effects of preprocessing on the solving time of the longmult12, 2dlx_cc_mc_ex_bp_f, barrel8, and clauses-2 benchmarks.

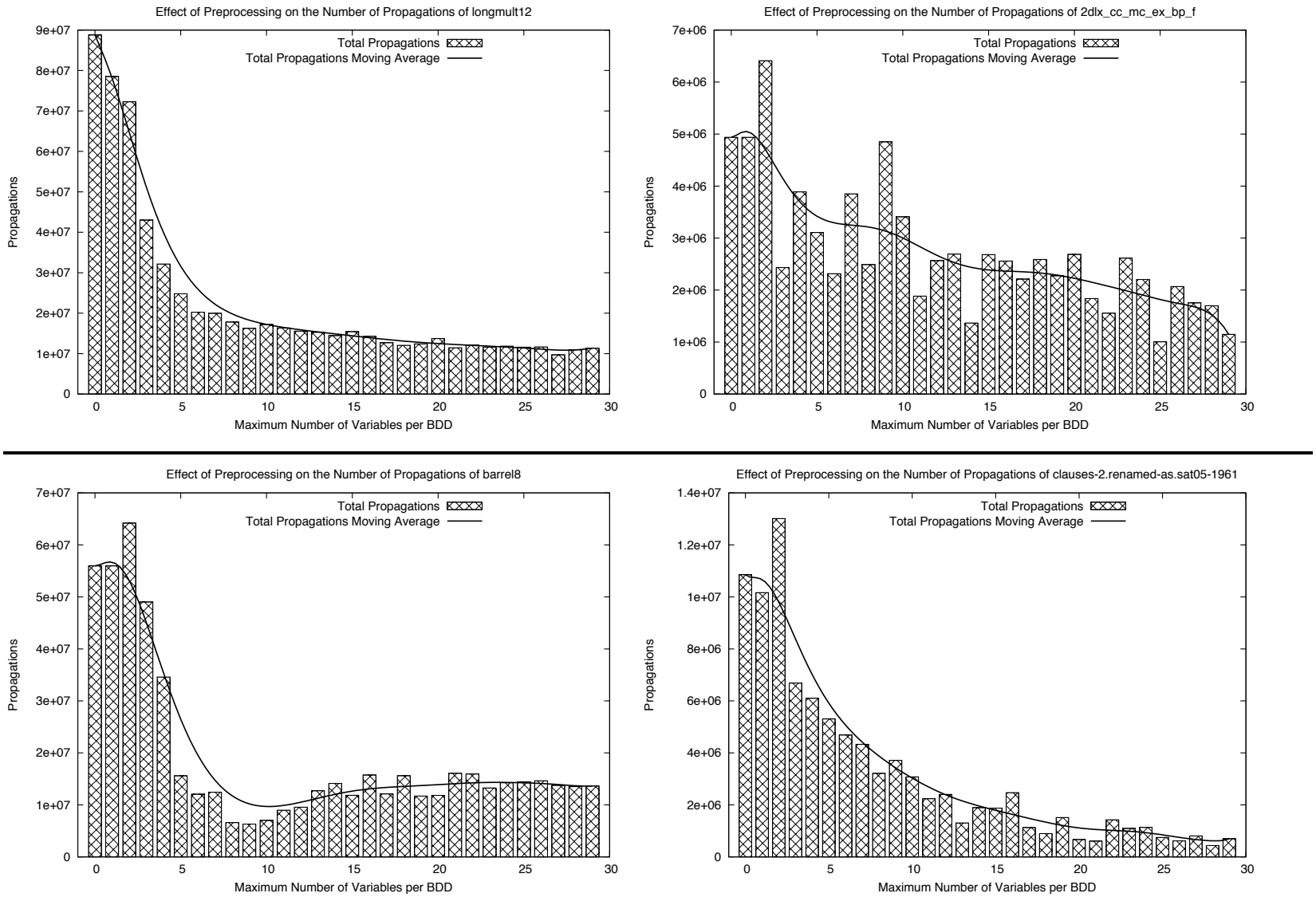

Figure 5.3 — These graphs show the effects of preprocessing on the number of propagations taken to solve the longmult12, 2dlx_cc_mc_ex_bp_f, barrel8, and clauses-2 benchmarks.

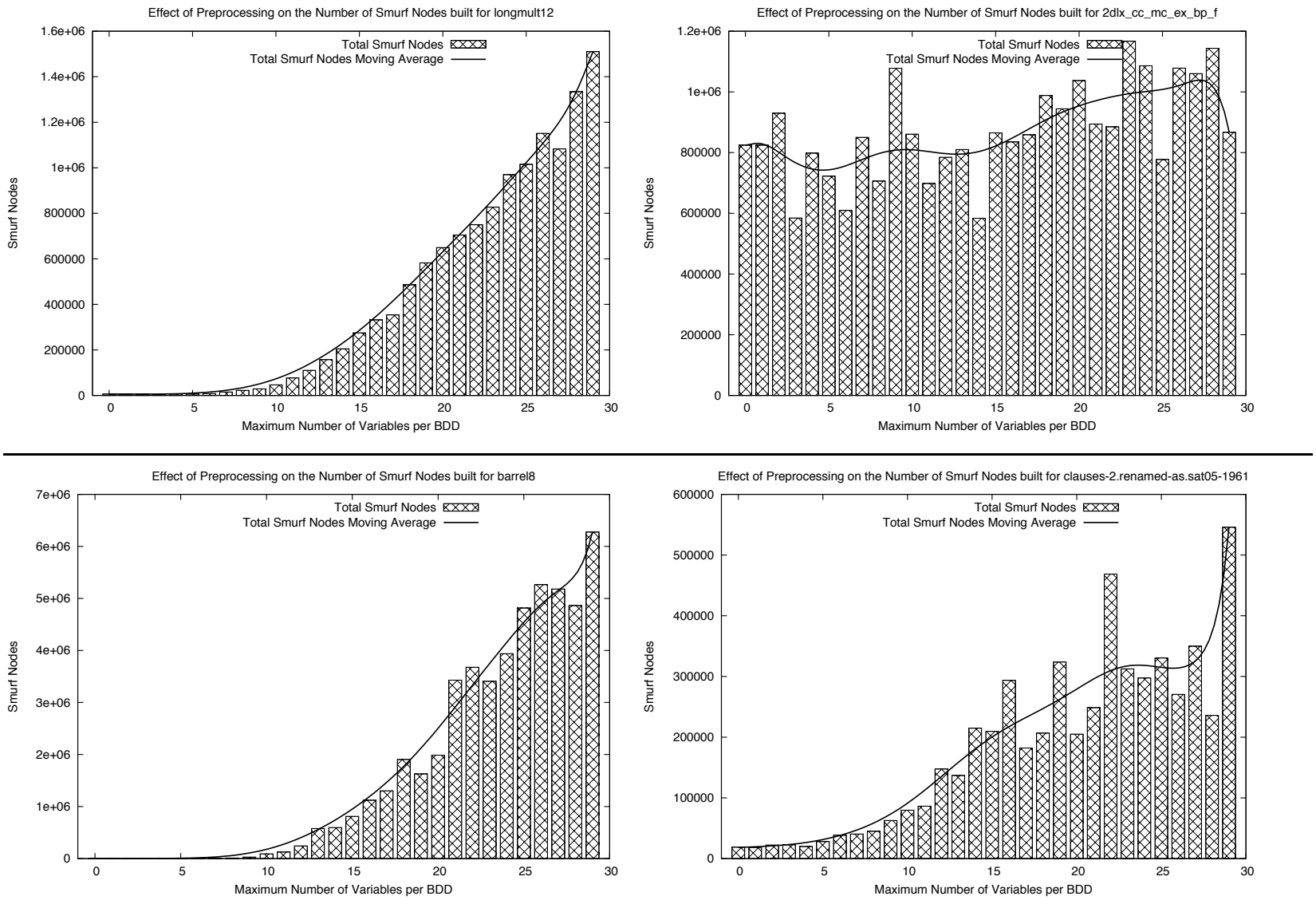

Figure 5.4 — These graphs show the effects of preprocessing on the number of SMURF nodes needed to solve the longmult12, 2dlx_cc_mc_ex_bp_f, barrel8, and clauses-2 benchmarks.

5.2 Support for Special Purpose Solvers

Recent interest in Satisfiability Modulo Theories (SMT), the mixing of SAT solvers with specialized theory solvers, has enabled developers to create powerful solvers. This is, in part, due to both heavy use of extensible SAT solvers as well as use of an input language (called SMT-lib) that is much more expressive than CNF. Allowing the user to write high-level constraints, such as $z \leq x+y$ where x, y, and z are integers, provides special purpose solvers with information that enables them to make cuts in the search space that could not easily be made with purely CNF input. This section discusses the use of special purpose solvers in support of stronger search and SMURF compression. As an example, introduced here are techniques for discovering and memoizing parity constraints (`XOR`s) on SMURF transitions and making use of Gaussian elimination. Finding `XOR` factors is pertinent to state-based SAT because non-clausal domains, such as verification, have constraints that naturally partition into linear (`XOR`) and non-linear components.

