The Dissertation Committee for Nathan David Wetzler
certifies that this is the approved version of the following dissertation:

Efficient, Mechanically-Verified Validation of Satisfiability Solvers

Committee:

Warren A. Hunt, Jr., Supervisor

Marijn J. H. Heule, Co-Supervisor

Armin Biere

Vladimir Lifschitz

J Strother Moore

Vijaya Ramachandran

Efficient, Mechanically-Verified Validation of Satisfiability Solvers

by

Nathan David Wetzler, B.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2015

Dedicated to my parents.

Acknowledgments

This dissertation would not have been possible without the support of my advisors, colleagues, friends, and family. First and foremost, I would like to thank my supervisors Warren Hunt, Jr. and Marijn Heule. Much of the work in this dissertation was done in collaboration with Marijn Heule, and it was his ideas and encouragement that spurred this research. He also explained the benefit of this work to the SAT community and helped ensure its widespread adoption. Warren Hunt, Jr. has served as my advisor for six years, and during this time he has been a source of moral and technical support (not to mention financial support). I cannot imagine completing this dissertation under different tutelage.

I want to thank my colleagues for their time and support over the years. I will forever be in debt to Shilpi Goel, Matt Kaufmann, and David Rager for their sage advice when working with ACL2. They always "had a minute" to listen to my ideas and help solve my technical problems. Their wisdom is spread throughout this work. I would also like to thank my dissertation committe members Armin Biere, Vladimir Lifschitz, J Strother Moore, and Vijaya Ramachandran for their time and expertise in this area. I would like to acknowledge my undergraduate research advisors Chaim Goodman-Strauss and Francine Blanchet-Sadri; I believe they are primarily responsible for my acceptance into such a prestigious university and computer science program.

I could not have completed this degree without my friends Travis Crone, Jeff Dorman, and Jay Gergen. They were always there to help me readjust after a long day of working on mechanical proofs. Finally, I want to acknowledge my loving parents, David and Debbie Wetzler, and my sister, Kristen Wetzler. They have been my emotional support structure throughout this entire process, and there are no words that can describe how grateful I am to them.

The work presented in this dissertation was supported largely by the Defense Advanced Research Projects Agency (DARPA) grant N66001-10-2-4087 on "Code Verification for Practical Machine Architectures" and was also supported by National Science Foundation (NSF) grants CNS-0916772 on "Producing Certifiable Hardware from Behavioral Synthesis" and CNS-0910913 on "A Formal Platform for Analyzing Internet Routing".

Efficient, Mechanically-Verified Validation of Satisfiability Solvers

Publication No. ______________

Nathan David Wetzler, Ph.D.
The University of Texas at Austin, 2015

Supervisors: Warren A. Hunt, Jr.
Marijn J. H. Heule

Satisfiability (SAT) solvers are commonly used for a variety of applications, including hardware verification, software verification, theorem proving, debugging, and hard combinatorial problems. These applications rely on the efficiency and correctness of SAT solvers. When a problem is determined to be unsatisfiable, how can one be confident that a SAT solver has fully exhausted the search space? Traditionally, unsatisfiability results have been expressed using resolution or clausal proof systems. Resolution-based proofs contain perfect reconstruction information, but these proofs are extremely large and difficult to emit from a solver. Clausal proofs rely on rediscovery of inferences using a limited number of techniques, which typically takes several orders of magnitude longer than the solving time. Moreover, neither of these proof systems has been able to express contemporary solving techniques such as bounded variable addition. This combination of issues has left SAT solver authors unmotivated to produce proofs of unsatisfiability.

The work from this dissertation focuses on validating satisfiability solver output in the unsatisfiability case. We developed a new clausal proof format called DRAT that facilitates compact proofs that are easier to emit and capable of expressing all contemporary solving and preprocessing techniques. Furthermore, we implemented a validation utility called DRAT-trim that is able to validate proofs in a time similar to that of the discovery time. The DRAT format has seen widespread adoption in the SAT community and the DRAT-trim utility was used to validate the results of the 2014 SAT Competition.

DRAT-trim uses many advanced techniques to realize its performance gains, so why should the results of DRAT-trim be trusted? Mechanical verification enables users to model programs and algorithms and then prove their correctness with a proof assistant, such as ACL2. We designed a new modeling technique for ACL2 that combines efficient model execution with an agile and convenient theory. Finally, we used this new technique to construct a fast, mechanically-verified validation tool for proofs of unsatisfiability. This research allows SAT solver authors and users to have greater confidence in their results and applications by ensuring the validity of unsatisfiability results.

Contents

Chapter 1

Introduction

Satisfiability (SAT) solvers [11] are commonly used for a variety of applications, including hardware verification [10, 70, 34, 59, 5, 47, 25], software verification [17, 28], theorem proving [24], debugging [16], and hard combinatorial problems [52, 51, 18]. These applications rely on the efficiency of SAT solvers to decide large Boolean problems and to provide the correct results, but solvers are treated as black-box utilities. Solvers are often used not only to find a solution for a Boolean formula, but also to make a claim that no solution exists. In most applications of SAT, a solution represents an error in a system; and, more importantly, the absence of a solution represents the absence of errors. If a solution is reported for a given formula, one can check the solution linearly in the size of the formula. But when no solution is reported to exist, how can one be confident that a SAT solver has fully exhausted the search space? This is complicated by the fact that state-of-the-art solvers employ a large array of complex techniques [12, 55, 46, 45] that are used to maximize efficiency. Errors may be introduced at a conceptual level [46] as well as an implementation level. Formal verification is an approach to detect errors and to assure that the results produced by SAT solvers are correct.

One method of assurance is to apply formal verification to the SAT solver itself. This involves modeling a SAT solver, specifying the desired behavior, and using a tool—such as a theorem prover—to show that the model meets its specification [54, 56, 57, 69, 65]. The benefit of such a direct approach is that no post-processing is needed after the solver halts. While the formal verification of a SAT solver is a noble endeavor, there are many problems with this approach. SAT solvers are constantly evolving, and each new feature requires a modification to the verification effort. This leads to a delicate balance between efficiency and verification.

Another approach is to validate the output of a SAT solver [33, 81, 75, 9]. A *proof trace* is a sequence of clauses that are claimed to be *redundant* with respect to a given formula. If a SAT solver reports that a given formula is unsatisfiable, it can provide a proof trace that can be checked by a smaller, trusted/verified program called a proof checker. If the empty clause is shown to be redundant for a given formula, the formula is unsatisfiable and the proof trace is called a *refutation*. Ideally, a proof format should facilitate proofs that are: compact, easy to obtain, efficient to validate, expressive enough to capture techniques used in state-of-the-art solvers, and validated by a simple proof checker implementation. In this way, one can focus mechanical verification efforts on a simple proof checker that employs a limited number of techniques. This avoids the need to verify a variety of solving techniques, and a single proof checker can validate the results of multiple solvers, making the approach modular with respect to the choice of solver.

Modeling a SAT proof checker and developing a *proof of correctness* for the model provides assurance that proof traces can be validated, but if the model is non-executable or inefficient, then it is of little use for applications in science and industry.

Instead, it is possible to create an efficient model that is also mechanically verified. The verification task is much more difficult, however, when design decisions favor efficiency over clarity. Proper abstractions can help ease the burden of verification and pave the way for efficient, mechanically-verified software.

In the remainder of this chapter, there is a description of the importance of this problem to researchers in mathematical science, industry, and the SAT community (Section 1.1). Then the contributions of this work (Section 1.2) in SAT proof formats, efficient proof validation, efficient formal modeling, and verified proof validation are presented.

1.1 Motivation

Satisfiability solvers have numerous applications in science (Section 1.1.1) and industry (Section 1.1.2). The SAT research community pushes for efficient and varied solving techniques, but errors are sometimes introduced in the development process (Section 1.1.3).

1.1.1 Applications in Science

The correctness of SAT solvers is important for new developments in combinatorial science where SAT solvers are often used to tighten theoretical bounds. For example, the Erdős Discrepancy Conjecture (EDC) [31] has been an open problem in mathematics for over eighty years. It states that for all positive integers C and infinite sequences (x_n) with values in the set $\{+1, -1\}$, there exists a subsequence $x_d, x_{2d}, \ldots, x_{kd}$ for positive integers k and d such that the absolute value of the sum of all elements of the subsequence is greater than C. The EDC holds for the case where $C = 1$, but the problem was still open (and widely studied) for any value of C greater than 1. Konev and Lisitsa [51] recently (2014) were able develop a proof for the EDC where $C = 2$ using an encoding to satisfiability. Their proof relied on an unsatisfiability result that was emitted by a state-of-the-art SAT solvers Lingeling and Glucose. In order to provide confidence in their result, they chose to validate the unsatisfiability result with the methods described in this dissertation.

SAT solvers have been used to compute van der Waerden numbers from Ramsey theory [35]. The van der Waerden theorem states that given natural numbers k and l, there exists a smallest natural number $n = W(k, l)$ such that every k-coloring of the numbers $\{1, 2, \ldots, n\}$ contains an arithmetic progression (an equi-spaced sequence) of length l in one of the colors. For a triple (k, l, m), Dransfield et al. [29] described an encoding to a propositional formula in conjunctive normal form (CNF) which is satisfiable if and only if $W(k, l) > m$. This procedure can be used to improve lower bounds for van der Waerden numbers by progressively checking satisfiability of CNF formulas. However, upper bounds are established by showing unsatisfiability of a formula. If the SAT solver contains an error, part of the search space may be skipped leading to an incorrect upper bound. In 2004, only five van der Waerden numbers had been claimed correct. Since then, many lower bounds have been improved with SAT, and a few new numbers are claimed to exist (e.g., $W(2, 6)$ is claimed to be 1132 [52]).

One application of satisfiability technology to theoretical computer science is minimal sorting networks [14, 50, 4]. A sorting network is an algorithm that can sort n values using k fixed comparisons. One can visualize the problem as a series of wires that transfer data values with "crossbars" connecting the wires, performing comparisons, and swapping wire values based on the result of the comparison. A sorting network of size n is considered minimal with respect to either the number of comparisons k, called "minimal size," or the number of layers d, called "minimal depth." The k values for minimal size sorting networks for $n \geq 9$ have been open problems for over sixty years. In 2014, Codish et al. [18] proved that the minimal size sorting network is $k = 25$ for $n = 9$ and $k = 29$ for $n = 10$ using a satisfiability encoding and SAT solver CryptoMiniSAT. Again, their construction depended on the correctness of an unsatisfiability result. To substantiate the solver's claims, they developed their own proof format and used Prolog to validate the proof.

SAT solvers have also been applied to problems in the celluar automaton known as Conway's Game of Life [19]. In this automaton, cells are designated "alive" or "dead", and four rules decide the state of each cell in the next generation based on the neighboring population. One problem in the Game of Life is discovering a Garden of Eden—a pattern that can never exist outside of an initial configuration. Hartman et al. [38] described a procedure for encoding the existence of a preceding state as a CNF formula. Unsatisfiability of the formula implies that the configuration is a Garden of Eden. Using this procedure, they broke the Garden of Eden records for the smallest bounding box, fewest defined cells, and lowest density of alive cells. Again, the validity of their results depends on the correctness of their unsatisfiability claims.

These are just a few of the scientific areas seeing new developments that were made possible with state-of-the-art SAT solvers. Satisfiability has also been applied to many other combinatorial problems including discovery of new Ramsey numbers [32, 42] and Schur numbers [29]. In all of these cases, the unsatisfiability results are critical; an error in the computation of an unsatisfiability result would unsubstantiate claims. In Section 1.1.3, some alarming data is presented on the correctness of contemporary SAT solvers.

1.1.2 Applications in Industry

The presence of an error in a SAT solver can have a large impact on industrial applications. In hardware model checking, a satisfiable result often represents a logical error in some refinement of a circuit, but an unsatisfiable result represents the absence of errors in a design. The latter is far more important to industrial applications, and it is much harder to show the correctness of an unsatisfiable result.

For example, the formal verification group at Centaur Technology uses symbolic execution in the ACL2 theorem proving system to verify x86 microprocessor execution units [72]. Symbolic execution is performed by "bit-blasting" finite ACL2 theorems to Boolean formulas [24]. In the past, these formulas have been represented as binary decision diagrams (BDDs), but verification groups found that a CNF based-representation with state-of-the-art SAT solvers can outperform a BDD-based approach. The difference between these two methods, however, is trust. The BDD-based approach has been mechanically verified in ACL2, while the SAT-based approach needs a "trust tag" in ACL2. This adds a caveat to their verification efforts: the x86 execution unit is correct provided that the SAT solver did not make a mistake when making an unsatisfiability claim. In their most recent work, Davis and Swords [25] developed a SATlink mode for symbolic execution where they proved that the bit-blasting encoding to Boolean formula and the interpretation of the SAT solver result is correct. The only piece missing in this mode is the trust associated with unsatisfiability results.

In software verification, a bounded model checker for C and C++ programs called CBMC [17] is able to detect array bounds exceptions, buffer overflows, unsafe pointers, general exceptions, and also validate user-provided assertions. It can also check consistency of C/C++ programs with other languages such as Verilog. This tool is built on top of satisfiability modulo theories (SMT) solvers like Boolector and Z3 and SAT solvers like MiniSAT. For example, users can ask the tool if a given program contains a potential buffer overflow exception. This property is then encoded as a Boolean formula and passed to a SMT/SAT solver. An unsatisfiability result indicates that the program is devoid of this type of exception. If a SAT solver erroneously arrived at this result, then the assurance from CBMC could be incorrect.

1.1.3 Bugs in SAT Solvers

The goal of a SAT solver is to decide a complex problem, which can potentially require exponential running time, efficiently and correctly. The SAT solving community presses for faster solvers with more efficient execution; however, like all software, errors can be introduced during development. Some SAT solving techniques are complicated and require a great deal of expertise to understand why they are conceptually correct and how to efficiently implement them while maintaining this correctness.

Järvisalo et al. [46] described a subtle error in the blocked clause addition routine of a version of Lingeling, arguably the best solver currently available. In the initial implementation of this routine, clauses were deemed to be redundant with respect to the original input formula (i.e., irredundant clauses) and were added to a list of redundant clauses so that they might be used to generate further redundant clauses. The author of Lingeling failed to account for the fact that these added clauses, while redundant with respect to the input formula, were not shown to be redundant with respect to the previously added redundant clauses. To make matters worse, Lingeling was "experimentally correct" on millions of instances and used in industry for over a year and a half before a bug related to blocked clause addition was demonstrated with a small hand-coded formula. During this one and a half year period, Lingeling won several awards in SAT competitions. These types of bugs can be hard to detect and can persist through many versions before they are identified.

Lingeling is not alone, though. All solvers are susceptible to bugs, even winners of SAT solving competitions. Brummayer et. al. [15] describes the application of an automated negative testing technique, known as *fuzzing*, to a subset of the solvers submitted to the 2007 and 2009 SAT solving competitions. In their testing, they discovered defects in six solvers submitted to the 2007 competition and three solvers submitted to the 2009 competition. Most of these defects led to incorrect results. Moreover, the defective solvers included winners from several tracks of the competitions. Furthermore, one track of SAT competitions has been focused on "portfolio" solvers which employ many different SAT solvers in their execution. If a faulty solver can produce a result faster than other solvers, that makes it a likely candidate for a portfolio solver to select during runtime.

While the efficiency of solvers is evaluated by industry and yearly competitions, the correctness of solvers is more difficult to evaluate. Developers can use fuzzing techniques to try to discover errors in implementation; however, fuzzing techniques alone are insufficient to detect all errors (as illustrated by the bug described above). Satisfiability problems often have millions of solutions or zero solutions. Suppose an unsound technique is developed that discards some small percentage (say 10%) of the solutions in order to narrow the search space and decrease solving time. If a problem has exactly one solution, there is only a small percentage chance (10%) that this solution will be discarded and an error is detected.

1.2 Contributions

The work discussed in this dissertation focuses on validating satisfiability solver output in the unsatisfiability case. The four major contributions of this work involve the development of a suitable proof format for satisfiability solvers, the efficient verification of proofs of unsatisfiability for SAT instances, the development of an new data structure used for efficient formal modeling, and the mechanical verification of an efficient satisfiability proof checker.

We developed a new unsatisfiability proof format called DRAT (short for Deletion Resolution Asymmetric Tautology) that is extremely compact when compared to even the smallest resolution proof format. This format is easy to emit from state-of-the-art solvers, and it can express all contemporary solving and preprocessing techniques. The DRAT proof format has been widely adopted by the SAT community. In the 2014 SAT Competition, solvers submitted to the UNSAT tracks of the competition were required to emit proofs of unsatisfiability so that the results could be validated. Thus, the track was changed from traditional UNSAT tracks to certified UNSAT tracks. In each certified UNSAT track, 15 participating solvers used the DRAT proof format, or a derivative such as RUP or DRUP, including all top-tier solvers. The certified UNSAT track only occurred once before the 2013 SAT Competition, in 2005.[1]

We implemented an unsatisfiability proof checking tool called DRAT-trim to validate proof of unsatisfiability in the DRAT format. This tool is able to validate proofs in a time similar to that of the discovery time and with much less memory than resolution proof checkers. This is quite remarkable in that DRAT-trim does not contain the advanced algorithms and techniques used by state-of-the-art solvers. DRAT-trim was used to verify all results in the certified UNSAT track of the 2014 SAT Competition. Furthermore, DRAT-trim is able to emit trimmed formulas, optimized proofs, and resolution/dependency graphs. Researchers have already begun to integrate the output of DRAT-trim into MUS extraction tools [62, 7] and interpolation utilities [76, 36].