Section 4.1.5 on page 55 shows that SMURFs can be further compressed by factoring out `XOR` components when possible and handing them to a special purpose solver when transitioned into during search. Specifically, `XOR` constraints memoized on SMURF transitions can be handed off to a backtrack-capable Gaussian elimination solver during search. Gaussian elimination, an algorithm with complexity polynomial in the number of variables ($O(n^3)$), can efficiently detect inferences and conflicts existing in a conjunction of `XOR` constraints. This is in contrast to some recent CDCL solvers that also make use of Gaussian elimination (e.g. March_eq [79], MoRsat [43], and CryptoMiniSat [132]). These solvers discover `XOR` functions during preprocessing, populate and row-reduce the Gaussian elimination table, and then periodically check the table against the current partial assignment to discover new inferences and conflicts. These solvers may also use their learned clause database to find new global `XOR` constraints, adding those to their Gaussian elimination table during search.

Described here are the techniques needed to discover and remove factors from BDDs. A Boolean function f is a *factor* of a Boolean function g if and only if $(\overline{f} \wedge g) \equiv$ `False`, i.e. the negation of the factor conjoined with the original function is `False` (unsatisfiable). As an example, consider

the following non-linear Boolean function:

$$x_1x_2 \oplus x_2x_3 \oplus x_1 \oplus x_3 \equiv \texttt{True}$$

which has two `XOR` factors, namely:

$$x_1 \oplus x_3 \equiv \texttt{True},\ x_2 \equiv \texttt{False}$$

Figure 5.5 shows that BDD operations can be used to verify conjectures about potential factors. This figure and many others in this section were created by the BDD Visualizer [140].

```
order(x4, x3, x2, x1)

xor(and(x1, x2), and(x2, x3), x1, x3)) ; $1

; Is (x1 ⊕ x3 ≡ True) a factor of
; (x1x2 ⊕ x2x3 ⊕ x1 ⊕ x3 ≡ True)?
and(not(xor(x1, x3)), $1) ; $2

; Is (x2 ≡ False) a factor of
; (x1x2 ⊕ x2x3 ⊕ x1 ⊕ x3 ≡ True)?
and(x2, $1) ; $3

print(or($2, $3))
```

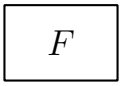

Figure 5.5 — A figure generated by the BDD Visualizer [140] showing that $x_1 \oplus x_3 \equiv \texttt{True}$ and $x_2 \equiv \texttt{False}$ are factors of $x_1x_2 \oplus x_2x_3 \oplus x_1 \oplus x_3 \equiv \texttt{True}$. The input to the BDD Visualizer is given on the left and the generated BDD (the `False` leaf node) is produced on the right.

Unfortunately, using this method to find `XOR` and constant factors requires *guessing* the potential factors, of which there are 2^{n+1} for functions with n variables. However, BDD operations can be used to discover all possible `XOR` factors without guessing. For the next example, the following degree 3 function (named *complex*) will be used. This function is visualized in Figure 5.6:

$$x_1x_3x_4 \oplus x_3x_4 \oplus x_1x_2x_4 \oplus x_1x_2x_3 \oplus x_2x_3 \oplus x_1x_2 \equiv \texttt{True}$$

```
order(x4, x3, x2, x1)

; $1 - Complex function
xor(and(x1, x3, x4),
    and(x3, x4),
    and(x1, x2, x4),
    and(x1, x2, x3),
    and(x2, x3),
    and(x1, x2))

print($1)
```

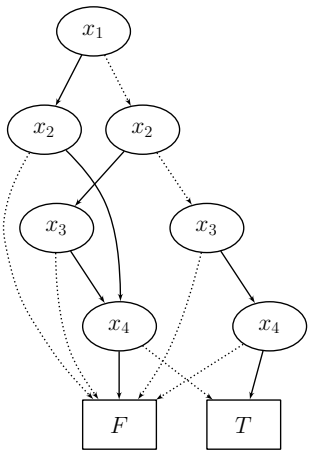

Figure 5.6 — A visualization of the BDD for the complex function: $x_1x_3x_4 \oplus x_3x_4 \oplus x_1x_2x_4 \oplus x_1x_2x_3 \oplus x_2x_3 \oplus x_1x_2 \equiv \texttt{True}$.

This function has one `XOR` factor. The following quadratic function will be used to help discover the factor automatically:

$$c_1x_1 \oplus c_2x_2 \oplus c_3x_3 \oplus c_4x_4 \oplus c_{\texttt{T}} \equiv \texttt{True}$$

The c_i variables, where $1 \leq i \leq n$, and the c_T variable in the quadratic function are coefficients that act as selectors, denoting which x_i variables are part of the resulting `XOR` factors or whether the `XOR` factor is negated, as is the case when c_T takes value `True`, i.e. assigning a value `True` or `False` to each of the $n+1$ coefficients will result in one of 2^{n+1} `XOR` or constants. It's worth noting that the technique described here can be used to search for factors of any form; simply use a different quadratic function to find factors of a different form.

All configurations of the coefficients that denote `XOR` factors of the complex function from above can be found automatically. This requires two steps and is done by first substituting the quadratic function for f and the complex function for g in $(\overline{f} \wedge g)$. This gives the function:

$$((\overline{c_1x_1 \oplus c_2x_2 \oplus c_3x_3 \oplus c_4x_4 \oplus c_{\texttt{T}}}) \wedge$$

$$(x_1x_3x_4 \oplus x_3x_4 \oplus x_1x_2x_4 \oplus x_1x_2x_3 \oplus x_2x_3 \oplus x_1x_2)).$$

See Figure 5.7 on the following page for a visualization of the BDD for this function.

```
order(x4, c4, x3, c3, x2, c2, x1, c1, cT)

; $1 - Quadratic function
xor(and(c1, x1), and(c2, x2), and(c3, x3),
    and(c4, x4), cT)

; $2 - Complex function
xor(and(x1, x3, x4),
    and(x3, x4),
    and(x1, x2, x4),
    and(x1, x2, x3),
    and(x2, x3),
    and(x1, x2))

; $3 - Find all possible factors, step 1
and(not($1), $2)

print($3)
```

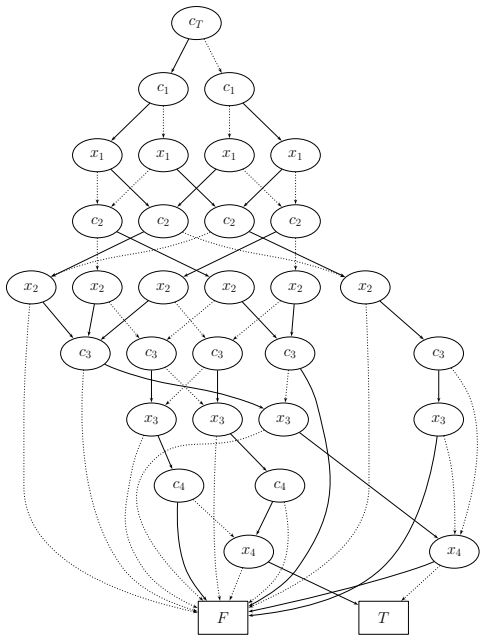

Figure 5.7 — A visualization of the BDD for the conjunction of the quadratic and complex functions.

As mentioned earlier, the c_i variables denote which x_i variables are part of the `XOR` factors. The second step to finding `XOR` factors is to existentially quantify away the x_i variables. This produces the function where every falsifying assignment represents a valid configuration of the coefficients, and hence an `XOR` factor:

$$\exists x_i \quad ((\overline{c_1x_1 \oplus c_2x_2 \oplus c_3x_3 \oplus c_4x_4 \oplus c_\mathtt{T}}) \wedge$$
$$(x_1x_3x_4 \oplus x_3x_4 \oplus x_1x_2x_4 \oplus x_1x_2x_3 \oplus x_2x_3 \oplus x_1x_2)).$$

This function is visualized in Figure 5.8 on the next page.

Each factor of the complex function is denoted by a path from the root to the `False` leaf of the BDD. In Figure 5.8 on the following page, there are two paths to `False`, namely,

1. $c_\mathtt{T}, \overline{c_1}, \overline{c_2}, \overline{c_3}, \overline{c_4}$
2. $\overline{c_T}, \overline{c_1}, c_2, \overline{c_3}, c_4$

These paths correspond to the `XOR` and constant factors of the complex function. The coefficients

```
order(x4, c4, x3, c3, x2, c2, x1, c1, cT)

; $1 - Quadratic function
xor(and(c1, x1), and(c2, x2), and(c3, x3),
    and(c4, x4), cT)

; $2 - Complex function
xor(and(x1, x3, x4),
    and(x3, x4),
    and(x1, x2, x4),
    and(x1, x2, x3),
    and(x2, x3),
    and(x1, x2))

; $3 - Find all possible factors, step 1
and(not($1), $2)

; $4 - Find all possible factors, step 2
exist($3, x1, x2, x3, x4) ; $4
print($4)
```

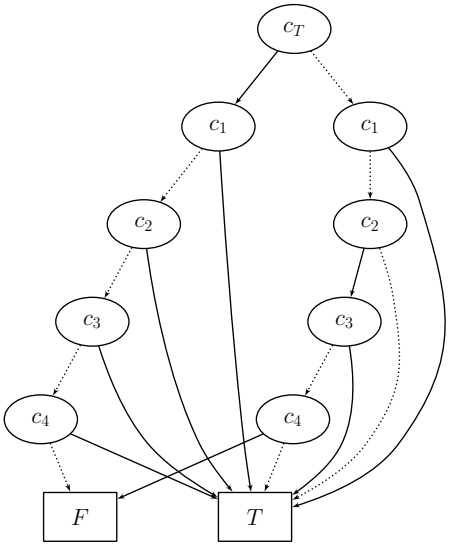

Figure 5.8 — A visualization of the BDD such that every path to `False` represents an `XOR` factor of the complex function.

taking value `True` represent these factors, namely:

$$\texttt{True} \equiv \texttt{True}, \;\; x_2 \oplus x_4 \equiv \texttt{True}$$

The first factor is trivial; `True` is a factor of every Boolean function. The second factor is more interesting, and was found automatically from among the 2^5 possible `XOR` factors. During SMURF creation, this factor can be memoized on a SMURF transition. The SMURF state that is transitioned into has this factor removed. An implementation of generalized co-factoring, named CONSTRAIN, can be used to remove this factor from the original complex function. This operation is illustrated in Figure 5.9 on the next page. As described in Section 2.3 on page 18, CONSTRAIN takes two BDDs as arguments, f and c, and if c is a factor of f, returns a BDD g such that $f \equiv g \wedge c$. Of course, there may be many such BDDs g. CONSTRAIN is not a purely logical operation and can return logically different BDDs depending on BDD variable ordering. In other words, many of the different possible BDDs g, such that $f \equiv g \wedge c$, can be generated by CONSTRAIN by selecting different BDD variable orderings. The variable order that seems to work best, i.e. removes the most of c from f, is when the variables in c are nearest the leaves of the BDD and the variables in f but not in c are nearest the root. From this standpoint, CONSTRAIN can basically be thought of as performing Boolean division. Boolean division is a complex subject, generally used when performing logic decomposition during circuit synthesis (see [134] for an introduction). Figure 5.9 on the next page shows the result of using CONSTRAIN to remove the factor $x_2 \oplus x_4 \equiv \texttt{True}$ from the complex function.

The resulting function is $x_1x_2 \oplus x_3 \oplus x_1x_3 \equiv \texttt{True}$, which can also be written as $ite(x_1, x_2, x_3)$. This means that the original complex function can be factored into the following two functions:

$$\begin{aligned} x_1x_2 \oplus x_3 \oplus x_1x_3 &\equiv \texttt{True} \\ x_2 \oplus x_4 &\equiv \texttt{True}. \end{aligned}$$

This representations is much simpler than the original complex function:

```
order(x4, x3, x2, x1)

; $1 - complex function
xor(and(x1, x3, x4),
    and(x3, x4),
    and(x1, x2, x4),
    and(x1, x2, x3),
    and(x2, x3),
    and(x1, x2))

; Remove the XOR factor
print(constrain($1, xor(x2, x4))
```

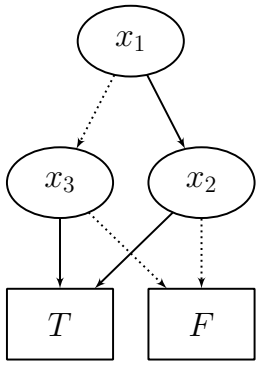

Figure 5.9 — A visualization of the BDD for the complex function after removing the `XOR` factor $x_2 \oplus x_4 \equiv \texttt{True}$ using CONSTRAIN

$$x_1x_3x_4 \oplus x_3x_4 \oplus x_1x_2x_4 \oplus x_1x_2x_3 \oplus x_2x_3 \oplus x_1x_2 \equiv \texttt{True}$$

5.2.1 Experimental Results

These techniques have been implemented in SBSAT and provide the means to memoize `XOR` factors on SMURF transitions, drastically reducing the size of a SMURF. They also allow search to make use of a Gaussian elimination solver to derive inferences and conflicts earlier. Figure 5.10 on the following page provides a diagram showing the communication between the two types of solvers.

Three tables of results are given. The first, Table 5.4 on page 89, shows results of PicoSAT, CryptoMiniSat, and SBSAT solving an unsatisfiable suite of SAT competition benchmarks that were translated into CNF from conjunctions of `XOR` constraints. As such, each benchmark can be solved by discovering `XOR` constraints and using Gaussian elimination. Results of PicoSAT demonstrate that purely resolution-based solvers perform poorly on conjunctions of `XOR`s. A "-" symbol in PicoSAT's "Choices" column indicates that more than 2^{32} decisions were needed to solve the particular benchmark and the corresponding time listed in the next column is the number of seconds PicoSAT took to either solve the benchmark or reach 2^{32} decisions. CrytoMiniSat was configured to make heavy use of Gaussian elimination and is able to solve every benchmark with

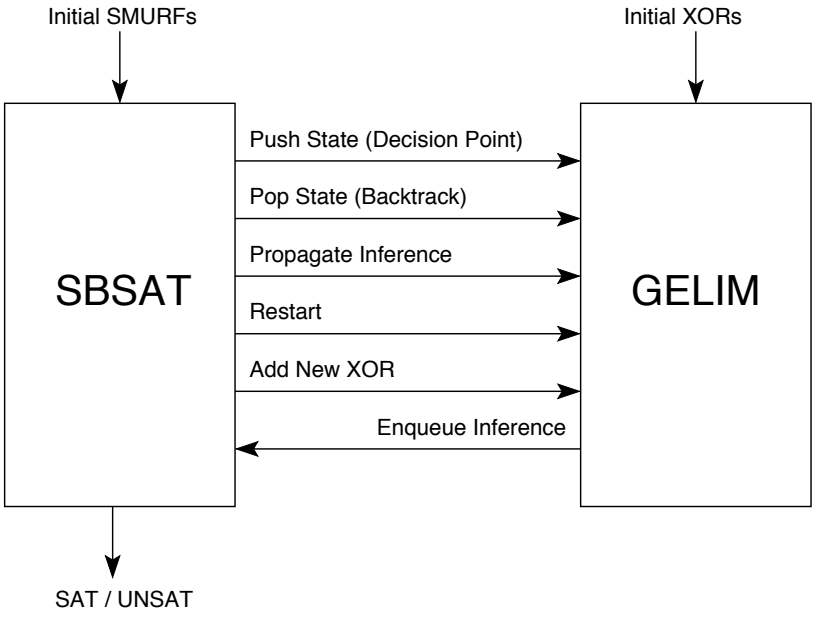

Figure 5.10 — Communication between SBSAT and a backtrack capable Gaussian elimination solver.

few decisions. CNF pattern matching and Gaussian elimination were used by SBSAT, which solved every benchmark without needing to make any decisions.

Table 5.5 on page 91 shows the results of the same three solvers on SAT competition benchmarks that were translated into CNF from satisfiable conjunctions of `XOR` constraints. As with the previous benchmarks, these can be solved by discovering `XOR` constraints and using Gaussian elimination, but, due to being under constrained, a small amount of additional branching is required. Again, PicoSAT and CryptoMiniSat are shown along side SBSAT. For this set of benchmarks, SBSAT was configured to also perform BDD clustering and then either to build SMURFs with and without factoring and memoizing `XOR` factors on SMURF transitions. The column where `XOR` factors were not memoized shows that BDD clustering blurs the lines between `XOR` constraints and hence their power to contribute to the Gaussian elimination solver is considerably reduced. The last column ("with `XOR` factors") demonstrates that the harmful effects of BDD clustering on the power of Gaussian elimination are removed by factoring out and memoizing `XOR` factors on SMURF transitions. In this table the "-" symbol indicates that more than 50000 seconds were needed to solve the corresponding

Table 5.4 — Statistics of PicoSAT, CryptoMiniSat, and SBSAT solving unsatisfiable sets of XOR functions encoded into CNF.