We designed a new ACL2 data structure called `farray` that combines the execution efficiency of ACL2 arrays with an efficient and convenient theory. This new

[1]Several voluntary certified UNSAT tracks were organized in 2005, 2007, 2009, and 2011, but they had little participation.

abstraction is able to provide linear-time access and updates to multiple "fields" that are located within a single array, similar to C structs on a heap. This allows an ACL2 user to perform convenient state-based modeling without sacrificing performance. The development of this data structure was invaluable to our work and we believe it will be adopted by other ACL2 users wishing to construct efficient models.

Finally, we mechanically verified a RAT proof checking algorithm using ACL2, and then showed that an array-based implementation of that algorithm is equivalent to the algorithm. The array-based implementation utilizes the `farray` data structure to represent assignments almost exactly as they are represented in DRAT-trim. This reduces the complexity of operations on assignments from linear time to constant time while maintaining a proof of correctness. The end result is a fast, mechanically-verified RAT proof checker.

Organization The remainder of this dissertation is divided into chapters. In Chapter 2, background material is reviewed and the state-of-the-art is discussed as it applies to satisfiability solvers, unsatisfiability proof formats, theorem proving, and mechanical-verification of SAT solvers and proof checkers. Related work appears in this chapter. In Chapter 3, the contributions of this work towards an expressive proof format and efficient proof validation utility are described in detail. In Chapter 4, a formal specification for unsatisfiability in the ACL2 theorem proving system is presented. In Chapter 5, an algorithm is developed for validating proofs in a new, expressive proof format. In Chapter 6, the soundness proof is constructed showing that the algorithm meets the unsatisfiability specification. In Chapter 7, a more efficient implementation of a proof checker is presented, including a new data structure designed to facilitate efficient models with convenient theories. In Chapter 8, the new implementation is proven to be equivalent to the original proof-checking algorithm. Finally, future work is proposed in Chapter 9 and conclusions are drawn in Chapter 10.

Chapter 2

State-of-the-Art

In this chapter, the state-of-the-art is presented as it relates to the satisfiability problem, SAT solvers, and mechanical verification. The satisfiability problem is defined in Section 2.1 along with notation used in the rest of this dissertation. The standard input format for satisfiablility instances is also described. Proofs of unsatisfiability are constructed of clauses that have been deemed redundant with respect to a Boolean formula. Various methods of establishing redundancy are detailed in Section 2.2. Contemporary SAT search, simplification, and inference techniques should be supported by any proof system. These techniques are explained in Section 2.3 and existing proof systems are depicted in in Section 2.4. The concept of mechanical verification is introduced in Section 2.5, and the related work of mechanically-verified SAT solvers and SAT proof checkers are chronicled in Sections 2.6 and 2.7.

2.1 The Satisfiability Problem

An introduction to the satisfiability problem and basic satisfiability terminology are presented in in Section 2.1.1 and the concrete representation of satisfiability instances is described in Section 2.1.2.

2.1.1 Preliminaries

The satisfiability (SAT) problem [11] asks if there is a truth assignment of Boolean values to Boolean variables for a formula such that the formula evaluates to TRUE. A Boolean variable x is associated with two *literals*, the positive literal x and the negative literal $\bar{x}$. A *clause* is a finite disjunction of literals and a *conjunctive normal form* (CNF) *formula* is a finite conjunction of clauses. For the purposes of this work, all formulas are assumed to be in CNF.

A *tautology* is a clause that contains the *conflicting* literals l and $\bar{l}$. The notation $x \in F$ denotes that variable x appears as either a positive literal or negative literal in formula F. Similarly, the notation $l \in F$ denotes that literal l appears in some clause in formula F. A *truth assignment* (hereafter called an *assignment*) for a formula F is a partial function that maps the literals of F to the Boolean values TRUE and FALSE. Assignments may be represented as a conjunction of (non-conflicting) literals or even a set of (non-conflicting) literals.

Example 1. The CNF formula $F_1 = ((a \vee b \vee c) \wedge (a \vee \bar{b}) \wedge (c \vee \bar{c}))$ consists of three clauses. The last clause is a tautology containing the conflicting literals c and $\bar{c}$. The assignment $\tau = (\bar{a} \wedge b)$ asserts that the literals $\bar{a}$ and b are TRUE.

A clause C is *satisfied* by an assignment τ if $l \in \tau$ for some literal $l \in C$. A clause C is *falsified* by an assignment τ if $\bar{l} \in \tau$ for all literals $l \in C$. A formula F is *satisfied* by an assignment τ if C is satisfied by τ for all clauses $C \in F$. A formula F is *falsified* by an assignment τ if C is falsified by τ for some clause $C \in F$. A formula F is *satisfiable* if there exists a satisfying assignment for F. This assignment is called a *solution*. A formula F is *unsatisfiable* if there does not exist a satisfying assignment for F. Two formulas are *logically equivalent* if they are satisfied by the same set of assignments. If F is logically equivalent to $F \wedge F'$, then F' is *logically implied* by F. Two formulas are *satisfiability equivalent* if they are both satisfiable or both unsatisfiable. Note that two formulas that are logically equivalent are also satisfiability equivalent.

Example 2. The CNF formula $F_2 = ((a \vee b \vee c) \wedge (a \vee \bar{b}))$ is satisfied by the assignment $\tau_1 = (a \wedge b)$ because all clauses in F_2 are satisfied. F_2 is falsified by the assignment $\tau_2 = (\bar{a} \wedge b)$ because the second clause in F_2 is falsified. In this case, F_2 is satisfiable because there exists a solution for F_2 (e.g., τ_1). The formula $F_2' = F_2 \wedge (a \vee c)$ is logically and satisfiability equivalent to F_2 (because F_2 and F_2' have the same set of satisfying assignments) while the formula $F_2'' = F_2 \wedge (a)$ is satisfiability equivalent to F_2 but not logically equivalent (because there are fewer satisfying assignments for F_2'').

2.1.2 DIMACS

Satisfiability (SAT) instances are typically stored in a format called the DIMACS SAT/CNF format. The name DIMACS comes from the Rutgers University research group Center for Discrete Mathematics and Theoretical Computer Science (DIMACS). The DIMACS group hosted several challenges in the 1990s on algorithms and implementations related to graphs and other NP hard problems. In 1992, the Second DIMACS Implementation Challenge included problems on graph cliques, graph coloring, and satisfiability. The original DIMACS format for satisfiablity problems comes from this challenge, although it is hard to find evidence of this today.[1]

The SAT community has since taken ownership of this format and uses versions of it for SAT competitions. The format seems to change at times, but is always based around the DIMACS CNF format. Recent competitions use the definition from the 2009 SAT Competition. This specification is far from complete, however, and it is unfortunate there is no document that defines and evaluates a SAT problem specification.

An example formula in the DIMACS format is depicted in Figure 2.1. The instance begins with the *preamble* which consists of comment lines and a problem line. *Comment lines* begin with the character "`c`" followed by a space. The *problem line* begins with the character "`p`" followed by the string "`cnf`" indicating that this is a problem in conjunctive normal form. Then, the number of variables n and number of clauses c that will appear in the problem are reported. After the preamble, the formula is described. Positive integers 1, 2, ..., n represent literals x_1, x_2, ..., x_n and negative integers -1, -2, ..., $-n$ represent literals $\bar{x}_1$, $\bar{x}_2$, ..., $\bar{x}_n$. The value 0 indicates the end of a clause. Specifications differ on several points: if comments can be included in the formula, if clauses are limited to one per line, if all literals less than or equal to n must be used, etc.

The DIMACS format is relevant to this dissertation in that new proof formats

[1]In fact, there used to be a document online describing the format used in this challenge, but that has since been removed. It's quite difficult to find any mention of the format or challenge in the literature either. This makes the reason behind the "DIMACS" part of the satisfiability format somewhat mysterious.

CNF formula

```
c A comment.
p cnf 4 8
 1  2 -3 0
-1 -2  3 0
 2  3 -4 0
-2 -3  4 0
-1 -3 -4 0
 1  3  4 0
-1  2  4 0
 1 -2 -4 0
```

Figure 2.1: An example CNF formula in DIMACS format.

developed during this work are based on the DIMACS representation.

2.2 Redundancy

Given a formula F, the addition of clause C to F *may* remove solutions from F, potentially changing a formula from satisfiable to unsatisfiable. The removal of a clause C from F *may* add solutions, potentially making an unsatisfiable formula satisfiable. A clause C is *redundant* with respect to a formula F if and only if $F \wedge C$ is satisfiability equivalent to F. For an unsatisfiable formula, all clauses are considered to be redundant because there are no solutions to remove. The justification for the redundancy of a clause can be classified according to certain properties [46]. All SAT solving techniques and unsatisfiability proof systems can be expressed as a sequence of adding and removing redundant clauses. If a solver emits a sequence of additions and deletions, then validation of unsatisfiability claims can be accomplished by checking that each clause added or removed is redundant. In this section, various methods of establishing redundancy are presented.

2.2.1 Resolution, Tautologies, and Subsumption

The addition or removal of clauses with basic forms of redundancy preserve *logical equivalence*, which means that their addition/deletion of the clause does not alter the number of solutions for a formula.

Given two clauses C_1 and C_2 with $l \in C_1$ and $\bar{l} \in C_2$, the *resolution rule* [67] states that the clause $C_\mathrm{R} = C_1 \bowtie C_2 = (C_1 \setminus \{l\}) \vee (C_2 \setminus \{\bar{l}\})$ is logically implied by and redundant with respect to $C_1 \wedge C_2$. The clause C_R is the *resolvent* of C_1 and C_2 and C_1 and C_2 are the *antecedents* of C_R. More explicitly, the notation $C_\mathrm{R} = C_1 \bowtie_l C_2$ states that resolution was performed with *resolution literal* l. Note that resolution is not associative. Any assignment that satisfies antecedents C_1 and C_2 will also satisfy resolvent C_R because either the resolution literal $l \in C_1$ is satisfied, which indicates that $C_2 \setminus \{\bar{l}\} \subset C_\mathrm{R}$ is satisfied, or its negation $\bar{l} \in C_2$ is satisfied, which indicates that $C_1 \setminus \{l\} \subset C_\mathrm{R}$ is satisfied. Thus, the addition of a resolvent preserves logical equivalence for any formula F that contains its antecedents. For a formula F, the set of clauses that contain a literal l is denoted by F_l. Resolution can be performed on sets of clauses where $F_l \bowtie_l F_{\bar{l}} = \{C \bowtie_l C' \mid C \in F_l, C' \in F_{\bar{l}}\}$. Tautologies may be removed from the resulting formula as the set of clauses is an implied conjunction.

Example 3. Given the clause $C_1 = (a \vee b \vee c)$ and the clause $C_2 = (\bar{a} \vee c \vee d)$, the resolvent is $C_1 \bowtie_a C_2 = (b \vee c \vee d)$.

A clause C has redundancy property *Tautology* (T) if and only if $l \in C$ and $\bar{l} \in C$ for some literal l. Tautologies evaluate to TRUE for any assignment, and their addition for a formula F has no impact on the number of solutions, preserving logical equivalence of F.

A clause C has redundancy property *Subsumption* (S) with respect to a formula F if and only if there exists a clause $C' \in F \setminus \{C\}$ such that $C' \subseteq C$. Any solution that satisfies $C\prime$ will also satisfy C, so the addition of C to F preserves logical equivalence of F.

2.2.2 Blocked Clauses and Extended Resolution

The addition or deletion of clauses with stronger forms of redundancy preserve *satisfiability equivalence* for a formula but not logical equivalence (i.e., a formula with zero solutions will continue to have zero solutions, and a formula with more than zero solutions will continue to have more than zero solutions).

Given a formula F, the literal l in clause C *blocks* C with respect to F if $\bar{l} \in C$ (i.e., C is a tautology) or if for all clauses $C' \in F$ such that $\bar{l} \in C'$, the resolvent $C \bowtie_l C'$ is a tautology. The clause C is *blocked* with respect to formula F if there exists a literal that blocks C. Furthermore, if C is blocked with respect to F, then C is also redundant with respect to F.

Example 4. Given the formula $F_4 = ((a \vee b) \wedge (b \vee c) \wedge (\bar{b} \vee \bar{c}))$, the clause $C = (\bar{a} \vee \bar{b})$ is blocked with respect to F_4 because literal $\bar{a}$ blocks C with respect to F_4 (although the literal $\bar{b}$ does not block C because only one of the resolvents is tautology).

Given a formula F, two variables $a \in F$ and $b \in F$, and a variable $x \notin F$, the *extension rule* [74] states that the formula $F \wedge (x \vee \bar{a} \vee \bar{b}) \wedge (\bar{x} \vee a) \wedge (\bar{x} \vee b)$ is satisfiability equivalent to F. In other words, the extension rule adds the definition $x := a \wedge b$ to F and preserves satisfiability. *Extended resolution* is the procedure defined by repeated applications of the extension rule followed by repeated applications of the resolution rule. Extended resolution can polynomially simulate extended Frege systems [21], one of the most powerful known proof systems. Note that the clauses introduced by extended resolution are trivially blocked with respect to the new variable.

One can observe the satisfiability equivalence of the addition and deletion of blocked clauses with the *solution reconstruction method.* Given a formula F and clause C that is blocked on literal l with respect to F, consider some assignment τ that satisfies F but falsifies C. The assignment $\tau' = \tau \setminus \{\bar{l}\} \cup \{l\}$ satisfies C by construction and satisfies F because every clause $C' \in F$ that was satisfied by $\bar{l}$ is also satisfied by another literal $l' \in C'$.

2.2.3 Unit Propagation and Resolution Asymmetric Tautology

A clause C is *unit* for an assignment τ if both (1) there exists exactly one literal $l \in C$ such that $l \notin \tau$ and $\bar{l} \notin \tau$, and (2) for all $l' \in C$ such that $l' \neq l$, $\bar{l}' \in \tau$. The literal l is the *unit literal* for a *unit clause* C. Given a formula F and an assignment τ, *unit propagation*, also known as *Boolean constraint propagation* (BCP), repeatedly extends τ with a unit literal (for a unit clause $C \in F$) until a fixed point is achieved. Unit propagation results in a *conflict* if a clause becomes falsified by unit propagation or if a conflicting literal is added to the assignment.

Example 5. Given the formula $F_5 = ((a \vee b \vee c) \wedge (a \vee \bar{b}))$ and the assignment $\tau = (\bar{a})$, unit propagation returns the assignment $\tau' = (\bar{a} \wedge \bar{b} \wedge c)$.

Given a formula F and a clause C, *reverse unit propagation* (RUP) assigns all literals in C to false, and then performs unit propagation. If a conflict is generated, then C is a *RUP clause* (C has the *RUP property*) and is redundant for F. This process is called "reverse" unit propagation because an assignment is extended in "reverse" order of unit clause discovery in a SAT solver.

Given a clause C and a formula F, *asymmetric literal addition* (ALA) computes the unique clause obtained from repeatedly adding a literal l to C if there exists a clause $C' \in F \setminus \{C\}$ such that $C' = C'' \vee \bar{l}$ where $C'' \subseteq C$.

Given a redundancy property $\mathcal{P}$, a clause C has property A$\mathcal{P}$ with respect to formula F if and only if C has property $\mathcal{P}$ or $ALA(F, C)$ has property $\mathcal{P}$. Given a redundancy property $\mathcal{P}$, a clause C has property R$\mathcal{P}$ on literal l with respect to a formula F if and only if C has property $\mathcal{P}$ or $C \bowtie_l C'$ has property $\mathcal{P}$ for all $C' \in F_{\bar{l}}$. Figure 2.2 shows a hierarchy of redundancy properties and their relation to contemporary SAT techniques [46].

Example 6. Given the formula $F_6 = ((a \vee b) \wedge (b \vee c) \wedge (\bar{b} \vee \bar{c}))$

- The clause $(a \vee \bar{a})$ has property T with respect to F_6.
- The clause $(a \vee b \vee c)$ has the property S with respect to F_6.
- The clause $(a \vee \bar{c})$ has the property AT with respect to F_6.
- The clause $(a \vee \bar{c})$ also has the property RT on a with respect to F_6.
- The clause $(\bar{a} \vee c)$ has the property RAT on $\bar{a}$ with respect to F_6.