			PicoSAT		CryptoMiniSat		SBSAT	
Instance	Clauses	Vars	Choices	Time	Choices	Time	Choices	Time
urquhart2_25bis	96	36	1130	0.0	1858	0.01	0	0.01
urquhart2_25	160	60	37437	0.1	45867	0.20	0	0.02
urquhart3_25bis	264	99	2263042	6.5	3010	0.02	0	0.21
urquhart3_25	408	153	80223721	297.7	205289	1.02	0	0.28
urquhart4_25bis	512	192	-	13021.6	217156	1.00	0	0.28
urquhart4_25	768	288	-	18673.7	3296	0.03	0	0.46
x1.1_40	314	118	10214388	34.8	2843	0.03	0	0.24
x1.1_44	346	130	70207716	261.5	3232	0.02	0	0.21
x1.1_48	378	142	13980039	41.9	3150	0.02	0	0.30
x1.1_56	442	166	-	12150.4	295994	1.52	0	0.24
x1.1_64	506	190	-	14765.6	302231	1.84	0	0.22
x1.1_72	570	214	-	16926.5	313907	1.79	0	0.37
x1.1_80	634	238	-	17314.4	3089	0.02	0	0.34
x1.1_96	762	286	-	20874.6	3199	0.02	0	0.35
x1.1_128	1018	382	-	31616.9	3455	0.04	0	0.68
x1_36	282	106	2527092	7.1	3131	0.02	0	0.22
x1_40	314	118	9089985	29.8	3342	0.02	0	0.26
x1_44	346	130	53476293	190.0	3088	0.02	0	0.28
x1_48	378	142	17775926	60.9	3266	0.02	0	0.27
x1_56	442	166	-	11768.9	206229	0.92	0	0.30
x1_64	506	190	-	13512.5	295054	1.52	0	0.26
x1_72	570	214	-	17540.8	335951	2.98	0	0.33
x1_80	634	238	-	18647.8	3127	0.03	0	0.35
x1_96	762	286	-	22828.3	3436	0.04	0	0.44
x1_128	1018	382	-	28004.4	3268	0.03	0	0.58
x2_40	314	118	8869047	31.5	2898	0.04	0	0.21
x2_44	346	130	104373538	440.4	3008	0.03	0	0.24
x2_48	378	142	109551245	464.8	2834	0.02	0	0.27
x2_56	442	166	-	12859.3	211062	1.49	0	0.30
x2_64	506	190	-	16051.1	220468	1.27	0	0.34
x2_72	570	214	-	17991.2	330922	1.93	0	0.30
x2_80	634	238	-	18258.3	2888	0.03	0	0.34
x2_96	762	286	-	21182.9	3319	0.04	0	0.40
x2_128	1018	382	-	29657.0	3624	0.02	0	0.60

benchmark.

Table 5.6 on page 92 shows the results of the solvers on SAT competition benchmarks that were translated into CNF from both unsatisfiable and satisfiable conjunctions of a mixture of `XOR` constraints, `AND/OR` gates, and clauses. As such, many of these benchmarks require branching and are greatly assisted by CNF pattern matching, BDD clustering, `XOR` factor memoization, and Gaussian elimination. All three solvers are identically to how they are in Table 5.5 on the following page.

Table 5.5 — Statistics of PicoSAT, CryptoMiniSat, and SBSAT solving satisfiable sets of XOR functions encoded into CNF. Some benchmarks required an excess of 50000 seconds. There are indicated by "-" in the table.

							SBSAT			
			PicoSAT		CryptoMiniSat		Without XOR Factors		With XOR Factors	
Instance	Clauses	Vars	Choices	Time	Choices	Time	Choices	Time	Choices	Time
mod2-rand3bip-sat-200-1	800	200	732171	11.4	2363	0.05	2407934	18.25	66	0.49
mod2-rand3bip-sat-200-2	800	200	492595	8.5	259718	16.21	1422251	17.17	73	0.49
mod2-rand3bip-sat-200-3	800	200	1211954	19.3	2354	0.05	31747848	299.15	73	0.59
mod2-rand3bip-sat-210-1	840	210	2314671	39.5	2317	0.05	41408558	275.08	76	0.55
mod2-rand3bip-sat-210-2	840	210	690088	10.2	2337	0.05	21797752	181.55	67	0.56
mod2-rand3bip-sat-210-3	840	210	3742815	68.6	2419	0.05	8513244	93.34	72	0.60
mod2-rand3bip-sat-220-1	880	220	1609693	27.9	2417	0.05	29837658	237.90	73	0.57
mod2-rand3bip-sat-220-2	880	220	10754958	221.7	2383	0.05	117109188	1300.98	73	0.61
mod2-rand3bip-sat-220-3	880	220	4714361	85.0	2439	0.05	49840558	430.35	76	0.59
mod2-rand3bip-sat-230-1	920	230	11336075	229.8	2418	0.05	25283878	231.41	77	0.60
mod2-rand3bip-sat-230-2	920	230	14566733	294.4	2352	0.04	680661489	8417.81	75	0.66
mod2-rand3bip-sat-230-3	920	230	2732624	50.2	2488	0.06	12524208	134.96	83	0.60
mod2-rand3bip-sat-240-1	960	240	22657621	489.4	2377	0.05	41142493	461.68	82	0.64
mod2-rand3bip-sat-240-2	960	240	3253346	67.7	2391	0.04	118284155	1090.84	84	0.63
mod2-rand3bip-sat-240-3	960	240	3244083	60.5	2416	0.05	81969106	969.28	82	0.81
mod2-rand3bip-sat-250-1	1000	250	4335960	95.3	2358	0.05	364526854	4459.51	84	0.68
mod2-rand3bip-sat-250-2	1000	250	5601671	119.0	2408	0.05	329904878	3519.39	83	0.68
mod2-rand3bip-sat-250-3	1000	250	21677872	533.0	2418	0.05	1256588334	15024.89	83	0.70
mod2-rand3bip-sat-260-1	1040	260	28610815	733.5	2383	0.05	2229157458	18340.39	87	0.69
mod2-rand3bip-sat-260-2	1040	260	68793912	1951.7	2450	0.05	524055195	3353.07	87	0.82
mod2-rand3bip-sat-260-3	1040	260	57607402	1677.9	2351	0.05	193780814	1686.81	91	0.72
mod2-rand3bip-sat-270-1	1080	270	321106608	11536.1	2486	0.06	1349285293	12367.49	81	0.76
mod2-rand3bip-sat-270-2	1080	270	232531238	8527.1	2410	0.05	2241682992	22645.08	92	0.74
mod2-rand3bip-sat-270-3	1080	270	51349937	1580.4	2443	0.04	2528576057	25516.49	92	0.77
mod2-rand3bip-sat-280-1	1120	280	63730615	2043.7	2517	0.07	1821830240	12133.98	94	0.77
mod2-rand3bip-sat-280-2	1120	280	329644078	14201.8	2441	0.06	3309838754	24877.75	89	0.70
mod2-rand3bip-sat-280-3	1120	280	57692392	1898.3	2398	0.06	-	-	95	0.68
mod2-rand3bip-sat-290-1	1160	290	125689528	4554.9	2384	0.06	-	-	95	0.79
mod2-rand3bip-sat-290-2	1160	290	532869776	24150.9	2469	0.05	3135243932	35747.49	102	0.72
mod2-rand3bip-sat-290-3	1160	290	251893381	10153.5	2449	0.04	2744688261	45144.08	95	0.84
mod2-rand3bip-sat-300-1	1200	300	363501140	16312.5	2469	0.04	-	-	98	0.90
mod2-rand3bip-sat-300-2	1200	300	240947655	9348.1	2490	0.06	1543341849	37301.21	97	0.80
mod2-rand3bip-sat-300-3	1200	300	645756972	30283.4	2397	0.06	-	-	95	0.88

Table 5.6 — Statistics of PicoSAT, CryptoMiniSat, and SBSATsolving a satisfiable and unsatisfiable benchmarks with a mixture of linear (XOR) and non-linear functions encoded into CNF.