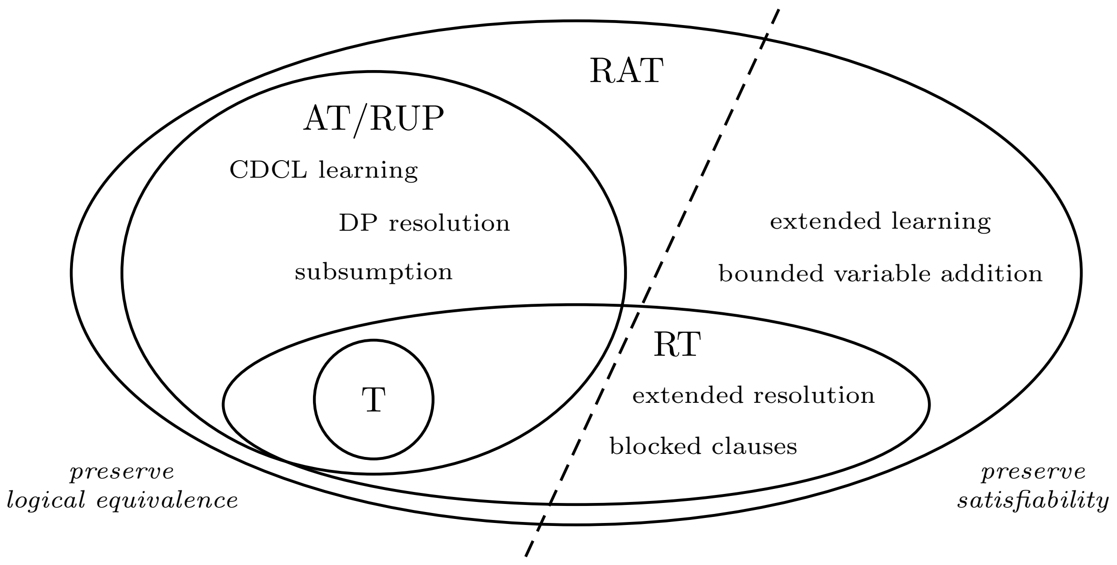

Figure 2.2: Relationships between clause redundancy properties (large text) that can be computed in polynomial time. Techniques (small text) shown in an area denote the most efficient check one can apply to verify a proof trace from a SAT solver that uses that technique. All of the techniques used in contemporary SAT solvers can expressed as a sequence of RAT clauses [46]. The dashed line separates techniques that preserve logical equivalence and those that preserve satisfiability.

It is important to note that the RUP property is equivalent to the AT redundancy property. The RUP property converts a clause to an assignment by negating each literal and then performing unit propagation until a conflict is achieved. The AT property extends a clause by ALA and then checks that the resulting clause is a tautology. Futhermore, RUP clauses also have the RAT redundancy property. In the remainder of this document, the RUP property will be preferred over AT because of its relation to unit propagation.

The most general property in the hierachy above is resolution asymmetric tautology or RAT. The addition (and removal) of RAT clauses to (or from) a formula results in a satisfiability equivalent formula [46] and is a generalization of unit propagation, blocked clauses, and extended resolution. One can repeatedly apply the solution reconstruction method in reverse to reconstruct a satisfying assignment for a formula that has been extended with RAT clauses. Notice that RAT clauses encompass all other forms of redundancy (tautologies, resolvents, blocked clauses, extended resolution, reverse unit propagation, etc.) described in this section. This observation will be key to the proposed proof systems and formats described in the next chapter.

2.3 SAT Solving

Satisfiability (SAT) solving has enabled countless technologies by efficiently deciding Boolean formulas. State-of-the-art SAT solvers employ complex techniques to achieve such efficiency.

2.3.1 Paradigms

The conflict-driven, clause-learning (CDCL) approach [58] is the most prevalent solving paradigm among contemporary SAT solvers. CDCL solvers (like their Davis Putnam Logemann Loveland (DPLL) [27, 26] predecessors) explore the search space by assigning variables until a conflict occurs. Conflict analysis produces a *learned clause* (sometimes called a *conflict clause*) that represents a section of the search space that does not contain a solution and should not be explored again. The learned clause is added to a *learned-clause database* and incorporated into inference steps like unit propagation. This process is repeated until a satisfying assignment is found or the empty clause is learned. The addition of too many learned clauses can inhibit a CDCL solver's performance, so clauses are frequently purged from the learned-clause database. The CDCL paradigm is most effective on application benchmarks where the problems contain some internal structure, which is introduced during the encoding process. CDCL solvers will be the primary focus of the proof techniques presented in this document.

Another leading solving paradigm is the lookahead approach [44]. Lookahead solvers focus on choosing the "best" literal to assign by analyzing the impact of the assignment. These solvers perform well on random benchmarks and hard combinatorial benchmarks where there is a distinct lack of internal structure to the problem.

2.3.2 Preprocessing/Inprocessing

CDCL solvers appeal to suites of *preprocessing* techniques to simplify input formulas before search, strengthening the ability of solvers on hard benchmarks.

Bounded variable elmination (BVE) [30], blocked clause elimination (BCE) [45], hidden tautology elimination (HTE) [43], blocked clause addition (BCA) [46], bounded variable addition (BVA) [55], and subsumption (S) are a few of the many preprocessing techniques used in state-of-the-art solvers. Some SAT solvers even use a method called *inprocessing* where search is frequently interrupted and preprocessing techniques are applied [46]. Preprocessing and inprocessing techniques can be very complicated and are often underspecified. While many techniques can be expressed as a short series of resolution steps, some techniques cannot be expressed using resolution alone and must appeal to extended resolution or a generalization.

Bounded Variable Elimination (BVE) [30] is one of the most effective preprocessing and inprocessing techniques. This technique uses a heuristic that prefers smaller numbers of variables and clauses in a formula: BVE removes a variable $x \in F$ by replacing all clauses with positive F_x and negative $F_{\bar{x}}$ occurrences of the variable by the pointwise resolvents $F_x \bowtie_x F_{\bar{x}}$ if the number of resolvents is smaller than the number of clauses in F_x and $F_{\bar{x}}$. This technique is is defined by resolution and can be expressed as a series of resolution steps.

The counterpart to BVE is Bounded Variable Addition (BVA) [55]; this technique uses the same heuristic as BVE but will add a new variable in order to reduce the number of variables and the number of clauses. BVA constructs two sets of clauses for a new variable x, a set G_x containing the literal x and a set $G_{\bar{x}}$ containing the literal $\bar{x}$, whose resolvents $G_x \bowtie_x G_{\bar{x}}$ appear in a formula F. BVA replaces the resolvents by the two sets of clauses containing the variable x if the number of clauses in the two sets is less than the number of resolvents. Expressing BVA as a series of resolution steps, or even extended resolution steps, is difficult [49]; however, BVA can be easily expressed using a generalization of extended resolution, namely RAT clauses.

As a final example, Blocked Clause Addition (BCA) [46] is the preprocessing/inprocessing technique by which a clause C may be added to a formula F if C is blocked with respect to F. This is the technique that contained an error in the solver LINGELING [46] (Section 1.1.3). This technique can only be expressed using extended resolution or a generalization.

These state-of-the-art techniques enable solvers to efficiently decide satisfiability instances. A proof system for unsatisfiability claims should be able to express *all* of these tecniques—not just some of them. Existing proof systems are described in the next section, but none are able to capture all of the techniques presented above. A new proof system that is capable of expressing these techniques is presented in the next chapter.

2.4 Proof Systems

A *proof trace* (sometimes simply called a *proof*) for a formula F is a sequence of clauses called *lemmas* where each lemma is redundant with respect to F conjoined with all lemmas preceeding it in the sequence. A *refutation* is a proof trace that contains the (unsatisfiable) empty clause. From this point forward, the term “clause” will refer to a clause in a formula and the term “lemma” will refer to a clause in a proof trace. Unsatisfiability proofs have traditionally been expressed in a resolution or clausal proof system. These proof systems have appeared in different formats, but no format has experienced widespread adoption among the SAT community.

2.4.1 Resolution Proofs

Resolution proofs consist of a sequence of lemmas that can be constructed by resolution chains. A *resolution chain* is a sequence of resolutions where the result of each resolution is an antecedent to the next resolution in the sequence. The most commonly-used resolution proof format is TraceCheck [9]. Every clause in the formula and proof is given a clause identifier, and resolution chains are expressed as a sequence of clause identifiers. Figure 2.3 shows an example formula in DIMACS CNF format and a TraceCheck proof.

The benefit of resolution proofs is that every lemma can be derived from resolution information contained in the proof trace, making the proofs easy and efficient to validate. There is no guesswork or rediscovery required by a validation tool. Once a lemma has been validated by checking the resolution chain, it is added to the formula. Resolution is such an elementary operation that simple and fast checking algorithms exist [81, 75].

It is easy to emit proofs in this format for CDCL solvers using "simple" SAT techniques [6], but more advanced techniques and optimizations can be extremely difficult to express. Minimization of learned clauses [12] and bounded variable addition [55] are examples. Variables that do not appear in the formula may be used in resolution proofs (creating a form of extended resolution proofs), but expressing techniques like BVA even in extended resolution is hard [49]. Again, one cannot stress the expressibility problem enough. As the SAT community develops new tech-

CNF formula

```
p cnf 4 8
 1  2 -3 0
-1 -2  3 0
 2  3 -4 0
-2 -3  4 0
-1 -3 -4 0
 1  3  4 0
-1  2  4 0
 1 -2 -4 0
```

TraceCheck proof

```
 1  1  2 -3 0               0
 2 -1 -2  3 0               0
 3  2  3 -4 0               0
 4 -2 -3  4 0               0
 5 -1 -3 -4 0               0
 6  1  3  4 0               0
 7 -1  2  4 0               0
 8  1 -2 -4 0               0
 9 -1  2    0 5 3 7         0
10 -1       0 5 4 2  9      0
11  2       0 3 6 1 10      0
12          0 4 6 8 11 10   0
```

Figure 2.3: An example CNF formula in DIMACS format (left) and a TraceCheck proof (right) are shown in this figure. Each line of the TraceCheck proof begins with a clause identifier (displayed in **bold** print). Then, the literals in the clause are listed, followed by a zero. Finally, the clause identifiers of any antecedents used to justify the addition clause are listed, followed by a zero. Note that clauses from the formula have an empty list of antecedents. The proof ends with the addition of the empty clause.

niques for use in preprocessing, inprocessing, and search, the need for a proof system that can support a generalization of extended resolution only grows.

Another key problem with resolution proofs is that the memory cost can be extremely large (for standard benchmarks). A resolution proof records antecedents for all resolutions necessary to reconstruct each lemma, but the number of antecedents in a resolution proof is often 300 to 400 times larger than the number of lemmas [39]. Figure 2.4 shows a scatter plot where the number of antecedents (or "core arcs") is compared to the number of ("core") lemmas and the number of literals in lemmas. The number of antecedents is 10 times larger than the number of literals in lemmas. Even in a compact resolution format like TraceCheck where literals and antecedents are both expressed as integers, resolution proofs are too large to be practical. Table 3.1 (in the next chapter) shows resolution proof sizes in megabytes for selected benchmarks.

Finally, some resolution proof formats, like TraceCheck, modify the input formula by adding clause identifiers and include the modified formula in the proof trace. While this may not seem like a significant problem, it has a huge impact on the trust associated with proof validation. A solver may incorrectly reproduce the original formula in the resolution proof, causing an inconsistency. This is large concern for applications of proof validation (e.g., satisfiability competitions). However, there are resolution proof formats that avoid this problem by adding clause identifiers implicitly to the original formula [60, 81, 77].

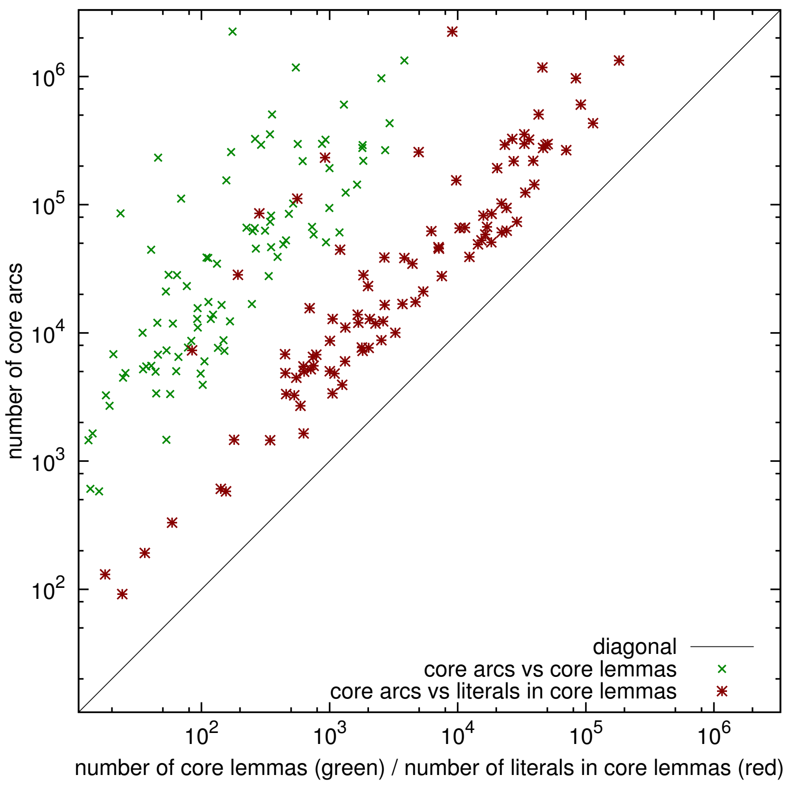

Figure 2.4: A scatter plot comparing the antecedents (called "core arcs") on the y-axis to the "core" lemmas (green x's) and literals in "core lemmas" (red stars) on the x-axis is shown in this figure. The data for this graph was produced by solving the application benchmarks of the 2009 SAT Competition with PicoSAT.

2.4.2 Clausal Proofs

Goldberg and Novikov [33] proposed an alternative to resolution-based proofs. They observed that each clause added by a CDCL solver during conflict analysis can be validated using unit propagation. Clausal proofs are composed of clauses, which are called *lemmas* in this context. Each lemma must be validated by a process known as reverse unit propagation [75]. Once a lemma is validated, it is deemed redundant and may be added to the proof. A RUP validation algorithm will repeatedly check that lemmas of a proof trace are RUP clauses and then extend the formula with each lemma until the empty clause is deemed redundant. Figure 2.5 shows an example RUP proof for a formula in DIMACS CNF format.

CNF formula

```
p cnf 4 8
 1  2 -3 0
-1 -2  3 0
 2  3 -4 0
-2 -3  4 0
-1 -3 -4 0
 1  3  4 0
-1  2  4 0
 1 -2 -4 0
```

RUP proof

```
-1  2 0
-1    0
 2    0
      0
```

Figure 2.5: An example CNF formula in DIMACS format (left) and a RUP proof (right) are shown in this figure. Each line in the RUP proof contains a clause terminated with a zero. The proof ends with the addition of the empty clause.

Figure 2.6 shows the pseudocode for a simple RUP proof checking algorithm. The function RUPchecker (top) accepts a CNF formula F and a RUP proof represented here as a queue of lemmas Q (line 1). If the empty clause is not a member of the proof, then the refutation is ill-formed (lines 2-3). For every lemma L in the queue (lines 4-5), check that L has the RUP property (line 6). If it does not, return that the validation has failed (line 7). Otherwise, extend the formula with L and continue (line 8). If the queue is empty, then the validation succeeded (line 9).

The RUP function (bottom) accepts a formula F and a lemma L that will be tested for the RUP property with respect to F (line 1). Create an assignment τ from L by negating each literal in L (line 2). If there exists a clause in F that evaluates to FALSE under the assignment τ, then L has the RUP property (lines 4-5). If there exists a unit clause in F with respect to assignment τ, then extend τ with the unit literal (lines 6-7) and continue. If a unit clause cannot be found, then the lemma L does not have the RUP property (line 8).

RUP proofs are quite elegant as they can be expresed purely in conjunctive normal form (although the order of the lemmas is important). Each learned clause from a CDCL solver is a RUP clause. One strength of RUP proofs is that they are much more compact than resolution proofs: the redundancy of each lemma can be determined without additional antecedent information. This strength is also a weakness, however, as the antecedents for each lemma must be rediscovered at validation time. Throughout a RUP proof, lemmas that have been deemed redundant via RUP are repeatedly added to the formula, futher increasing the cost of unit propagation. This can be expensive and more complicated to implement efficiently, thereby reducing the trust that the validation utility is correct.

Another strength of RUP proofs is that they are relatively easy to emit for many SAT solving and preprocessing techniques. Modifying a CDCL solver to emit

```
RUPchecker (Formula $F$, Queue $Q$)
  if ($\emptyset \notin Q$)
    return "invalid refutation";
  while ($Q \neq \emptyset$)
    $L = Q$.pop();
    if (RUP($F$, $L$) == FALSE)
      return "validation failed";
    $F = F \cup L$;
  return "refutation validated";
```

```
RUP (Formula $F$, Lemma $L$)
  $\tau = \overline{L}$;
  while (TRUE)
    if ($\exists\ C \in F : \text{eval}(C, \tau) == $ FALSE)
      return TRUE;
    if ($\exists\ C \in F : \text{unit}(C, \tau)$)
      $\tau = \tau \cup \text{unit}(C, \tau)$;
    else return FALSE;
```

Figure 2.6: Psuedocode for a RUP validation algorithm (top) and RUP property (bottom).

RUP proofs can be as easy as adding a line of code that writes each learned clause to a RUP proof file. There is no need to store clause identifiers (as in resolution proofs) in order to emit RUP proofs and there is no memory overhead.

Resolution and clausal proofs systems have changed very little in the past decade and have not adapted to new techniques, such as those described in the previous section. In the next chapter, these proof systems will be revisted.

2.5 Mechanical Verification

Mechanical verification is the process by which an *automated* or *interactive theorem prover* (also called a *proof assistant*) is used to validate the provability of mathematical theorems. These validations may be much larger than those that can be completed by hand as a machine assists in the organization and determines provability. Interactive theorem provers may be based on first-order logic [48] or a higher-order logic [63, 8].

Mechanical verification begins with a formal *model* of a problem, system, or algorithm in the logic of the theorem prover. A *specification* is then constructed that describes the mathematical properties one wishes to exhibit of the model. Finally, the theorem prover is employed to determine that the model satisfies those properties. The theorem prover is often treated as a trusted source, although there are mechanically-verified or self-verifying theorem provers [23, 61].

2.5.1 ACL2

The ACL2 system [48] was used to develop the formalization, specification, and proofs presented in this dissertation. ACL2 is a freely-available system that provides a programming language and a theorem prover, both of which are based on a first-order logic of recursive functions. The programming language and logic are compatible with Common Lisp ("ACL2", or "ACL^2", is an acronym for "A Computational Logic for Applicative Common Lisp"), and an executable ACL2 system can be built on a number of Common Lisp implementations. ACL2 provides efficient execution by way of contemporary Common Lisp compilers. Users of ACL2 often contribute "community books" which may include definitions and theorems certified by ACL2, or even utilities that extend the ACL2 system.

The initial theory for ACL2 has axioms for primitives such as `cons` (an ordered pair constructor), `car` (the first component of a pair), and `cdr` (the second component of a pair). Axioms for Common Lisp functions, such as `member`, are also included, and each user-defined function definition introduces a definitional axiom equating the body of the function with the call of the function on its formal parameters. ACL2 provides interactive use by a top-level read-eval-print loop where arbitrary expressions may be evaluated. ACL2 events include definitions and theorems, both of which modify the the theorem prover's logical database for subsequent proof and evaluation. Theorems are typically expressed using a `defthm` event and function definitions using a `defun` event. An ACL2 function is a *predicate* if it can only return a strictly Boolean value. In ACL2, the constants `t` and `nil` correspond to Boolean values TRUE and FALSE and are designated as special symbols. For the purposes of the logic, however, any term that is non-`nil` is considered to be TRUE.

The syntax of ACL2 is generally case-insensitive (as it is built on Lisp which upcases all input) and uses prefix notation: `(function argument1 ... argumentk)`. As an example, the term denoting the sum of `x` and `y` is `(+ x y)`. A semicolon "`;`" starts a comment that extends to the rest of the line. The functions `let` and `let*` support parallel and sequential bindings, respectively.

In ACL2, functions may return mulitple values by invoking the constructor `mv`, which stands for "multiple value". Elements returned by a `mv` expression may be accessed with the function `mv-nth` which retrieves the nth value of an `mv` (using zero-based numbering). The function `mv-let` accepts three arguments: a list of distint variables, a function that returns an `mv` (with the same number of values as the variables list), and a body. For example, suppose that a function `f` returns two values by using an `mv`. One can compute the sum of these values with the following term:

```
(mv-let (x y)     ; let x and y be the two values returned by
        (f a b c) ; the function f on arguments a, b, and c
        (+ x y))  ; which are used in the computation of x + y
```

One example of an ACL2 community book is `bstar`. This book introduces a macro called `b*` which is an alternative, more powerful way of creating sequential bindings that can include multiple value returns, control statements, and even side effects in a state. Of particular note are the constructs `when`, `if`, and `unless`. Consider the following code that sums a series of values and reports if the sum is negative or non-negative.

```
(b* ((base 0)              ; let base be bound to 0,
     ((mv x y) (f a b c)) ; let x and y be values returned by f,
     (sum (+ x y base))   ; let sum by x + y + base,
     ((if (<= 0 sum))     ; if sum is greater than or equal to 0,
      'non-negative))     ; then return the symbol non-negative,
    'negative)            ; else return the symbol negative
```

ACL2 has no native support for quantification in the logic; however, ACL2 allows a user to define quantified notions using *Skolem functions* via the `defun-sk` event. This event will introduce a witness function that returns a witness object, if such an object exists. If one wants to express the mathematical statement, "there exists an x such that $x < y$", then one could submit the `defun-sk` event:

```
(defun-sk exists-x-<-y (y) ; define a function on y
  (exists x (< x y)))      ; that behaves like ∃ x : x < y
```

This event introduces a non-executable function with one formal parameter, `y`, called `exists-x-<-y-witness` that is axiomatized to return an `x` satisfying the the expression `(< x y)`, if one exists. The non-executable function `exists-x-<-y` returns `t` if `exists-x-<-y-witness` is able to find such an object.

The functions `nth` and `update-nth` are the accessor and updater functions for ACL2 lists, and run in linear-time. When faster execution is needed, one can use ACL2 STOBJs. STOBJs (Single-Threaded OBJects) allow users to benefit from the execution efficiency of Lisp arrays from within ACL2. Arrays enable constant-time lookup and updates for STOBJ fields and arrays, but the cost of this approach is that STOBJs are syntactically limited so that updates to the STOBJ data structure are serialized (hence "single-threaded"). STOBJs are supported in the logic and may logically be treated as lists whose values are accessed and updated with the list-based functions `nth` and `update-nth`.

Links to papers that apply ACL2, as well as detailed hypertext documentation and installation instructions, may be found on the ACL2 home page.[2]

2.6 Verified SAT Solving

One method of assurance for SAT solvers is to develop a mechanically-verified SAT solver. This approach avoids the need for any post-processing, but this does not provide assurance for state-of-the-art solvers that have not been mechanically verified. There is a delicate balance between efficiency and verification in this approach; some techniques may improve the performance of the SAT solver model but each technique must be properly specified and proven correct.

The Davis-Putnam-Logemann-Loveland (DPLL) [27, 26] algorithm is one of the most basic SAT solving algorithms where unit propagation is combined with backtracking. Lescuyer and Conchon [54] formalized and mechanically verified a DPLL algorithm with the Coq [8] proof assistant using a process called reflection. A certified Ocaml implementation was extracted but the efficiency was not evaluated on larger, industrial-scale problems, and the perofrmance was poor for small examples (ranging from 50 variables and 80 clauses to 72 variables and 297 clauses).

Shankar and Vaucher [69] also mechanicallly-verified a DPLL solver using the Prototype Verification System (PVS) [66]. The model they constructed contained a non-executable function for variable selection, so the performance was never evalu-

[2]`http://cs.utexas.edu/users/moore/acl2/`

ated. This work served as a "preliminary and exploratory attempt" at fully specifying a SAT solver based on the DPLL algorithm.

Modern SAT solvers are built around the conflict-driven, clause-learning (or CDCL) [58] paradigm. In CDCL-based solvers, learned clauses are added to a formula during solving to reduce the search space. Marić first verified pseudocode fragments of a CDCL solver in 2009 [56] using the Isabell/HOL proof assistant [63]. This work was completed with an iterative modeling approach where features were gradually introduced to the solver. A large number of CDCL solving techniques were implemented including backtracking, restarts, conflict analysis, and learned-clause minimization, including the the first formalization of the two-watched literal data structure [60]. In 2010, Marić completed the work to finish the mechanical-verification of a full CDCL solver [57]. While the author spent years on the verification process, the performance was never clearly evaluated because the variable selection heuristic was non-executable. Given a proper definition, the author could have extracted an executuable model.

Oe et al. [65] developed a mechanically-verified CDCL solver called VerSAT using the Guru proof assistant [73]. Their implementation featured array-based clauses and conflict analysis. An executable model was extracted to C, and the performance of VerSAT was compared to PicoSAT (which is no longer a state-of-the-art solver) with RUP and TraceCheck proof generation enabled followed by proof validation using checker3 (including RUPtoRES proof conversion). VerSAT solved signicantly fewer benchmarks within the timeout of 3600 seconds (6 of 16 chosen benchmarks). On the benchmarks VerSAT completed, it was often an order or two of magnitude slower than PicoSAT with proof emission and validation. On the benchmarks of the SAT Race 2008, VerSAT solved 19 of the 50 selected benchmarks, while the top-tier solver MiniSAT solved 47 of the 50. Still, this is an incredible result as no other mechanically-verified solver has provided an efficient executable model.

The mechanical-verification of SAT solvers is an excellent goal and has received significant attention from experts in the theorem proving community. However, the performance of a mechanically-verified solver has never reached the level of state-of-the-art SAT solvers. Preprocessing and inprocessing continue to play a larger role in the performance of SAT solvers, and none of these appraoches account for these techniques.

2.7 Validating Unsatisfiability Proofs

Instead of applying mechanical verification to a SAT solver, one can verify a SAT proof checker based on resolution (Section 2.4.1) or clausal proofs (Section 2.4.2). This focuses the verification effort on a small set of techniques designed to validate a proof rather than decide a Boolean formula. The verified proof checker will continue to be relevant, even as solving techniques change. This approach also provides a means to validate the results of preprocessing techniques.

Weber [77, 78] and Bohme [13] demonstrated the first mechanically-verified resolution proof checker using the Isabelle/HOL proof assistant [63] and evaluated the verified proof checker with resolution proofs produced by zChaff. The verified checker was compared to the built-in zChaff proof checker using CPU times (not wall clock times) and was one to two orders of magnitude slower than the zChaff built-in checker and took about 50% longer than the zChaff solver. Memory problems were described during evaluation as MiniSAT failed to produce resolution proofs for all but one of the selected benchmarks. It should be noted that the author's main focus was developing a verified proof checker in order to integrate SAT solving technology into the Isabelle/HOL theorem prover, and this effort was considered a success.

Darbari et al. [22] verified a resolution-based proof checker using the Coq proof assistant [8], which is able to execute models outside of the theorem-prover environment. The performance was better than that of Weber's, but memory issues were still a concern. The authors found that up to 60% of the total time was spent in garbage collection. Armand et al. [1, 2] extended a SAT resolution-based proof checker to include SMT proofs using Coq.

Oe and Stump [64] implemented a non-verified RUP proof checker in C++ and proposed a verified RUP proof checker (VerCheck in Guru [73]), but it is unknown if an implementation for the verified checker exists.

All of these approaches are based on either resolution or clausal proofs and are limited by those proof systems. More advanced techniques such as bounded variable addition or blocked clause addition are difficult (if not impossible) to express. Futhermore, the mechanically-verified proof checkers are unable to to compete with their unverified counterparts. In the next chapter, a new proof format and validation tool are presented that allow more complex techniques. In the chapters that follow, a construction of mechanically-verified proof checker is described for proofs in this new format.

Chapter 3

Expressive Proofs and Efficient Validation

One of the contributions of this work is a new way of expressing unsatisfiability. There are downsides associated with existing proof systems (Section 2.4). Resolution proofs (Section 2.4.1) are large and hard to emit because they contain perfect information about the solving process in terms of low-level resolution operations. Clausal proofs (Section 2.4.2) are inefficient to validate because it is necessary to rediscover all resolution operations used during solving. Neither of these formats allow for advanced preprocessing techniques such as bounded variable addition [55] or blocked clause addition [46]. Solvers often remove learned clauses from their clause database for efficiency, and neither of these proof systems provide a way to represent this important operation.

To solve these problems, a new clausal proof format, called DRAT [80], builds on clausal proofs by employing a generalization of extended resolution, which is a stronger form of redundancy [46]. This extension allows all contemporary solving and preprocessing techniques to be expressed in the proof format [40]. The new proof format benefits from the compactness and easy emission of clausal proof formats. Deletion information ensures that the number of "active" clauses remains small and enables a more efficient validation process [39, 41].

The viability of the DRAT proof format as a standard depends on the existence of a fast validation algorithm. A new validation tool called DRAT-trim is able to efficiently check proofs in this format and serves other functions: DRAT-trim can produce a reduced version of the original formula, an optimized version of the original DRAT proof, and a dependency graph for all clauses involved in the validation [39, 80].

The state-of-the-art in terms of SAT solving and SAT proof validation was presented in Chapter 2. This chapter will describe improvements to the state-of-the-art and begins by discussing the benefits of extending clausal proofs with deletion information and presents two experimental proof formats (IORUP and DRUP) that are based on this extension (Section 3.1). Next, the expressiveness of clausal proofs is addressed by adding support for RAT clauses, and an intermediate proof format called RAT is described. A specification is given for the new DRAT proof format—one of the contributions of this work (Section 3.2). Finally, the input and output, optimizations/techniques, applications, and evaluation of the new proof validation tool constructed for this work are detailed (Section 3.3).

3.1 Adding Deletion Information

RUP proofs [33, 75] are compact and easy to emit, but they can be expensive to validate because the RUP property (Section 2.4.2) requires rediscovery of antecedents to validate each clause. In CDCL-style learning [58], learned clauses are added to a list of redundant clauses as they are discovered. If the addition of clauses was the only operation on a learned-clause database, CDCL solvers would encounter performance problems on large benchmarks. Every learned clause in the database can potentially provide new inferences via unit propagation, and some clauses are even subsumed by newer learned clauses. Thus, learned clauses are removed periodically to improve performance. If a redundant clause is removed during search, it cannot contribute to the discovery of new redundant clauses. The RUP format does not allow for the removal of clauses during validation. As proof validation progresses, each clause may become increasingly more difficult to validate.

One can substantially lower validation costs by including deletion information in the proof trace [39, 41]. Deletion information is just as easy to emit as emitting a learned clause for clausal proofs. Whenever a solver removes a learned clause from the database, the deletion information is written to a proof file. This information may be encoded as "active times" for each clause and lemma in the proof, or it may be stored as deletion instructions that inform the validation tool when a clause is to be removed.

3.1.1 IORUP Proof Format

One experiemental proof format based on deletion information is the IORUP proof format (short for In/Out RUP) [41]. In this format, deletion information is incorporated into each clause in the formula and each lemma in the proof. This is done by prepending *timestamps* to clauses and lemmas. A timestamp represents an index in the sequence of clauses and lemmas that appear in the proof. An "in" timestamp indicates the time a clause or lemma becomes *active* and is available to the validation tool for unit propagation. An "out" timestamp indicates that the clause or lemma is no longer necessary (because it was removed during solving) and should not be considered for unit propagation. Figure 3.1 shows an example. This format in inherently flawed, however, as it shares the same problem that resolution proof formats have (described in Section 2.4.1): the original input formula must be modified. In this case, the modification is that clause in the original formula should include timestamps. This reduces the trust that the proof checker validated a proof for the "original" formula.

CNF formula

```
p cnf 4 8
 1  2 -3 0
-1 -2  3 0
 2  3 -4 0
-2 -3  4 0
-1 -3 -4 0
 1  3  4 0
-1  2  4 0
 1 -2 -4 0
```

IORUP proof

```
iorup 4 8
 1 11   1  2 -3 0
 2 10  -1 -2  3 0
 3 11   2  3 -4 0
 4 12  -2 -3  4 0
 5 10  -1 -3 -4 0
 6 12   1  3  4 0
 7  9  -1  2  4 0
 8 12   1 -2 -4 0
 9 10  -1  2    0
10 12  -1       0
11 12   2       0
12 12           0
```

Figure 3.1: An example CNF formula in DIMACS format (left) and an IORUP proof (right) are shown in this figure. Each line in the IORUP proof contains an "in" timestamp (italics), followed by an "out" timestamp (italics), a clause, and finally a terminating zero. The proof ends with the addition of the empty clause.

IORUP proofs are more difficult to generate during solving than RUP proofs because the "out" timestamp is not known until a clause is removed; however, IORUP proofs can be generated from another experiemental format called DRUP (Section 3.1.2).

The IORUP format increases the size of RUP proofs by adding timestamps. In the worst case, every clause and lemma is binary, and the number of integers for each clause/lemma increases from three (two literals and a terminating zero) to five (two timestamps, two literals, and a terminating zero). This is far from the average case, however, as an average learned clause contains 40 literals (for applications benchmarks). Regardless, IORUP proofs are still more compact than resolution proofs.

3.1.2 DRUP Proof Format

Another transitional proof format called DRUP (short for Deletion RUP) integrates deletion information by interleaving *deletion instructions* into the proof [41, 39]. In this format, a clause that is preceded by the "`d`" flag is considered to be a deletion instruction, which indicates the clause's removal from the set of *active* clauses. The benefit of this approach is that it is just as easy for a CDCL solver to emit deletion instructions as it is to emit learned clauses. Whenever a learned clause is removed from the learned-clause database, the "`d`" flag is prepended to the literals in the learned clause and followed by a terminating zero. Figure 3.2 shows an example of a DRUP proof.

One challenge with a format that includes deletion instructions is that the literals of the clause in the deletion instruction may not be in the same order that the literals appear when the clause/lemma is added. It would be impractical to enforce the same order on literals in corresponding clause/lemma addition and deletion steps because solvers frequently reorder literals during unit propagation (as a part of the

CNF formula

```
p cnf 4 8
 1  2 -3 0
-1 -2  3 0
 2  3 -4 0
-2 -3  4 0
-1 -3 -4 0
 1  3  4 0
-1  2  4 0
 1 -2 -4 0
```

DRUP proof

```
  -1  2    0
d -1  2  4 0
  -1       0
d -1 -2  3 0
d -1 -3 -4 0
d -1  2    0
   2       0
d  1  2 -3 0
d  2  3 -4 0
           0
```

Figure 3.2: An example CNF formula in DIMACS format (left) and a DRUP proof (right) are shown in this figure. Each line in the DRUP proof contains a clause terminated with a zero. A "d" prefix indicates that the clause should be deleted from the database. The proof ends with the addition of the empty clause.

two-watched literal data structure [60]). There are a few solutions to this problem. One could sort literals in clauses and lemmas, but this can be expensive and does not help if deletion instructions are processed during validation. Another solution is to use a hash table to associate corresponding clauses. The hashing function can be selected to include properties that are independent of the literal order like the sum, product, and exclusive-or of literals. Collisions may be addressed by adding linked lists to each entry (e.g., separate-chaining collision resolution).