							SBSAT			
			PicoSAT		CryptoMiniSat		Without XOR Factors		With XOR Factors	
Instance	Clauses	Vars	Choices	Time	Choices	Time	Choices	Time	Choices	Time
pmg-11-UNSAT	562	169	10670017	201.4	25490361	11920.04	19705377	72.70	0	0.64
pmg-12-UNSAT	632	190	56286374	1226.7	-	-	185150621	758.63	0	0.62
pmg-13-UNSAT	1362	409	-	-	-	-	-	-	0	1.40
pmg-14-UNSAT	1922	577	-	-	-	-	-	-	0	2.63
par8-1-c	254	64	3	0.0	26	0.02	0	0.03	0	0.01
par8-2-c	270	68	7	0.0	22	0.02	55	0.68	56	0.27
par8-3-c	298	75	5	0.0	10	0.02	0	0.03	0	0.01
par8-4-c	266	67	10	0.0	29	0.01	56	0.82	54	0.28
par8-5-c	298	75	13	0.0	19	0.01	58	1.98	53	0.32
par8-1	1149	350	22	0.0	15	0.02	0	0.05	0	0.02
par8-2	1157	350	13	0.0	11	0.02	0	0.05	0	0.02
par8-3	1171	350	30	0.0	11	0.02	0	0.06	0	0.02
par8-4	1155	350	9	0.0	10	0.02	0	0.07	0	0.04
par8-5	1171	350	23	0.0	29	0.01	340	2.03	338	0.54
par16-1-c	1264	317	4971	0.1	5384	0.09	5223	5.45	290	1.24
par16-2-c	1392	349	7794	0.2	812	0.04	10669	7.25	293	1.60
par16-3-c	1332	334	1744	0.0	6579	0.12	5418	5.38	309	2.00
par16-4-c	1292	324	913	0.0	2985	0.06	3111	9.50	274	1.10
par16-5-c	1360	341	2257	0.0	1431	0.03	320	2.98	340	1.29
par16-1	3310	1015	5250	0.1	2790	0.05	5205	8.08	1016	3.46
par16-2	3374	1015	6499	0.2	647	0.03	3166	8.32	1116	3.97
par16-3	3344	1015	5098	0.1	2113	0.05	5194	14.21	960	2.87
par16-4	3324	1015	319	0.0	2062	0.04	8132	5.40	1092	3.58
par16-5	3358	1015	3528	0.1	1169	0.03	27961	7.46	1131	3.22
par32-1-c	5254	1315	473449798	52791.7	-	-	138677	13.00	9782	5.05
par32-2-c	5206	1303	-	-	42024560	32746.43	59675	19.48	1227	4.38
par32-3-c	5294	1325	580561571	61445.1	-	-	610765	25.07	90955	30.07
par32-4-c	5326	1333	186730471	23431.3	-	-	1770005	132.28	153058	8.38
par32-5-c	5350	1339	400511789	51084.9	-	-	15280	7.36	10701	5.08
par32-1	10277	3176	294096091	77129.5	28902	1.05	201586	38.38	309385	18.83
par32-2	10253	3176	76397975	10670.7	24571	1.01	810034	45.42	51529	8.68
par32-3	10297	3176	96479963	16810.6	-	-	2227730	219.94	143826	12.12
par32-4	10313	3176	204408467	30585.0	-	-	1028205	55.48	83297	9.70
par32-5	10325	3176	50826882	8274.0	-	-	1352884	69.88	60871	9.72

Chapter 6

Conclusion

6.1 Contributions

The many new techniques introduced here have been implemented and incorporated into SBSAT. These techniques, which include methods for preprocessing, precomputation, and state-based SAT search, are shown to significantly enhance SBSAT both in terms of scalability and power, enabling it to solve many problems that it could not solve before. Also, a teaching tool, the BDD Visualizer [140], has been developed and already used by many people to learn about and explore BDD-based techniques on their own problems.

Building one Smurf per CNF clause denies every advantage to state-based SAT search over CDCL solvers. The translation from a constraint rich user-domain problem into CNF often obscures the original structure, structure that can make it feasible for a state-based SAT solver to solve the problem. To overcome this deficiency, Chapter 3 on page 30 introduced new methods for preprocessing both CNF instances and conjunctions of BDDs. These preprocessing methods have been implemented in SBSAT, evaluated on standard SAT benchmarks for their ability to recover structure from CNF input, and were found to outperform previous methods.

Chapter 4 on page 48 introduced new data structures and precomputation methods for transforming preprocessed input into Smurfs. The new data structures provide special Smurf representations for common Boolean functions, specifically those produced by preprocessing. Special

SMURFs are more compact than their general SMURF counterparts, only taking linear space (or quadratic in the case of cardinality constraints) instead of exponential, as is the case with every corresponding general SMURF. Special SMURFs, like general SMURFs, are arc-consistent and support efficient inference propagation and heuristic computation during state-based SAT search. However, special SMURFs have only been introduced for a handful of common Boolean functions, too few to truly scale to industrial problems where user-level constraints may not fit a special SMURF type. To support scalability, new techniques were introduced to both increase sharing between SMURFs and delay prohibitively expensive precomputation. Each of the new special SMURF data structures and precomputation methods have been implemented in SBSAT and evaluated. Results show that the various techniques lead to faster search and a smaller memory footprint, greatly increasing scalability of SBSAT.

The preprocessing and precomputation methods introduced in previous chapters were mainly developed in support of state-based SAT search. The preprocessing, precomputation, and search algorithms in SBSAT have been compiled into a library that can be made use of by off-the-shelf CDCL solvers. This means that SBSAT as a library can be used to add arbitrary Boolean constraints to any CDCL solver, without translating to CNF, creating special purpose constraints, sacrificing arc-consistency, or decreasing the power of witness clauses. This library has been integrated into funcsat, an extensible state of the art CDCL solver and evaluated. Experimental results of this evaluation show that allowing SBSAT to maintain user-domain constraints, rather than translating into CNF, can speed up search as well as shrink the size of the search tree. However, there seems to be a local minimum in terms of how much preprocessing should be done where state-based SAT techniques transition from helping to hurting.

A Gaussian elimination solver was integrated into SBSAT and used to demonstrate methods for adapting other constraint solving techniques to state-based SAT. Specifically, methods were introduced that show how to factor `XOR` constraints from Boolean functions, leading to the creation of special SMURF transitions that infer `XOR` constraints during search. These `XOR` factors are handed off to the Gaussian elimination solver, that can produce inferences much earlier during search. This approach was experimentally evaluated against other similar techniques and found to be more

robust.

This dissertation showed how the state-based SAT paradigm is often superior to the current mainstream and that a true melding of search and symbolic techniques can produce scalable, powerful, and efficient generalizations and extensions of current SAT-based methods. Finally, this dissertation provides more of the support needed to move past the antiquated notion that SAT constraints are forever clauses.

6.2 Future Work

This section discusses some of the many open research directions in the realm of state-based SAT.

SBSAT is a *research grade* state-based SAT framework. For state-based SAT to propagate, there needs to be a professional grade extensible framework that manages SMURF node allocation, deletion, and transition. Such a framework could be used by any constraint solver developer to compactly represent and perform search on user-domain constraints. So, state-based SAT could benefit from leveraging serious software engineering expertise.

Currently, SMURFs produce witness *clauses* for inferences found during search. This can be, simply put, a huge drain on the power gained by using SMURFs. This sentiment is perfectly stated in [58]:

> ...the learning method is the key to improving the strength of the underlying proof system... Methods that use stronger representations, but fail to incorporate strong inference rules, do not gain the extra pruning power available when strong representations are combined with strong inference.

For state-based SAT search to truly shine, an equally powerful learning scheme must be discovered.

Developing new state-based SAT implementations of domain specific constraints is worthwhile. Versatile data-structures such as SMURFs have the potential to go far beyond their current state of use. So far, research has focused on Boolean SMURFs even though they could easily be used to represent constraints in other domains, for example, monomorphic arithmetic functions commonly used in cryptographic software verification [62]. Each new SMURF type can capture different problem structure, facilitating the development of new heuristics that capitalize on global semantic

relationships between constraints.

More speculative directions include tapping into the power of extended resolution by creating new variables to represent active Smurf nodes and using them akin to extended resolution during witness clause generation. Also, research into safe-assignments [143], a form of autarky generalized to BDDs, could be beneficial to Smurfs because safe-assignments can be memoized on Smurf transitions, potentially providing more cuts to the search space.

Bibliography

[1] S. B. Akers, *Binary decision diagrams*, IEEE Trans. Computers, 27 (1978), pp. 509–516.

[2] F. A. Aloul, I. L. Markov, and K. A. Sakallah, *Faster SAT and smaller BDDs via common function structure*, in ICCAD, 2001, pp. 443–448.

[3] ——, *Improving the efficiency of circuit-to-BDD conversion by gate and input ordering*, in ICCD, IEEE Computer Society, 2002, pp. 64–69.

[4] ——, *Force: a fast and easy-to-implement variable-ordering heuristic*, in ACM Great Lakes Symposium on VLSI, ACM, 2003, pp. 116–119.

[5] ——, *Mince: A static global variable-ordering heuristic for sat search and bdd manipulation*, J. UCS, 10 (2004), pp. 1562–1596.