Deletion instructions may be processed as they are encountered in the proof with a matching procedure like the one described above. An alternative approach to removing clauses from a clause database is to generate timestamps from deletion instructions before validation begins. Then, the unit propagation routine can compare each clause's "out" timestamp to a "global time" before checking to see if it is unit.

The DRUP format can greatly increase the size of a RUP proof. In the worst case, every clause in a formula F and lemma in a RUP proof P is deleted before the validation ends, increasing the proof size from $\sum_{L \in P} |L|$ to $\sum_{C \in F} |C| + 2 \times \sum_{L \in P} |L|$. However, this is still more compact than resolution proofs that store all antecedents (roughly ten times the number of literals [39]) in addition to lemmas. Table 3.1 shows a comparison of the size of proofs in RES, RUP, DRUP, and IORUP formats.

benchmark	RES	RUP	DRUP	IORUP
`aes-bottom12`	9,831.71	70.99	141.04	78.46
`aes-bottom13`	65,703.17	521.54	1,044.81	573.04
`AProVE07-08`	51,386.24	238.61	479.19	259.47
`eq.atree.09`	4,531.55	31.27	61.94	36.67
`eq.atree.10`	16,549.82	113.76	227.87	132.47
`maxor064`	62,010.42	246.32	478.61	260.53
`maxxor032`	46,831.80	289.96	574.58	302.83
`q_query_3_144`	3,949.44	121.79	243.14	137.55
`q_query_3_145`	3,807.54	118.51	237.99	134.20
`q_query_3_146`	3,554.44	117.81	236.91	133.35
`rbcl_xits_07`	3,747.44	79.17	158.56	95.46

Table 3.1: A comparison of the proof sizes (in Mb) on selected benchmarks when solved by GLUCOSE 2.1. The solver emits proofs in DRUP format, which are then converted to RUP (by removing lines that begin with "`d`") and IORUP formats (using a special-purpose conversion tool). RES proofs are produced by the RUPTORES tool [75].

An efficient proof validation utility called DRUP-trim was provided with the definition of the DRUP format. More information on the techniques used in this utility is found in Section 3.3. The DRUP proof format was made available before the 2013 SAT Competition, and solvers submitted to the UNSAT tracks were required to produce proofs to support claims of unsatisfiability. All top-tier solvers (including competition winners) that were submitted to these tracks chose to use the DRUP proof format and the DRUP-trim utility was able to verify all of these proofs.

3.2 Extended Resolution in Proofs

Resolution and (D)RUP proof formats lack the expressivity to capture a growing number of techniques used in state-of-the-art SAT solvers. SAT solvers often use preprocessing and inprocessing in addition to (CDCL-style) learning (Section 2.3.2), and some of these techniques cannot be expressed by resolution-style inference such as bounded-variable addition [55], blocked-clause addition [53, 46], and extended learning [3]. These techniques can be expressed by extended resolution (ER) [74] or a generalization of ER [53]. Järvisalo et al. [46] demonstrated a hierarchy of redundancy properties (Figure 2.2), the most expressive of which is Resolution Asymmetric Tautology (RAT), which is a generalization of RUP and extended resolution. All current preprocessing, inprocessing, and learning techniques used in contemporary solvers can be expressed by the addition and removal of RAT clauses. One key difference, however, between RAT and resolution is that RAT checks satisfiability equivalence instead of logical equivalence.

3.2.1 RAT Proof Format

A proof format based on the RAT property [40] was developed to address the growing number of techniques that go beyond the bounds of resolution. RAT proofs are not only more expressive than previously-defined SAT proof techniques, but they also have the potential to admit stronger redundant clauses which can significantly decrease checking time. Syntactically, RAT proofs are identical to RUP proofs; a RAT proof is a sequence of lemmas expressed as clauses. The only difference is that the RAT property may be used in addition to the RUP property to validate each lemma. In this case, the *resolution literal* should be first literal of the lemma. Figure 3.3 shows an example RAT proof.

CNF formula

```
p cnf 4 8
 1  2 -3 0
-1 -2  3 0
 2  3 -4 0
-2 -3  4 0
-1 -3 -4 0
 1  3  4 0
-1  2  4 0
 1 -2 -4 0
```

RAT proof

```
-1 0
 2 0
   0
```

Figure 3.3: An example CNF formula in DIMACS format (left) and a RAT proof (right) are shown in this figure. Each line in the RAT proof contains a clause terminated with a zero. The proof ends with the addition of the empty clause.

A lemma with the RUP property also has the RAT property, and it is more efficient to check clauses for the RUP property than the RAT property as the latter requires that all clauses/lemmas with the negation of the resolution literal, called *resolution candidates*, to be checked by RUP. In practice, most lemmas are expected to be RUP clauses. Therefore, a lemma is first checked for RUP property. If this fails, then the lemma is checked for the RAT property.

Figure 3.4 shows the pseudocode for a RAT proof checking algorithm including checking the RAT property. The function RATchecker (left) accepts a CNF formula

F and a RAT proof represented here as a queue of lemmas Q (line 1). If the empty clause is not a member of the proof, then the refutation is ill-formed (lines 2-3). For every lemma L in the queue (lines 4-5), check if L has the RAT property (line 6). If it does not, return that the validation failed (line 7). Otherwise, extend the formula with L and continue (line 8). If the queue is empty, then the validation succeeded (line 9).

The RAT function (right) accepts a formula F and a lemma L (line 1). First, the lemma is checked to see if it has the RUP property (from Figure 2.6) (line 2). If it does not, begin the check for the RAT property on the first literal in the lemma (line 3). Construct a set of resolution candidates (line 4); and, for each candidate (lines 5-6), compute the resolvent with the lemma (line 7). If the resolvent does not have the RUP property (line 8), return that the clause does not have the RAT property (line 9). After all resolvents have been checked for RUP, return that the clause has the RAT property (line 10).

```
1  RATchecker (Formula $F$, Queue $Q$)
2    **if** ($\emptyset \notin Q$)
3      **return** "invalid refutation";
4    **while** ($Q \neq \emptyset$)
5      $L = Q$.pop();
6      **if** (RAT($F$, $L$) == FALSE)
7        **return** "validation failed";
8      $F = F \cup L$;
9    **return** "refutation validated";
```

```
1  RAT (Formula $F$, Lemma $L$)
2    **if** (RUP($F$, $L$) == FALSE)
3      $l$ = first($L$);
4      $F_{\bar{l}} = \{C \mid C \in F \wedge \bar{l} \in C\}$;
5      **while** ($F_{\bar{l}} \neq \emptyset$)
6        $C = F_{\bar{l}}$.pop();
7        $R = L \bowtie_l C$;
8        **if** (RUP($F$, $R$) == FALSE)
9          **return** FALSE;
10   **return** TRUE;
```

Figure 3.4: Psuedocode for a RAT validation algorithm (left) and the RAT property (right).

Preprocessing techniques like Bounded Variable Addition (BVA) cannot be expressed in proof systems based on resolution or reverse unit propagation but drastically enhance a solver's ability to decide certain benchmarks. Table 3.2 shows the improvement in solving time with BVA preprocessing as well as the validation time with a simple RAT checker based on the algorithm from Figure 3.4. It also demonstrates the distribution of lemmas that can be validated by RUP (AT) and lemmas that can only be validated by RAT.

	original			BVA preprocessed			RAT proof checking		
benchmark	*#vars*	*#cls*	*time*	*#vars*	*#cls*	*time*	#AT	#RAT	*time*
PH_{10}	90	330	7.71	117	226	1.25	42,853	198	4.19
PH_{11}	110	440	84.42	151	281	12.34	225,959	295	152.82
PH_{12}	132	572	494.29	187	342	8.45	181,603	402	69.01
rbcl_07	1,128	57,446	52.92	1,784	7,598	2.88	72,073	19,681	6.76
rbcl_08	1,278	67,720	1,763.36	1,980	9,004	10.72	151,894	22,830	37.58
rbcl_09	1,430	79,118	—	2,190	10,492	129.20	882,213	26,639	2,631.28

Table 3.2: Evaluation of RAT proof validation with Bounded Variable Addition (BVA) preprocessing on pigeonhole and bioinformatics benchmarks. The first column (set) indicates the benchmark name. The second set of columns shows the number of variables, number of clauses, and solving time with Glucose 2.1. The third set of columns displays the number of variables, number of clauses, and solving time with Glucose 2.1 after BVA preprocessing with Coprocessor. The fourth column set shows the number of lemmas with AT (RUP), number of lemmas with RAT but not RUP, and the validation time with a RAT proof checker.

3.2.2 Complexity

The RAT proof system is strictly stronger than the resolution and reverse unit propagation proof systems presented in Section 2.4; there exist formulas for which there are no sub-exponential-sized resolution proofs, but polynomial-sized RAT proofs exist. Furthermore, RAT proofs can be checked in polynomial time in the size of the proof. Each of these ideas are formally explored in this subsection.

Recall that resolution proofs (such as those in the TraceCheck format) often assign each clause and lemma with an identifier. A resolution chain A for a lemma L and formula F can be expressed as a sequence of identifiers. Let the function CID map identifiers to clauses in F.

Theorem 3.2.1. *Given a formula F and a lemma L with resolution chain A, L can be checked for redundancy with respect to F by the resolution chain A in time $\sum_{0 \leq i < |A|} |CID(A[i])|$ multiplied by a small constant c.*

Proof. Let the sequence A be a zero-indexed sequence and let i be in index into A, initially 0. Let R be initialized to $CID(A[i])$ and increment i. While $i < |A|$, let $C = CID(A[i])$ and check that there exists a literal $l \in C$ such that $\bar{l} \in R$. If so, copy the literals of C into R and increment i. If not, then the redundancy check has failed. When $i = |A|$, test that $R \subseteq L$. $\square$

Theorem 3.2.2. *Given a formula F and a resolution proof P, P can be validated in time $\sum_{(L,A) \in P} (\sum_{0 \leq i < |A|} |CID(A[i])|)$ multiplied by a small constant c.*

Proof. Every lemma $(L, A) \in P$ is checked for redundancy with respect to F using the procedure from the proof of Theorem 3.2.1. $\square$

Theorem 3.2.3. *Given a formula F, a lemma L can be checked for redundancy with respect to F by reverse unit propagation in time $\sum_{C \in F} |C|$ multiplied by a small constant c.*

Proof. Construct the sets $F_l = \{C \mid l \in C \wedge C \in F\}$ for all literals l in F. Let the function $toInt(l)$ map positive literals $x_1, \ldots, x_n$ to integers $1, \ldots, n$ and negative literals $\bar{x}_1, \ldots, \bar{x}_n$ to $-1, \ldots, -n$. Annotate each clause $C \in F$ with the following information: $N_C = |C|$ and $S_C = \sum_{l \in C} toInt(l)$. Construct a zero-indexed propagation queue $Q = \{\bar{l} \mid l \in L\}$ and let i be an index into Q, initially 0. While $i < |Q|$, let $l = Q[i]$ and for each clause $C \in F_{\bar{l}}$, decrement N_C and subtract $toInt(\bar{l})$ from S_C. After each decrement operation, if $N_C = 1$ for some clause C, add $l = toInt^{-1}(S_C)$ to Q if $l \notin Q \wedge \bar{l} \notin Q$. This process will be repeated at most n times where n is the number of variables. In the worst case, each clause $C \in F$ will be visited once for every literal $l \in C$. □

Note that a resolution lemma can be checked linearly in the size of the formula if the antecedents do not form a resolution chain by using the same technique.

Theorem 3.2.4. *Given a formula F and a reverse unit propagation proof P, P can be validated in time $|P| \times (\sum_{C \in F} |C| + \sum_{L \in P} |L|)$ multiplied by a small constant c.*

Proof. Every $L \in P$ is checked for redundancy with respect to F using the procedure from the proof of Theorem 3.2.3. In the maximal case, all (but one) of the lemmas in P have been deteremined to be redundant and added to F and this process is iterated $|P|$ times. □

Theorem 3.2.5. *Given a formula F and a lemma L with resolution literal $l \in L$, L can be checked for redundancy with respect to F by RAT in time $(\sum_{C \in F} |C|)^2$ multiplied by a small constant c.*

Proof. In the worst case, the literal $\bar{l}$ is a member of every clause $C \in F$ and all resolvents of L and C are not tautologies. Each resolvent must be validated by reverse unit propagation in time $\sum_{C \in F} |C|$. $\square$

Theorem 3.2.6. *Given a formula F and a RAT proof P, P can be validated in time $|P| \times (\sum_{C \in F} |C| + \sum_{L \in P} |L|)^2$ multiplied by a small constant c.*

Proof. Every lemma $L \in P$ is checked for redundancy with respect to F using the procedure from the proof of Theorem 3.2.5. In the maximal case, all (but one) of the lemmas in P have been deteremined to be redundant and added to F and this process is iterated $|P|$ times. $\square$

Satisfiability is an NP-complete problem, and a key characteristic of NP-complete problems is that solutions can be checked very quickly with respect to the size of the problem. However, the search for these solutions may be exponential (super-polynomial). While unsatisfiability proofs can be validated in polynomial time (with respect to the size of the proofs), the size of unsatisfiability proofs may be exponential with respect to the size of the formula. Consider the pigeonhole problems. A pigeonhole problem of size n (denoted PH_n) asks if n pigeons can be assigned to $n-1$ holes such that each hole contains at most one pigeon. The problem can be formalized as a satisfiability problem in CNF by letting each Boolean variable $x_{i,j}$ denote that pigeon i is in hole j where i ranges from 1 to n and j ranges from 1 to $n-1$. A translation of PH_n to a CNF formula is given by:

$$\bigwedge_{i \in \{1..n\}} (x_{i,1} \vee x_{i,2} \vee \cdots \vee x_{i,n-1}) \wedge \bigwedge_{i,j \in \{1..n-1\}} \bigwedge_{k \in \{i+1..n\}} (\bar{x}_{i,j} \vee \bar{x}_{k,j})$$

The first set of clauses state that pigeon i must be in some hole. The second set of clauses state that if pigeon i is in hole j, then pigeon $k > i$ cannot be in hole j. This problem requires an exponential number of lemmas for resolution-based proofs [37] and takes an exponential amount of time, with respect to the size of the formula, to solve and validate.

Proof systems based on extended resolution (or a generalization such as RAT) can produce polynomial-sized proofs of the pigeonhole problems [20, 74]. A quadratic-sized RAT proof of PH_n can be constructed by introducing extended resolution definitions that reduce the problem PH_n to the problem PH_{n-1} [40]. While the RAT proof system is strictly stronger than resolution and reverse unit propagation proof systems, there may be formulas with exponential-sized RAT proofs. However, the RAT proof system can polynomially simulate extended Frege systems [21], which is the strongest known proof system and no exponential lower bound is known.

3.2.3 DRAT Proof Format

Finally, the DRAT (Deletion Resolution Asymmetric Tautology) proof format improves upon each of these clausal (RUP, IORUP, DRUP, and RAT) proof formats by using a stronger method of checking redundancy and/or adding deletion information to the proof.

DRAT proofs are expressed as sequence of lemmas and deletion instructions. Lemmas are written in the same way as clausal proofs (the literals of the clause followed by a zero) and may be redundant based on the RUP property or RAT property with respect to the first literal. Deletion instructions are preceded by the “d” flag, the literals of the clause to be deleted, and a terminating zero. The literals of the deletion instruction may appear in a different order than the clause or lemma to which the instruction refers. Figure 3.5 shows an example DRAT proof for a CNF formula in DIMACS format.

CNF formula

```
p cnf 4 8
 1  2 -3 0
-1 -2  3 0
 2  3 -4 0
-2 -3  4 0
-1 -3 -4 0
 1  3  4 0
-1  2  4 0
 1 -2 -4 0
```

DRAT proof

```
  -1        0
d -1 -2  3 0
d -1 -3 -4 0
d -1  2  4 0
   2        0
d  1  2 -3 0
d  2  3 -4 0
            0
```

Figure 3.5: An example CNF formula in DIMACS format (left) and a DRAT proof (right) are shown in this figure. Each line in the DRAT proof contains a clause terminated with a zero. A “d” prefix indicates that the clause should be deleted from the database. The proof ends with the addition of the empty clause.

The DRAT proof format allows the expression of RAT clauses, which includes RUP clauses. A validation utility will attempt to validate each clause by the RUP property. If this fails, the RAT property is checked with respect to the first literal in the clause. DRAT proofs are identical to DRUP proofs; only the method of checking redundancy is changed.

Like the DRUP proof format, an efficient proof validation utility called DRAT-trim was provided with the definition of the DRAT format. More information on the techniques that are used in this utility is located in the following section. The DRAT proof format was made publicly-available before the 2014 SAT Competition, and solvers submitted to the UNSAT tracks were required to produce proofs to support claims of unsatisfiability. All top-tier solvers (including competition winners) that were submitted to these tracks chose to use the DRAT proof format, and the DRAT-trim utility was used to verify these proofs.

3.3 Efficient Validation

The viability of the DRAT proof format as a standard depends on the existence of a fast checking algorithm. A new validation tool called DRAT-trim [80] is able to efficiently check proofs in this format and serves three additional functions: DRAT-trim can produce a trimmed version of the original formula, an optimized version of the original DRAT proof, and a dependency graph for all clauses involved in the validation.

In this section, the input and output of the DRAT-trim utility are defined (Section 3.3.1). Then, the optimizations that allow efficient validation are described: backward proof validation, clause marking, core-first unit propagation, and deletion information (Section 3.3.2). Finally, the applications of DRAT-trim are presented (Section 3.3.3) and the performance of DRAT-trim is evaluated (Section 3.3.4).

3.3.1 Input and Output

DRAT-trim requires two input files: a formula and a proof. Formulas must be expressed in DIMACS CNF format (Section 2.1.2). Proofs must be supplied in the DRAT format [80] (Section 3.2.3), which includes the DRUP [39] (Section 3.1.2) and RUP formats [75] (Section 2.4.2).

The default output of DRAT-trim is a message indicating the validity of a proof with respect to an input formula. This information is very useful when developing SAT solvers, especially since the DRAT format supports checking all existing techniques used in state-of-the-art SAT solvers [40].

Aside from checking the validity of proofs, DRAT-trim can optionally produce three outputs: a *trimmed formula*, an *optimized proof*, and a *dependency graph* in the new TraceCheck$^+$ format. Example input and output files are illustrated in Figure 3.6.

The trimmed formula produced by DRAT-trim is a subset of the input formula in DIMACS format, and the remaining clauses appear in the same order as the input file. The order of the literals in each clause may change.

The optimized proof contains lemmas and deletion information in the DRAT format. Lemmas in the optimized proof will be an ordered subset of the lemmas in the input proof. The first literal of each lemma is the same as the first literal of that lemma in the input proof, however the order of all other literals for the lemma may be permuted. Each line of deletion information will be preceded by a "`d`" tag. This proof is called "optimized" because it contains extra deletion information that is obtained during the backward checking process, described in Section 3.3.2. The optimized proof may be larger than the input proof in size; however, additional deletion information helps reduce validation time because fewer clauses are active during each check.

DRAT-trim is powerful enough to validate techniques that cannot be checked

CNF formula

```
p cnf 4 10
 1  2 -3 0
-1 -2  3 0
 2  3 -4 0
-2 -3  4 0
-1 -3 -4 0
 1  3  4 0
-1  2  4 0
 1 -2 -4 0
-1 -2 -3 0
-1  2 -4 0
```

DRAT proof

```
  -1        0
d -1  2  4  0
   2        0
            0
```

trimmed formula

```
p cnf 4 8
 1  2 -3 0
-1 -2  3 0
 2  3 -4 0
-2 -3  4 0
-1 -3 -4 0
 1  3  4 0
-1  2  4 0
 1 -2 -4 0
```

optimized DRAT proof

```
  -1        0
d -1 -2  3  0
d -1 -3 -4  0
d -1  2  4  0
   2        0
d  1  2 -3  0
d  2  3 -4  0
            0
```

TraceCheck$^+$ file

```
 1  1  2 -3 0              0
 2 -1 -2  3 0              0
 4  2  3 -4 0              0
 5 -2 -3  4 0              0
 6 -1 -3 -4 0              0
 7  1  3  4 0              0
 8 -1  2  4 0              0
 9  1 -2 -4 0              0
11 -1       0 2 6 8        0
12  2       0 1 4 7 11     0
13          0 5 7 9 11 12  0
```

Figure 3.6: DRAT-trim accepts a CNF formula in DIMACS format (top left) and a DRAT proof (top middle) as input. Optional DRAT-trim output includes a trimmed formula (top right), optimized DRAT proof (bottom left), and TraceCheck$^+$ dependency graph (bottom right) with clause identifiers displayed in **bold** print.

with resolution, and a new proof format was designed to be backward-compatible with TraceCheck [9] (Section 2.4.1), but allowing expression of all presently-known solving techniques. Resolution chains in the TraceCheck format begin each line with a clause identifier, followed by the literals of the clause, a zero, the identifiers of antecedents, followed by another zero. The new TraceCheck$^+$ format uses this syntax as well. The formats only differ in the expressing the *reasons* for a lemma's redundancy. If the RUP check succeeds, the reasons are the antecedents as in the TraceCheck format. If a RAT check is necessary, the reasons are the clauses required to let the RAT check succeed.

3.3.2 Optimiziations

Lemmas in a DRAT proof may be checked for redundancy by the RUP or RAT property, but both require an efficient implementation of unit propagation. There are several optimizations that can improve performance when checking DRAT proofs. Backward checking of lemmas combined with core-first unit propagation and deletion information enables clausal proofs to be efficiently validated. Furthermore, RAT checks may optimized by accounting for blocked clauses. These techniques are defined in this section.

3.3.2.1 Backward Checking

Traditionally, proofs of unsatisfiability are validated by checking each lemma in the order that it appears in the proof. This process is intuitive because each lemma must be shown to be redundant with respect to the formula and lemmas that are known to be redundant. As the validation process continues, the formula is extended with more and more lemmas from the proof. Deletion information (Section 3.1) is one way of reducing the number of active clauses; however, some clauses are never used in practice, and some clauses may be used but are unnecessary.

Goldberg and Novikov [33] proposed *backward checking* as a solution to this problem, but no known implementation of this process exists. In backward checking, lemmas are validated in reverse of the order they appear in the input proof file, allowing one to track and limit dependencies between clauses and lemmas. When each lemma is validated, clauses or lemmas that were crucial to the validation are *marked* as *core* clauses/lemmas. All clauses are initially unmarked with the exception of the empty clause. When an unmarked lemma is encountered during backward checking, it may be skipped because no lemma that precedes it or follows it will use it in the proof. The marking process can reducee the number of lemmas that need to be validated, lowering the checking time of the validation tool.

This process is more complex than forward checking because the validation tool needs to keep track of which clauses and lemmas are marked. Backward checking can also be more expensive than forward checking if the number of skipped lemmas is small.

The RATChecker algorithm from Figure 3.4 is redesigned to use backward checking in Figure 3.7. The BackwardRATChecker function accepts a formula F and a stack of lemmas S, as opposed to a queue in previous algorithms (line 1). The algorithm begins by checking for the empty clause in the proof and terminating if it is not present (lines 2-3). Next, all clauses and lemmas are initialized to be unmarked as core (lines 4-5), except for the empty clause which is marked as part of the core (line 6). For every lemma L in the stack (lines 7-8) that is also marked as part of the core (line 9), check if L has the RAT property and mark any clauses or lemmas used to establish this (line 10). If it does not, return that the validation failed (line 11). If the queue is empty, then the validation succeeded (line 12).

```
1  BackwardRATchecker (Formula $F$, Stack $S$)
2    **if** ($\emptyset \notin S$)
3      **return** "invalid refutation";
4    **for** ($C \in F \cup S$)
5      core[$C$] = 0;
6    core[$\emptyset$] = 1;
7    **while** ($S \neq \emptyset$)
8      $L = S$.pop();
9      **if** (core[$L$])
10       **if** (RATmark($F \cup S$, $L$) == FALSE)
11         **return** "validation failed";
12   **return** "refutation validated";
```

Figure 3.7: Psuedocode for a RAT validation algorithm with backward checking and clause marking.

The RATmark function in Figure 3.8 accepts a formula F and a lemma L (line 1). First, check if the clause for the RUP property and mark any clauses or lemmas used if it does (line 2). If it does not, begin the check for the RAT property on the first literal in the lemma (line 3). Construct a set of resolution candidates (line 4); and, for each candidate (lines 5-6), compute the resolvent with the lemma (line 7). If the resolvent does not have the RUP property (line 8), return that the clause does not have the RAT property (line 9). If the resolvent has the RUP property, mark the resolution candidate (line 10) and any clauses or lemmas used to establish the RUP property of the resolvent. After all resolvents have been checked for RUP, return that the clause has the RAT property (line 11).

```
1  RATmark (Formula $F$, Lemma $L$)
2    **if** (RUPmark($F$, $L$) == FALSE)
3      $l$ = first($L$);
4      $F_{\bar{l}} = \{C \mid C \in F \wedge \bar{l} \in C\}$;
5      **while** ($F_{\bar{l}} \neq \emptyset$)
6        $C$ = $F_{\bar{l}}$.pop();
7        $R = L \bowtie_l C$;
8        **if** (RUPmark($F$, $R$) == FALSE)
9          **return** FALSE;
10       core[$C$] = 1;
11   **return** TRUE;
```

Figure 3.8: Psuedocode for checking the RAT property with clause marking.

The RUPmark function (Figure 3.9, top) accepts a formula F and a lemma L that will be tested for the RUP property with respect to F (line 1). Create an assignment τ from L by negating each literal in L (line 2). A set U of clauses that were unit at one point during the procedure is initialized (line 3). If there exists a clause in F that evaluates to FALSE under the assignment τ (line 5), then mark the clauses and lemmas that were used to reach this point (line 6) and return that L has the RUP property (line 7). If there exists a unit clause in F with respect to assignment τ (line 8), then add the clause to the set U (line 9), extend τ with the unit literal (line 10), and continue. If a unit clause cannot be found, then the lemma L does not have the RUP property (line 8). No clauses will be marked.

The MarkCore function (Figure 3.9, bottom) accepts a stack U of clauses that were unit during a RUP check and a resolvent R that was falsified during a RUP check (line 1). Mark the resolvent R (line 2). For all clauses C in the stack U (lines 3-4), check to see if C resolves with R (line 5). If it does, mark C (line 6), replace R with the resolvent of R and C (line 7), and continue. After the stack is empty, return (line 8).

```
1   RUPmark (Formula $F$, Lemma $L$)
2     $\tau = \overline{L}$;
3     $U = \emptyset$;
4     **while** (TRUE)
5       **if** ($\exists\, C \in F : \text{eval}(C, \tau) ==$ FALSE)
6         MarkCore($U$, $C$);
7         **return** TRUE;
8       **if** ($\exists\, C \in F : \text{unit}(C, \tau)$)
9         $U$.push($C$);
10        $\tau = \tau \cup \text{unit}(C, \tau)$;
11      **else return** FALSE;
```

```
1   MarkCore (Unit Stack $U$, Resolvent $R$)
2     core[$R$] = 1;
3     **while** ($U \neq \emptyset$)
4       $C = U$.pop();
5       **if** ($\exists\, l \in R : \overline{l} \in C$)
6         core[$C$] = 1;
7         $R = R \bowtie_l C$;
8     **return** TRUE;
```

Figure 3.9: Psuedocode for checking the RUP property with clause marking (top) and marking clauses used during unit propagation (bottom).

Again, one benefit of this approach is that some lemmas may be skipped during the validation process, potentially improving performance. Another benefit is that the state of the validation algorithm upon completion indicates which clauses in the formula and lemmas in the proof were necessary to derive the empty clause.

This information can be extracted, producing a formula that contains a subset of the original clauses and a proof that contains a subset of the original lemmas. This information can be useful if the proof is to be replayed again.

The backward checking approach on its own is inefficient and impractical, however. There are two additions that can vastly improve the performance. First, deletion information can be combined with the backward checking approach to reduce the number of active clauses during validation. Second, marked clauses can be preferred over unmarked clauses during unit propagation.

3.3.2.2 Deletion Information

The benefits of adding deletion information to proofs have already been described in Section 3.1. This information behaves differently during backward checking of lemmas, however. During forward checking, a deletion instruction indicates that a clause or lemma is no longer active and will not be used in the remainder of a proof. During backward checking, a deletion instruction indicates that a clause or lemma is becoming active. This addresses one of the problems with backward checking in that the initial set of active clauses and lemmas is large when starting at the end of the proof. If deletion instructions are liberally used, then the initial set of active clauses is much smaller.

The BackwardDRATChecker function is shown in Figure 3.10 and it accepts a formula F and a stack of flag and lemma pairs S (line 1). First, the set of *added* lemmas A is calculated by taking the lemma from all pairs that do not have a delete flag “`d`” (line 2). Similarly, the set of *deleted* clauses/lemmas D is calculated by taking the clause/lemma from all pairs that have a delete flag "`d`" (line 3). The empty clause should be in the set of added lemmas; terminate if it is not present (lines 4-5). Next, all clauses in the formula and added lemmas are initialized to be unmarked as core (lines 6-7), except for the empty clause which is marked as part of the core (line 8). For every pair $(flag, L)$ in the stack (lines 9-10), if $flag$ is set to indicate a deletion instruction, then remove L from the set D and continue (lines 11-12). Else, remove L from the set A (line 13-14) and if L is marked as part of the core (line 15), check that L has the RAT property with respect to the clauses in the formula (F), the lemmas that have been added (A), but not the clauses/lemmas that have been deleted (D) (line 16). If it does not, return that the validation failed (line 17). If the queue is empty, then the validation succeeded (line 18).

One can exploit the marking procedure to generate optimal deletion information during validation. When a clause becomes marked, the timestamp and location of the clause are stored as deletion information. The first time a clause is marked and is determined to be necessary to the proof during backward checking is the last time a clause will be used during forward checking or optimized proof emission. Thus, as each clause is marked, optimized deletion information is stored in internal data structures. During post-processing, marked lemmas are printed in order and deletion information with the same timestamp is printed afterwards.

3.3.2.3 Core-first Unit Propagation

The other optimization that greatly improves the performance of backward checking is a core-first unit propagation algorithm. Inferences from unit propagation

```
BackwardDRATchecker (Formula $F$, Stack $S$)
  $A = \{L \mid (flag, L) \in S \wedge flag \neq$ "d"$\}$;
  $D = \{C \mid (flag, C) \in S \wedge flag =$ "d"$\}$;
  if ($\emptyset \notin A$)
    return "invalid refutation";
  for ($C \in F \cup A$)
    core[$C$] = 0;
  core[$\emptyset$] = 1;
  while ($S \neq \emptyset$)
    $(flag, L) = S$.pop();
    if ($flag =$ "d")
      $D = D \setminus \{L\}$;
    else
      $A = A \setminus \{L\}$;
      if (core[$L$])
        if (RATmark($F \cup A \setminus D$, $L$) == FALSE)
          return "validation failed";
  return "refutation validated";
```

Figure 3.10: Psuedocode for a DRAT validation algorithm with backward checking.

can often be obtained from different clauses. For example, suppose two clauses are both considered unit with the same unit literal, but one clause is marked as core and one clause is not. Choosing the latter will increase the size of the marked core unnecessarily. A basic unit propagation routine, such as the one in Figure 3.9, that does not account for this case will increase the size of the core, which in turn creates more lemmas that need to be checked. To avoid this problem, one can create a unit propagation algorithm that prefers marked clauses to unmarked clauses, called *core-first unit propagation*.

The RUPCoreFirst function, shown in Figure 3.11, accepts a formula F and a lemma L that will be tested for the RUP property with respect to F (line 1). Create an assignment τ from L by negating each literal in L (line 2). A set U of clauses that were unit at one point during the procedure is initialized (line 3). If there exists a clause in F that evaluates to FALSE under the assignment τ and is marked as core (line 5), then mark the clauses and lemmas that were used to reach this point (line 6) and return that L has the RUP property (line 7). If there exists a unit clause in F with respect to assignment τ and it is marked as core (line 8), then add the clause to the set U (line 9), extend τ with the unit literal (line 10), and continue. Else, if there exists a clause in F that evaluates to FALSE under the assignment τ (line 11), then mark the clauses and lemmas that were used to reach this point (line 12) and return that L has the RUP property (line 13). Else, if there exists a unit clause in F with respect to assignment τ (line 14), then add the clause to the set U (line 15), extend τ with the unit literal (line 16), and continue. If a unit clause cannot be found, then the lemma L does not have the RUP property (line 17). No clauses will be marked.

```
RUPCoreFirst (Formula $F$, Lemma $L$)
  $\tau = \overline{L}$;
  $U = \emptyset$;
  while (TRUE)
    if $(\exists\, C \in F : \text{eval}(C, \tau) == \text{FALSE} \wedge \text{core}[C])$
      MarkCore($U$, $C$);
      return TRUE;
    if $(\exists\, C \in F : \text{unit}(C, \tau) \wedge \text{core}[C])$
      $U$.push($C$);
      $\tau = \tau \cup \text{unit}(C, \tau)$;
    else if $(\exists\, C \in F : \text{eval}(C, \tau) == \text{FALSE})$
      MarkCore($U$, $C$);
      return TRUE;
    else if $(\exists\, C \in F : \text{unit}(C, \tau))$
      $U$.push($C$);
      $\tau = \tau \cup \text{unit}(C, \tau)$;
    else return FALSE;
```

Figure 3.11: Psuedocode for checking the RUP property with core-first unit propagation.