[6] F. A. Aloul, M. N. Mneimneh, and K. A. Sakallah, *ZBDD-based backtrack search SAT solver*, in IWLS, 2002, pp. 131–136.

[7] A. Anuchitanukul, Z. Manna, and T. E. Uribe, *Differential bdds*, in Computer Science Today, J. van Leeuwen, ed., vol. 1000 of Lecture Notes in Computer Science, Springer, 1995, pp. 218–233.

[8] K. Apt, *Principles of constraint programming*, Cambridge University Press, 2003.

[9] A. Atserias, P. G. Kolaitis, and M. Y. Vardi, *Constraint propagation as a proof system*, in CP, M. Wallace, ed., vol. 3258 of Lecture Notes in Computer Science, Springer, 2004, pp. 77–91.

[10] G. Audemard and L. Simon, *Predicting learnt clauses quality in modern SAT solvers*, in IJCAI, C. Boutilier, ed., 2009, pp. 399–404.

[11] ——, *Glucose: a solver that predicts learnt clauses quality*, (2010).

[12] F. Bacchus and T. Walsh, *A non-cnf dimacs style*.

[13] O. Bailleux and Y. Boufkhad, *Efficient CNF encoding of boolean cardinality constraints*, in CP, F. Rossi, ed., vol. 2833 of Lecture Notes in Computer Science, Springer, 2003, pp. 108–122.

[14] C. Baral, *Knowledge representation, reasoning and declarative problem solving*, Cambridge University Press, 2003.

[15] K. A. Bartlett, R. K. Brayton, G. D. Hachtel, R. M. Jacoby, C. R. Morrison, R. L. Rudell, A. L. Sangiovanni-Vincentelli, and A. R. Wang, *Multi-level logic minimization using implicit don't cares*, IEEE Trans. on CAD of Integrated Circuits and Systems, 7 (1988), pp. 723–740.

[16] P. Beame, *Proof complexity*. http://www.cs.toronto.edu/∼toni/Courses/Proofcomplexity/Papers/paul-lectures.ps, 2000.

[17] P. Beame, H. A. Kautz, and A. Sabharwal, *Understanding the power of clause learning*, in IJCAI, G. Gottlob and T. Walsh, eds., Morgan Kaufmann, 2003, pp. 1194–1201.

[18] ——, *Towards understanding and harnessing the potential of clause learning*, J. Artif. Intell. Res. (JAIR), 22 (2004), pp. 319–351.

[19] N. Beldiceanu and E. Contejean, *Introducing global constraints in CHIP*, J. Math. Comp. Model., 20 (1994), pp. 97–123.

[20] Y. Ben-Haim, A. Ivrii, O. Margalit, and A. Matsliah, *Perfect hashing and cnf encodings of cardinality constraints*, in Theory and Applications of Satisfiability Testing – SAT 2012, A. Cimatti and R. Sebastiani, eds., vol. 7317 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2012, pp. 397–409.

[21] D. L. Berre, O. Roussel, and L. Simon, *Sat competition*.

[22] C. Bessiere, ed., *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, vol. 4741 of Lecture Notes in Computer Science, Springer, 2007.

[23] A. Biere, *Aiger format*. Available from `http://fmv.jku.at/aiger/FORMAT.aiger`.

[24] ——, *Picosat essentials*, Journal on Satisfiability, Boolean Modeling and Computation (JSAT), 4 (2008), pp. 75–97.

[25] ——, *Lingeling, plingeling, picosat and precosat at sat race 2010.*, FMV Report Series Technical Report, Johannes Kepler University, Linz, Austria, 10 (2010).

[26] ——, *Ebddres version 1.1*. Available from http://fmv.jku.at/ebddres/index.html, 2011.

[27] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, *Symbolic model checking using SAT procedures instead of BDDs*, in DAC, 1999, pp. 317–320.

[28] A. Biere and C. P. Gomes, eds., *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, vol. 4121 of Lecture Notes in Computer Science, Springer, 2006.

[29] P. Bjesse, T. Leonard, and A. Mokkedem, *Finding bugs in an alpha microprocessor using satisfiability solvers*, in CAV, G. Berry, H. Comon, and A. Finkel, eds., vol. 2102 of Lecture Notes in Computer Science, Springer, 2001, pp. 454–464.

[30] M. Botinčan, M. Parkinson, and W. Schulte, *Separation logic verification of c programs with an smt solver*, Electron. Notes Theor. Comput. Sci., 254 (2009), pp. 5–23.

[31] R. Boute, *The binary decision machine as programmable controller*, Euromicro Newsletter, 2 (1976), pp. 16 – 22.

[32] R. S. Boyer and W. A. Hunt, Jr., *Function memoization and unique object representation for acl2 functions*, in Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications, ACL2 '06, New York, NY, USA, 2006, ACM, pp. 81–89.

[33] K. S. Brace, R. L. Rudell, and R. E. Bryant, *Efficient implementation of a BDD package*, in DAC, 1990, pp. 40–45.

[34] S. Brand, N. Narodytska, C.-G. Quimper, P. J. Stuckey, and T. Walsh, *Encodings of the sequence constraint*, in Bessiere [22], pp. 210–224.

[35] R. E. Bryant, *Graph-based algorithms for boolean function manipulation*, IEEE Trans. Computers, 35 (1986), pp. 677–691.

[36] ——, *On the complexity of vlsi implementations and graph representations of boolean functions with application to integer multiplication*, IEEE Trans. Computers, 40 (1991), pp. 205–213.

[37] ——, *Symbolic boolean manipulation with ordered binary-decision diagrams*, ACM Comput. Surv., 24 (1992), pp. 293–318.

[38] D. L. Bueno, *funcsat*. Available from `http://web.eecs.umich.edu/~dlbueno/`.

[39] ——, *funsat-0.6.2: A modern dpll-style sat solver*. Available from `http://hackage.haskell.org/package/funsat`.

[40] J. R. Burch, E. M. Clarke, and D. E. Long, *Symbolic model checking with partitioned transistion relations*, in VLSI, 1991, pp. 49–58.

[41] B. Chambers, P. Manolios, and D. Vroon, *Faster SAT solving with better CNF generation*, in DATE, IEEE, 2009, pp. 1590–1595.

[42] P. Chauhan, E. M. Clarke, S. Jha, J. H. Kukula, T. R. Shiple, H. Veith, and D. Wang, *Non-linear quantification scheduling in image computation*, in ICCAD, 2001, pp. 293–.

[43] J. Chen, *Building a hybrid sat solver via conflict-driven, look-ahead and xor reasoning techniques*, in Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing, SAT '09, Berlin, Heidelberg, 2009, Springer-Verlag, pp. 298–311.

[44] W. Chen and W. Zhang, *A direct construction of polynomial-size OBDD proof of pigeon hole problem*, Inf. Process. Lett., 109 (2009), pp. 472–477.

[45] A. Cimatti and R. B. Jones, eds., *Formal Methods in Computer-Aided Design, FMCAD 2008, Portland, Oregon, USA, 17-20 November 2008*, IEEE, 2008.

[46] S. A. Cook, *A short proof of the pigeon hole principle using extended resolution*, SIGACT News, 8 (1976), pp. 28–32.

[47] S. A. Cook and R. A. Reckhow, *The relative efficiency of propositional proof systems*, J. Symb. Log., 44 (1979), pp. 36–50.

[48] O. Coudert and J. C. Madre, *A unified framework for the formal verification of sequential circuits*, in ICCAD, 1990, pp. 126–129.

[49] J. M. Crawford and A. B. Baker, *Experimental results on the application of satisfiability algorithms to scheduling problems*, in AAAI, 1994, pp. 1092–1097.

[50] J. Daemen and V. Rijmen, *The design of Rijndael: AES-the advanced encryption standard*, Springer, 2002.

[51] R. F. Damiano and J. H. Kukula, *Checking satisfiability of a conjunction of BDDs*, in DAC, ACM, 2003, pp. 818–823.

[52] A. Darwiche, *Recursive conditioning*, Artif. Intell., 126 (2001), pp. 5–41.

[53] A. Darwiche and M. Hopkins, *Using recursive decomposition to construct elimination orders, jointrees, and dtrees*, in ECSQARU, S. Benferhat and P. Besnard, eds., vol. 2143 of Lecture Notes in Computer Science, Springer, 2001, pp. 180–191.

[54] M. Davis, G. Logemann, and D. W. Loveland, *A machine program for theorem-proving*, Commun. ACM, 5 (1962), pp. 394–397.

[55] M. Davis and H. Putnam, *A computing procedure for quantification theory*, J. ACM, 7 (1960), pp. 201–215.

[56] R. Dechter, *Bucket elimination: A unifying framework for probabilistic inference*, in UAI, E. Horvitz and F. V. Jensen, eds., Morgan Kaufmann, 1996, pp. 211–219.

[57] R. Dechter, *Constraint processing*, Morgan Kaufmann, 2003.

[58] H. Dixon, *Automating Pseudo-Boolean Inference within a DPLL Framework*, PhD thesis, University of Oregon, December 2004.

[59] M. Dransfield, V. Marek, and M. Truszczyński, *Satisfiability and computing van der waerden numbers*, Theory and Applications of Satisfiability Testing, (2004), pp. 288–289.

[60] N. Eén, *SAT Based Model Checking*, PhD thesis, Chalmers University of Technology, 2005.

[61] J. Ellson, E. Gansner, E. Koutsofios, S. North, and G. Woodhull, *Graphviz*. Available from `http://www.research.att.com/sw/tools/graphviz`, 2012.

[62] L. Erkök and J. Matthews, *Pragmatic equivalence and safety checking in cryptol*, in Proceedings of the 3rd workshop on Programming languages meets program verification, ACM, 2009, pp. 73–82.

[63] P. Ferraris and V. Lifschitz, *Weight constraints as nested expressions*, Theory Pract. Log. Program., 5 (2005), pp. 45–74.

[64] J. V. Franco, M. Dransfield, W. M. Vanfleet, and J. S. Schlipf, *State-based propositional satisfiability solver*. United States Patent 10/164,203, 2005.

[65] J. V. Franco, M. Kouril, J. S. Schlipf, J. Ward, S. A. Weaver, M. Dransfield, and W. M. Vanfleet, *SBSAT: a state-based, BDD-based satisfiability solver*, in SAT, E. Giunchiglia and A. Tacchella, eds., vol. 2919 of Lecture Notes in Computer Science, Springer, 2003, pp. 398–410.

[66] J. V. Franco, M. Kouril, J. S. Schlipf, S. A. Weaver, M. Dransfield, and W. M. Vanfleet, *Function-complete lookahead in support of efficient SAT search heuristics*, J. UCS, 10 (2004), pp. 1655–1695.

[67] Z. Fu and S. Malik, *Extracting logic circuit structure from conjunctive normal form descriptions*, in VLSI Design, IEEE Computer Society, 2007, pp. 37–42.

[68] G. Gange, V. Lagoon, and P. J. Stuckey, *Fast set bounds propagation using BDDs*, in ECAI, M. Ghallab, C. D. Spyropoulos, N. Fakotakis, and N. M. Avouris, eds., vol. 178 of Frontiers in Artificial Intelligence and Applications, IOS Press, 2008, pp. 505–509.

[69] A. V. Gelder, *Generalizations of watched literals for backtracking search*, in AMAI, 2002.

[70] ——, *Improved conflict-clause minimization leads to improved propositional proof traces*, in Kullmann [101], pp. 141–146.

[71] I. P. Gent, *Arc consistency in SAT*, in ECAI, F. van Harmelen, ed., IOS Press, 2002, pp. 121–125.

[72] C. P. Gomes, B. Selman, N. Crato, and H. Kautz, *Heavy-tailed phenomena in satisfiability and constraint satisfaction problems*, J. Autom. Reason., 24 (2000), pp. 67–100.

[73] S. Gopalakrishnan, V. Durairaj, and P. Kalla, *Integrating CNF and BDD based SAT solvers*, in HLDVT '03: Proceedings of the Eighth IEEE International Workshop on High-Level Design Validation and Test Workshop, Washington, DC, USA, 2003, IEEE Computer Society, p. 51.

[74] J. F. Groote, *Hiding propositional constants in BDDs*, Formal Methods in System Design, 8 (1996), pp. 91–96.

[75] J. F. Groote and H. Zantema, *Resolution and binary decision diagrams cannot simulate each other polynomially*, Discrete Applied Mathematics, 130 (2003), pp. 157–171.

[76] G. Hagen and C. Tinelli, *Scaling up the formal verification of lustre programs with smt-based techniques*, in Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design, FMCAD '08, Piscataway, NJ, USA, 2008, IEEE Press, pp. 15:1–15:9.

[77] A. Haken, *The intractability of resolution*, Theor. Comput. Sci., 39 (1985), pp. 297–308.

[78] P. Hawkins and P. J. Stuckey, *A hybrid bdd and sat finite domain constraint solver*, in IN: PROCEEDINGS OF PADL., Springer-Verlag, 2006, pp. 103–117.

[79] M. Heule, M. Dufour, J. van Zwieten, and H. van Maaren, *March_eq: Implementing additional reasoning into an efficient look-ahead sat solver*, in Theory and Applications of Satisfiability Testing, vol. 3542 of Lecture Notes in Computer Science, 2005, pp. 345–359.

[80] Y. Hong, P. A. Beerel, J. R. Burch, and K. L. McMillan, *Safe BDD minimization using don't cares*, in DAC, 1997, pp. 208–213.

[81] H. Hoos and D. G. Mitchell, eds., *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings*, 2004.

[82] J. Huang, *Mup: a minimal unsatisfiability prover*, in ASP-DAC, T. Tang, ed., ACM Press, 2005, pp. 432–437.

[83] ——, *The effect of restarts on the efficiency of clause learning*, in IJCAI, M. M. Veloso, ed., 2007, pp. 2318–2323.

[84] J. Huang and A. Darwiche, *Toward good elimination orders for symbolic SAT solving*, in ICTAI, IEEE Computer Society, 2004, pp. 566–573.

[85] W. A. Hunt and S. Swords, *Centaur technology media unit verification*, in CAV, A. Bouajjani and O. Maler, eds., vol. 5643 of Lecture Notes in Computer Science, Springer, 2009, pp. 353–367.

[86] H. Jain and E. M. Clarke, *Efficient SAT solving for non-clausal formulas using DPLL, graphs, and watched cuts*, in DAC, ACM, 2009, pp. 563–568.

[87] S.-W. Jeong and F. Somenzi, *A new algorithm for the binate covering problem and its application to the minimization of boolean relations*, in ICCAD, 1992, pp. 417–420.

[88] R. G. Jeroslow and J. Wang, *Solving propositional satisfiability problems*, Ann. Math. Artif. Intell., 1 (1990), pp. 167–187.

[89] H. Jin, M. Awedh, and F. Somenzi, *Circus: A satisfiability solver geared towards bounded model checking*, in CAV, R. Alur and D. Peled, eds., vol. 3114 of Lecture Notes in Computer Science, Springer, 2004, pp. 519–522.

[90] H. Jin, A. Kuehlmann, and F. Somenzi, *Fine-grain conjunction scheduling for symbolic reachability analysis*, in TACAS, J.-P. Katoen and P. Stevens, eds., vol. 2280 of Lecture Notes in Computer Science, Springer, 2002, pp. 312–326.

[91] H. Jin and F. Somenzi, *Circus: A hybrid satisfiability solver*, in Hoos and Mitchell [81].

[92] D. S. Johnson, *Optimization algorithms for combinatorial problems*, Journal of Computer and System Sciences, 9 (1974), pp. 256–278.

[93] T. Jussila, C. Sinz, and A. Biere, *Extended resolution proofs for symbolic sat solving with quantification*, in Biere and Gomes [28], pp. 54–60.

[94] H. A. Kautz, E. Horvitz, Y. Ruan, C. P. Gomes, and B. Selman, *Dynamic restart policies*, in AAAI/IAAI, 2002, pp. 674–681.

[95] M. Kouril and J. V. Franco, *Resolution tunnels for improved SAT solver performance*, in SAT, F. Bacchus and T. Walsh, eds., vol. 3569 of Lecture Notes in Computer Science, Springer, 2005, pp. 143–157.

[96] M. Kouril and J. L. Paul, *The van der Waerden number W(2, 6) is 1132*, Experimental Mathematics, 17 (2008), pp. 53–61.

[97] D. Kroening and O. Strichman, *Decision Procedures: An Algorithmic Point of View*, Springer Publishing Company, Incorporated, 1st ed., 2010.

[98] A. Kuehlmann, *Dynamic transition relation simplification for bounded property checking*, in ICCAD, IEEE Computer Society / ACM, 2004, pp. 50–57.

[99] A. Kuehlmann and F. Krohm, *Equivalence checking using cuts and heaps*, in DAC, 1997, pp. 263–268.

[100] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, *Robust boolean reasoning for equivalence checking and functional property verification*, IEEE Trans. on CAD of Integrated Circuits and Systems, 21 (2002), pp. 1377–1394.

[101] O. Kullmann, ed., *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, vol. 5584 of Lecture Notes in Computer Science, Springer, 2009.

[102] D. Le Berre and A. Parrain, *The sat4j library, release 2.2 system description*, Journal on Satisfiability, Boolean Modeling and Computation, 7 (2010), pp. 59–64.