It has previously been proposed that one should postpone unit propagation on *interesting constraints* [68]. This algorithm is one instance where the interesting constraints are clauses and lemmas that are not yet in the core. Furthermore, it has been demonstrated that roughly 90% of the work in a CDCL solver is “useless” [71]. In other words, a solver with perfect heuristics would be rougly ten times faster. A core-first approach to unit propagation tries to remove some of this unnecessary work.

3.3.2.4 RAT Checks

During a RAT check, the validation process needs access to all clauses containing the negation of the resolution literal. One solution is to build a literal-to-clause lookup table of the original formula and modify it after each lemma addition and deletion step. These updates can be expensive and the lookup table potentially doubles the memory usage of the validation tool. Most lemmas emitted by state-of-the-art SAT solvers can be validated using the RUP check. A lookup table has been omitted in favor of a naive, brute-force approach. When a RUP check fails, the currently active formula is scanned to find all clauses containing the complement of the resolution literal.

A worthwhile optimization for RAT checks, however, is to see if resolution candidates are blocked before adding them to the resolution candidate list. If the candidate is blocked, it will trivially satisfy the RUP check and may be skipped.

3.3.3 Applications

Recall that DRAT-trim can optionally produce trimmed formulas, optimized proofs, and dependency graphs. Each of these has applications that go beyond the validation of unsatisfiability proofs.

- Trimming a formula can be a useful preprocessing step in extracting a Minimal Unsatisfiable Subset (MUS) [62, 7] or computing Craig interpolants [36].
- An optimized proof is very useful for mechanically-verified solvers. After one round of validation, the optimized proof is often much smaller than the original,

but it is still far from minimal. One can repeatedly apply the validation process to further optimize a proof by submitting the reduced output proof of one round of checking to the tool for another round of checking and trimming.

– A dependency graph may be provided to a MUS extractor as input, avoiding the need to rediscover clause dependencies [62, 7].

– Another important use is for solver debugging: a dependency graph gives a step-by-step account of the modifications to a clause database.

3.3.4 Evaluation

DRAT-trim outperforms its RAT checker [40] (because of the addition of deletion information) and DRUP-trim [39] (because of various implementation optimizations) predecessors . Consider the rbcl_xits_09_unknown benchmark, one of the harder benchmarks in the 2009 application suite [40]. The solving time with Coprocessor and Glucose 3.0 is 95 seconds and the validation time with DRAT-trim is 91 seconds, compared to 1096 seconds for a previous RAT validation tool. This particular benchmark can only be solved with bounded variable addition, making a DRAT proof necessary. On the application suite of 2009 SAT Competition, DRAT-trim is 2% faster than its predecessor DRUP-trim on average (within a range of 85% faster to 13% slower). The addition of RAT checking in DRAT-trim does not have a noticeable impact on DRUP proof checking.

DRAT-trim was used to validate the unsatisfiability results of the 2014 SAT

Competition [1]. The competition is divided into various “tracks” that test solvers on different sets of benchmarks. In the 2014 competition, there were two “Certified UNSAT” tracks of the competition: one for a set of benchmarks based on industrial applications and one for a set of benchmarks based on hard combinatorial problems. In these tracks, solvers are only given benchmarks known to be unsatisfiable and the solver must produce a proof of unsatisfiability. In the 2013 SAT Competition, solvers were allowed to produce proofs in a format of their choosing, but all participants chose to emit proofs in the DRUP format. In 2014, solvers were limited to proofs in the DRAT format, and proofs would only be checked using the DRAT-trim validation utility. The adoption of the DRAT format was staggering: dozens of solvers supported the format, including most of the competition winners.

The data from the 2014 SAT Competition has been used to evaluate the performance of DRAT-trim. Solver submissions and the DRAT-trim utility were run on the Lonestar cluster at the Texas Advanced Computing Center (TACC) in Austin, Texas. Each cluster node has 2 hex-core Xeon 5680 processors, 24GB of DDR3-1333MHz RAM, 256KB L2 cache per core, 12MB L3 cache per node, and runs Linux Centos 5.5 OS with a 2.6 x86_64 kernel. Solvers were given 5,000 seconds for each benchmark and the DRAT-trim utility was given 20,000 seconds for each proof validation. The winners of each track of the competition are chosen based on the number of solved (and validated) benchmarks within the timeout.

Results from these competitions are usually displayed as “cactus plots”. In

[1]http://www.satcompetition.org/2014/

a cactus plot, the x-axis values correspond to individual benchmarks and the y-axis corresponds to time; however, the x-axis is sorted based on the y-axis values. This produces a monotonically-increasing plot for each series and is designed to show the overall performance of the series. A cactus plot does not show how each solver compares on individual benchmarks, but instead shows how each solver performed over all benchmarks.

In Figure 3.12, the medalists for the Certified UNSAT Application track of the competition are compared to the validation times of DRAT-trim on the unsatisfiability proofs emitted by each medalist. The top solver, LINGELING, solved and produced proofs for 130 benchmarks in the 5,000 second (per problem) timeout. DRAT-trim was able to validate all unsatisfiability proofs within the individual 20,000 second timeout (although the graph is limited to 5,000 seconds). The important part is that the validation times for DRAT-trim are very similar to the solving times for each of the medalists. In Figure 3.13, the silver and bronze medalists are removed from the graph to show a comparison between LINGELING solving times and DRAT-trim validation times.

In Figure 3.14, the medalists for the Certified UNSAT Hard Combinatorial track of the competition are compared to the validation times of DRAT-trim on the unsatisfiability proofs emitted by each medalist. Similar to the Application track, DRAT-trim was able to validate all proofs within the timeout and validated proofs in a time similar to solving. In Figure 3.15, the silver and bronze medalists are removed from the graph to show a comparison between RISS BLACKBOX solving times and DRAT-trim validation times.

One feature of DRAT-trim is that it is able to output a trimmed formula that contains an unsatisfiable subset of the original clauses, called the *core clauses.* A smaller formula may reduce solving time because it eliminates unnecessary clauses (although this is not always the case). Figure 3.16 shows the size of unsatisfiable formulas (in the number of clauses) before and after trimming with DRAT-trim. This data is from the Certified UNSAT Application track and is displayed for the track medalists. The y-axis is displayed with a logarithmic scale. DRAT-trim is able to reduce the size of the input formula, often by an order of magnitude. Similarly, DRAT-trim is able to emit optimized proofs where unnecessary lemmas have been removed and extra deletion information is added. In Figure 3.17, the size of proofs (in number of lemmas) emitted by the medalists for the Certified UNSAT Application track and the size of proofs emitted by DRAT-trim are compared. Similar to trimmed formulas, lemmas deemed necessary by DRAT-trim are called *core lemmas.*

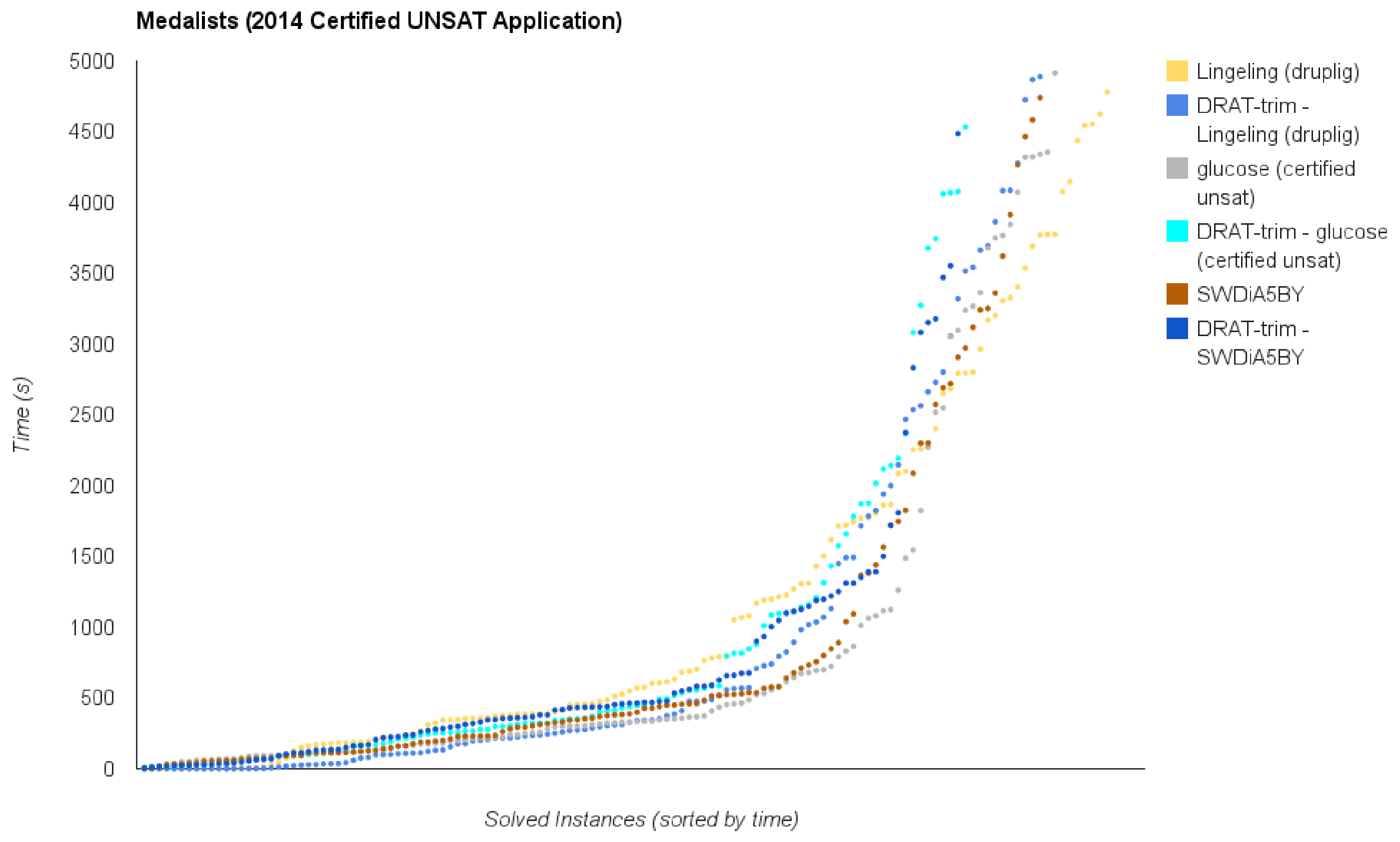

Figure 3.12: A cactus plot showing the solving time (in seconds) for the medalists (in gold, silver, and bronze) in the Certified UNSAT Application track of the 2014 SAT Competition and validation times of DRAT-trim (in shades of blue).

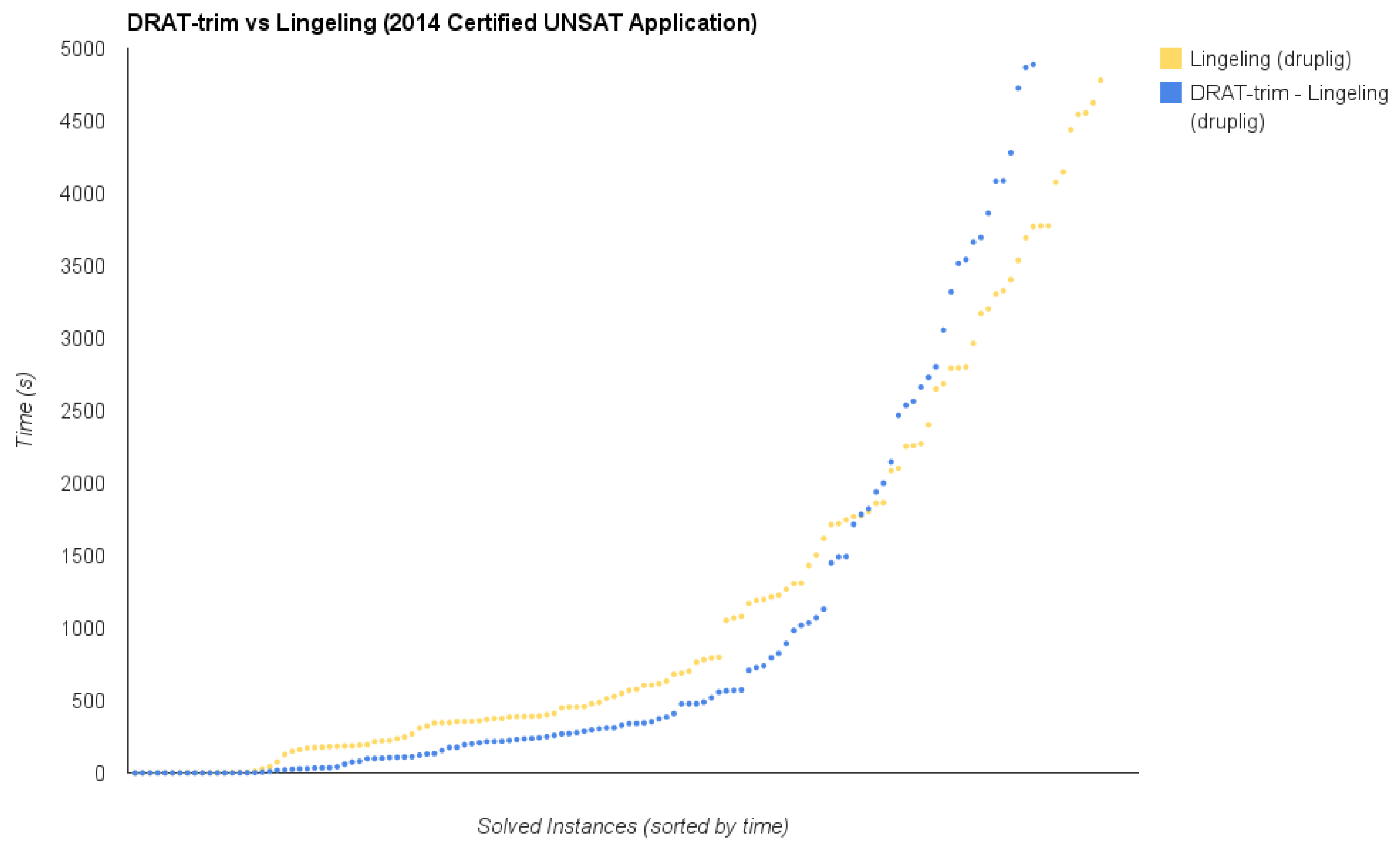

Figure 3.13: A cactus plot showing the solving time (in seconds) for the gold medalist, Lingeling (in gold) in the Certified UNSAT Application track of the 2014 SAT Competition and validation times of DRAT-trim (in blue).

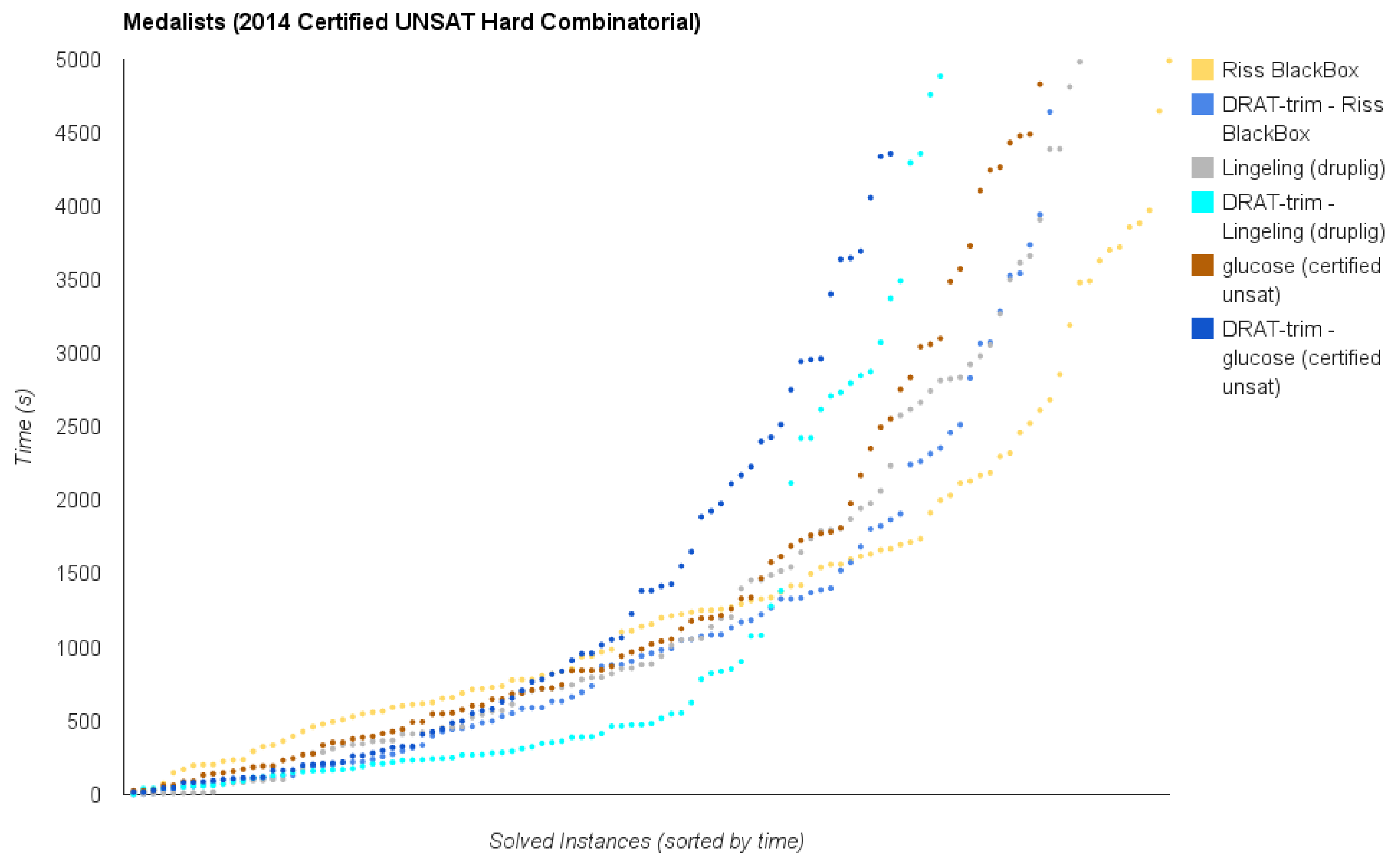

Figure 3.14: A cactus plot showing the solving time (in seconds) for the medalists (in gold, silver, and bronze) in the Certified UNSAT Hard Combinatorial track of the 2014 SAT Competition and validation times of DRAT-trim (in shades of blue).