[103] C. Y. Lee, *Representation of switching circuits by binary-decision programs*, Bell Systems Technical Journal, 38 (1959), pp. 985–999.

[104] J. Lewis and B. Martin, *Cryptol: High assurance, retargetable crypto development and validation*, in Military Communications Conference, 2003. MILCOM'03. 2003 IEEE, vol. 2, IEEE, 2003, pp. 820–825.

[105] F. Lu, L.-C. Wang, K.-T. Cheng, and R. C.-Y. Huang, *A circuit SAT solver with signal correlation guided learning*, in DATE, IEEE Computer Society, 2003, pp. 10892–10897.

[106] I. Lynce and J. P. Marques-Silva, *Efficient data structures for backtrack search SAT solvers*, Ann. Math. Artif. Intell., 43 (2005), pp. 137–152.

[107] ——, *SAT in bioinformatics: Making the case with haplotype inference*, in Biere and Gomes [28], pp. 136–141.

[108] V. W. Marek, *Introduction to Mathematics of Satisfiability*, CRC Press, 2009.

[109] J. P. Marques-Silva and I. Lynce, *Towards robust CNF encodings of cardinality constraints*, in Bessiere [22], pp. 483–497.

[110] J. P. Marques-Silva, I. Lynce, and S. Malik, *Conflict-driven clause learning SAT solvers*, in Handbook of Satisfiability, A. Biere, M. Heule, H. van Maaren, and T. Walsh, eds., vol. 185 of Frontiers in Artificial Intelligence and Applications, IOS Press, 2009, pp. 131–153.

[111] J. P. Marques-Silva and K. A. Sakallah, *Grasp - a new search algorithm for satisfiability*, in ICCAD, 1996, pp. 220–227.

[112] K. L. McMillan, *Interpolation and SAT-based model checking*, in CAV, W. A. H. Jr. and F. Somenzi, eds., vol. 2725 of Lecture Notes in Computer Science, Springer, 2003, pp. 1–13.

[113] G. H. Mealy, *A method to synthesizing sequential circuits*, Bell Systems Technical Journal, 34 (1955), pp. 1045–1079.

[114] C. Meinel, H. Sack, and V. Schillings, *Visbdd–a web-based visualization framework for obdd algorithms*, in IWLS, 2002, pp. 385–390.

[115] R. Meolic, *Biddy—a multi-platform academic bdd package*, Journal of Software, 7 (2012), pp. 1358–1366.

[116] I. Mironov and L. Zhang, *Applications of SAT solvers to cryptanalysis of hash functions*, in Biere and Gomes [28], pp. 102–115.

[117] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, *Chaff: Engineering an efficient SAT solver*, in DAC, ACM, 2001, pp. 530–535.

[118] A. Nadel, *Understanding and Improving a Modern SAT Solver*, PhD thesis, Tel Aviv University, 2009.

[119] Y. Novikov and R. Brinkmann, *Foundations of hierarchical SAT-solving*, Proc. Int'l Workshop Boolean Problems, (2004).

[120] R. Ostrowski, É. Grégoire, B. Mazure, and L. Sais, *Recovering and exploiting structural knowledge from CNF formulas*, in CP, P. V. Hentenryck, ed., vol. 2470 of Lecture Notes in Computer Science, Springer, 2002, pp. 185–199.

[121] V. Paruthi, C. Jacobi, and K. Weber, *Efficient symbolic simulation via dynamic scheduling, don't caring, and case splitting*, in CHARME, D. Borrione and W. J. Paul, eds., vol. 3725 of Lecture Notes in Computer Science, Springer, 2005, pp. 114–128.

[122] V. Paruthi and A. Kuehlmann, *Equivalence checking combining a structural SAT-solver, BDDs, and simulation*, in ICCD, 2000, pp. 459–464.

[123] M. R. Prasad, A. Biere, and A. Gupta, *A survey of recent advances in SAT-based formal verification*, STTT, 7 (2005), pp. 156–173.

[124] R. Rudell, *Dynamic variable ordering for ordered binary decision diagrams*, in ICCAD, M. R. Lightner and J. A. G. Jess, eds., IEEE Computer Society, 1993, pp. 42–47.

[125] L. Ryan, *Efficient algorithms for clause-learning SAT solvers*, Master's thesis, Simon Fraser University, 2004.

[126] R. Sebastiani, *Lazy satisability modulo theories*, JSAT, 3 (2007), pp. 141–224.

[127] T. R. Shiple, R. Hojati, A. L. Sangiovanni-Vincentelli, and R. K. Brayton, *Heuristic minimization of BDDs using don't cares*, in DAC, 1994, pp. 225–231.

[128] A. Slobodová, *Formal verification of hardware support for advanced encryption standard*, in Cimatti and Jones [45], pp. 1–4.

[129] E. W. Smith and D. L. Dill, *Automatic formal verification of block cipher implementations*, in Cimatti and Jones [45], pp. 1–7.

[130] F. Somenzi, *Colorado University Decision Diagram package*. Available from `http://vlsi.colorado.edu/~fabio/CUDD/`.

[131] ——, *Binary decision diagrams*, in Calculation System Design, M. Broy and R. Steinbruggen, eds., vol. 173, 1999, pp. 303–366.

[132] M. Soos, K. Nohl, and C. Castelluccia, *Extending SAT solvers to cryptographic problems*, in Kullmann [101], pp. 244–257.

[133] I. Spence, *sgen1: A generator for small but difficult satisability instances*. Available from `www.cs.qub.ac.uk/~i.spence/sgen/sgen1descr.pdf`.

[134] T. Stanion and C. Sechen, *Boolean division and factorization using binary decision diagrams*, Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 13 (1994), pp. 1179 –1184.

[135] F. S. Taylor, *Quintic Abelian Fields*, PhD thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, December 1997.

[136] C. Thiffault, F. Bacchus, and T. Walsh, *Solving non-clausal formulas with dpll search*, in Principles and Practice of Constraint Programming – CP 2004, vol. 3258 of Lecture Notes in Computer Science, 2004, pp. 663–678.

[137] G. S. Tseitin., *On the complexity of derivation in propositional calculus*, Automation of Reasoning 2: Classical Papers on Computational Logic, 1967-1970, (1983), pp. 466–483.

[138] M. N. Velev and R. E. Bryant, *Effective use of boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors*, Journal of Symbolic Computation, 35 (2003), pp. 73 – 106.

[139] T. Villa, T. Kam, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, *Explicit and implicit algorithms for binate covering problems*, IEEE Trans. on CAD of Integrated Circuits and Systems, 16 (1997), pp. 677–691.

[140] S. A. Weaver, *BDD Visualizer*. Available from `www.cs.uc.edu/~weaversa/BDD_Visualizer.html`.

[141] S. A. Weaver and J. V. Franco, *A CNF analogue to strengthening*, The Morehead Electronic Journal of Applicable Mathematics, 3 (2006).

[142] S. A. Weaver, J. V. Franco, V. W. Marek, and M. Kouril, *Workshop on satisfiability: Assessing the progress*, tech. rep., U.S. Department of Defense, http://gauss.ececs.uc.edu/franco_files/dodreport.pdf, 2008.

[143] S. A. Weaver, J. V. Franco, and J. S. Schlipf, *Extending existential quantification in conjunctions of BDDs*, JSAT, 1 (2006), pp. 89–110.

[144] S. A. Weaver and M. Kouril, *SBSAT*. Available from `www.cs.uc.edu/~weaversa/SBSAT.html`.

[145] I. Wegener, *The complexity of symmetric boolean functions*, in Computation Theory and Logic, E. Börger, ed., vol. 270 of Lecture Notes in Computer Science, Springer, 1987, pp. 433–442.

[146] B. Yang, *BDD trace driver*. Available from `http://www.cs.cmu.edu/~bwolen/software/`.

[147] B. Yang, Y.-A. Chen, R. E. Bryant, and D. R. O'Hallaron, *Space- and time-efficient BDD construction via working set control*, in ASP-DAC, 1998, pp. 423–432.

[148] R. Zabih and D. A. McAllester, *A rearrangement search strategy for determining propositional satisfiability*, in AAAI, 1988, pp. 155–160.

[149] H. Zhang, D. Li, and H. Shen, *A SAT based scheduler for tournament schedules*, in Hoos and Mitchell [81].

[150] L. Zhang and S. Malik, *Searching for truth: techniques for satisfiability of boolean formulas*, PhD thesis, Princeton University, 2003.

[151] Q. Zhu, N. Kitchen, A. Kuehlmann, and A. L. Sangiovanni-Vincentelli, *SAT sweeping with local observability don't-cares*, in DAC, E. Sentovich, ed., ACM, 2006, pp. 229–234.