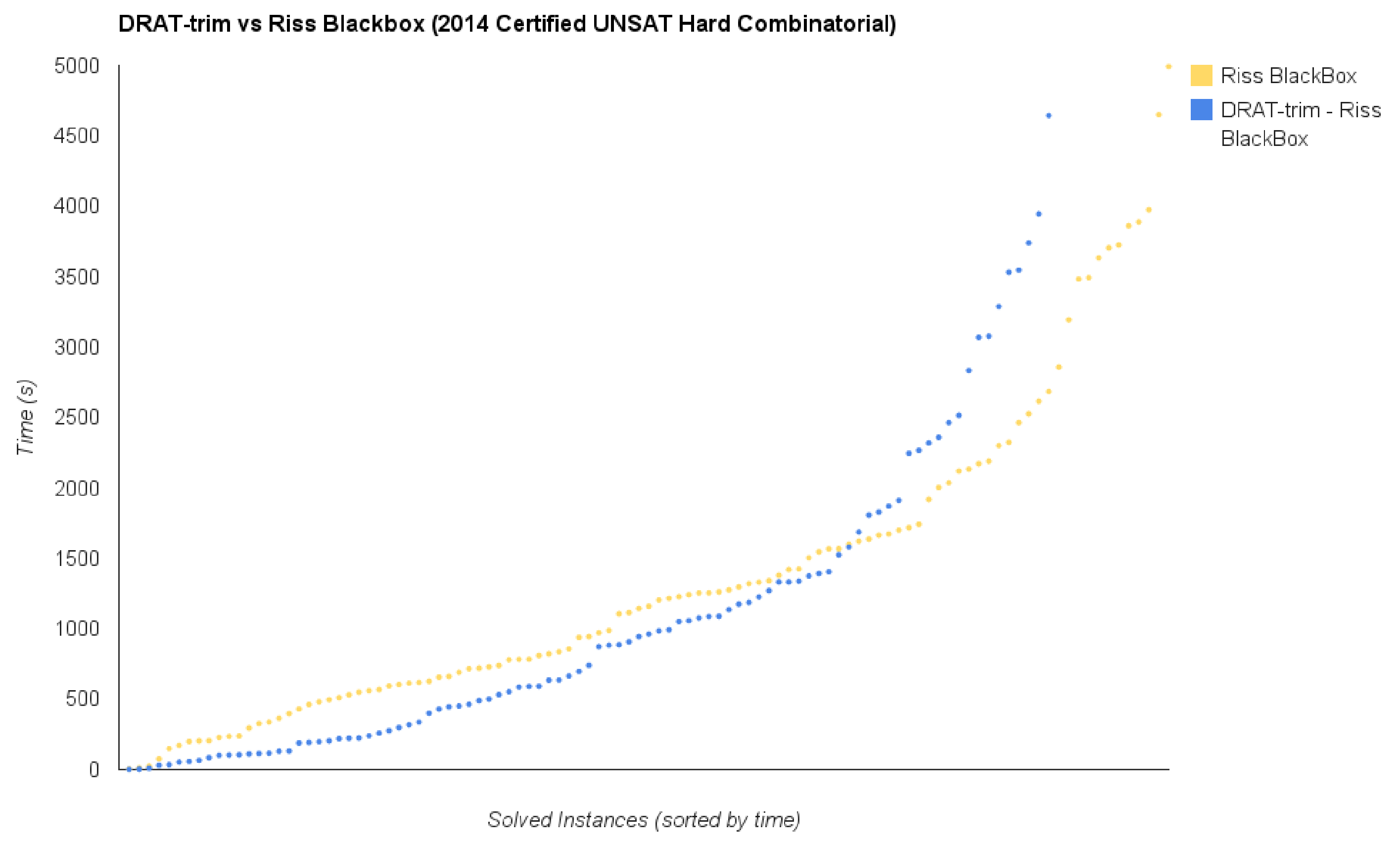

Figure 3.15: A cactus plot showing the solving time (in seconds) for the gold medalist, Riss Blackbox (in gold) in the Certified UNSAT Hard Combinatorial track of the 2014 SAT Competition and validation times of DRAT-trim (in blue).

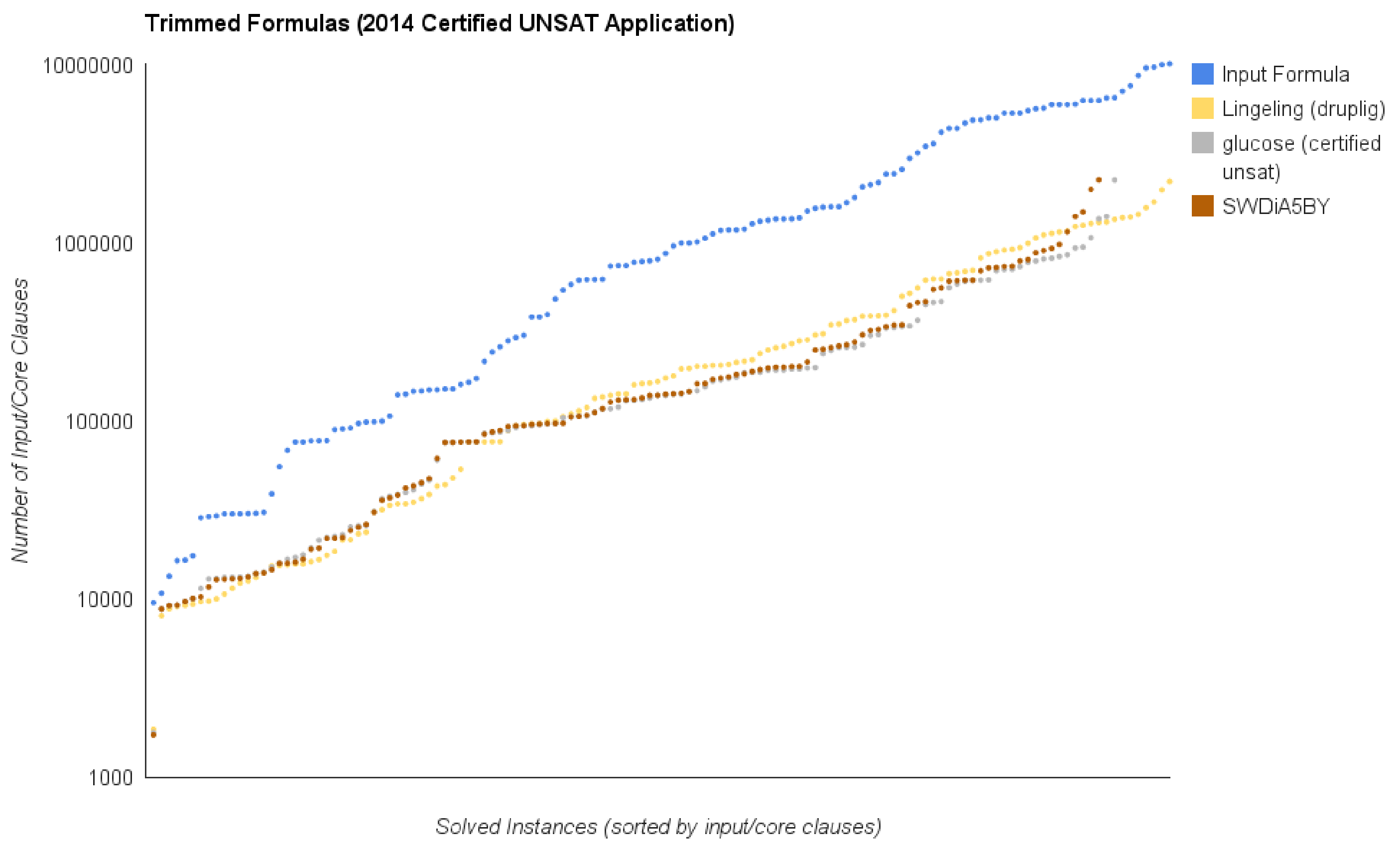

Figure 3.16: A cactus plot showing the formula size (in number of clauses) for the medalists (in gold, silver, and bronze) in the Certified UNSAT Application track of the 2014 SAT Competition and trimmed formula size of DRAT-trim (in shades of blue).

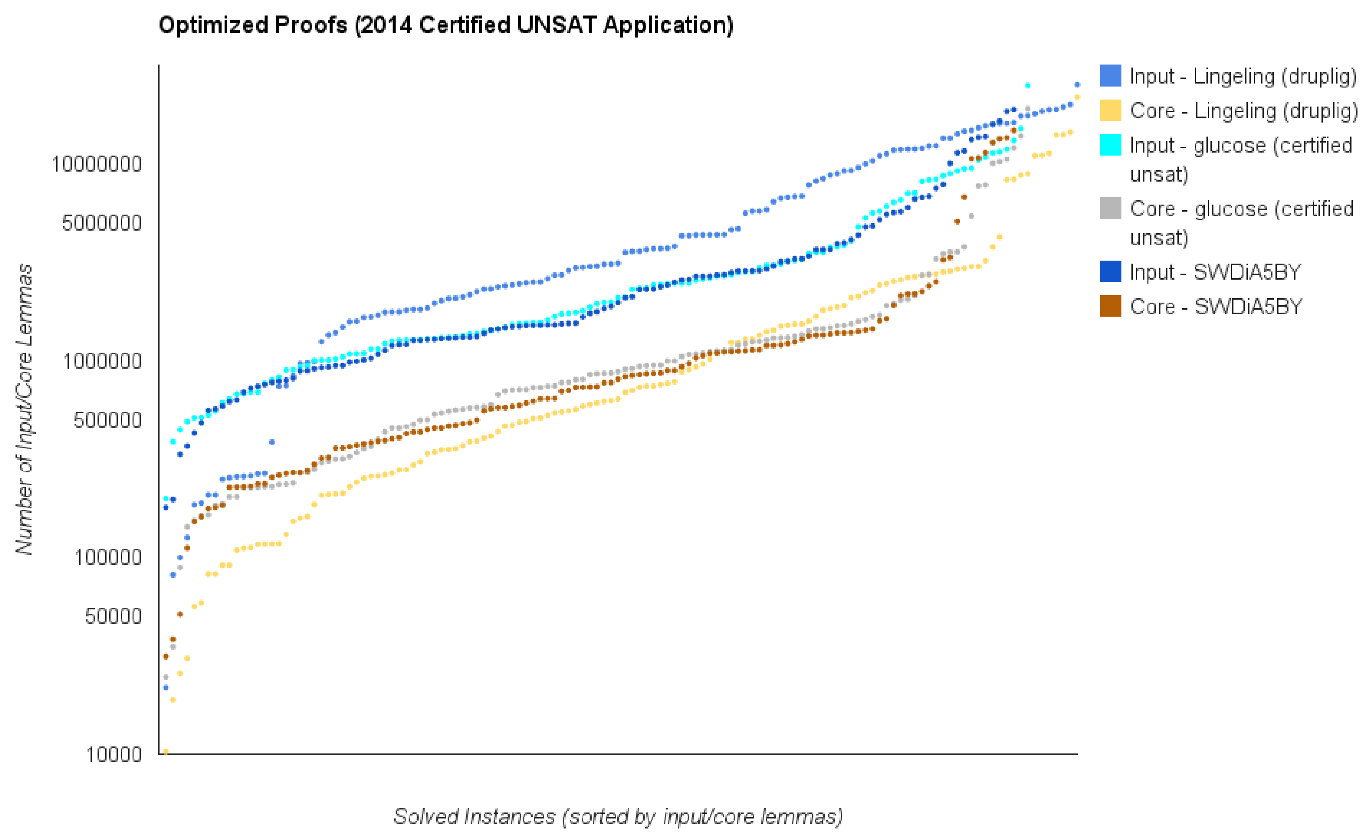

Figure 3.17: A cactus plot showing the proof size (in number of lemmas) for the medalists (in gold, silver, and bronze) in the Certified UNSAT Application track of the 2014 SAT Competition and optimized proof size of DRAT-trim (in shades of blue).

Chapter 4

Unsatisfiability Specification

In order to reason about the soundness of a (proof checking) algorithm, one must first specify what it means for that algorithm to be correct. This necessitates formally describing literals, clauses, formulas, truth assignments, and the evaluation of formulas over those assignments in the logic of the theorem proving system, ACL2. It is important for the reader to acknowledge these descriptions and compare them to their mathematical counterparts from Section 2.1: a proof of correctness for an algorithm is useless if the specification is incorrect. The specification of unsatisfiability used in this work is presented here in a top-down style to be consistent with the top-down approach of later chapters. That is, the definition of unsatisfiability is presented first and then supporting concepts like evaluation are presented later.

4.1 Unsatisfiability

A formula is satisfiable if there exists a satisfying assignment, or solution, for the formula and unsatisfiable if there does not exist a satisfying assignment. To construct this statement, one can use the `defun-sk` event in ACL2 combined with a predicate to test if an assignment is a solution.

```
(defun-sk exists-solution (f)  ; quantified expression (f)ormula
  (exists s (solutionp s f)))  ; ∃ s : s is solution for f
```

Thus, a formula `f` is satisfiable if `(exists-solution f)` is true, and unsatisfiable if `(not (exists-solution f))` is true. Note that if one wants to prove satisfiability (i.e., `(exists-solution f)` is in the conclusion of a `defthm` event), a solution must be exhibited as a witness. If one wants to prove unsatisfiability (i.e., `(not (exists-solution f))` is in the conclusion of a `defthm` event), one must refute that a generic solution is obtained from the rule `exists-solution`.

A solution is an assignment that satisfies a formula—the formula evaluates to true with respect to the assignment. A predicate `solutionp` takes a solution `s` and a formula `f` and tests the solution for the `assignmentp` predicate and tests the evaluation of the formula on the solution for the value `true`.

```
(defun solutionp (s f)                    ; (s)olution, (f)ormula
  (declare (xargs :guard (formulap f)))
  (and (assignmentp s)                    ; s is an assignment and
       (truep (evaluate-formula f s))))   ; f evaluates true wrt s
```

4.2 Assignments and Evaluation

An assignment is a partial function that maps the literals of a formula to truth values. There are many represenions for an assignment, but perhaps the simplest one for an ACL2 specification is a set-based representation stored as a list. Membership of a literal in the list implies that the literal is mappped to TRUE and membership of the negation of a literal implies that the literal is mapped to FALSE.

If neither the literal or its negation is a member of the assignment, then the literal is unassigned and it is mapped to UNDEF.

Thus, assignments are specified as finite lists of unique and non-conflicting literals and are recognized by the predicate `assignmentp`.

```
(defun assignmentp (a)                  ; (a)ssignment
  (declare (xargs :guard t))
  (and (literal-listp a)                ; a proper list of literals
       (unique-literalsp a)             ; each literal appears once
       (no-conflicting-literalsp a)))   ; cannot contain l and l̄
```

Evaluation with respect to an assignment can result in three possible values: TRUE, FALSE, and undefined. Many of the theorems and function definitions related to satisfiability are concerned with which of these three values has been encountered. Because the evaluation is ternary, and not binary, a system for recognizing and reasoning about these results must be developed.

This specification defines nullary functions `true`, `false`, and `undef` returning values `t`, `nil`, and `0`, respectively. The predicates `truep`, `falsep`, and `undefp` recognize the concrete values `t`, `nil`, and `0`, respectively. Several theorems are estbalished to implement the exclusiveness of ternary values (and then all functions are disabled to maintain the abstraction).

Aside. This approach has been somewhat unreliable. The main benefit of the approach is that checkpoints and subgoals are easier to read because the predicates are clearly visible. However, at multiple points, a loss of proof efficiency (the time it takes a proof to complete) was experienced. An alternative approach is to remove the functions described above and rely on ACL2 type-set reasoning or equality-based reasoning. These approaches are very efficient from a theorem proving perspective but lack readability during proof debugging.

A formula is evaluated with respect to an assignment by the ACL2 function `evaluate-formula` and returns `true` if all clauses in the formula evaluate to `true`, `false` if some clause in the formula evaluate to `false`, and `undef` otherwise.

```
(defun evaluate-formula (f a)           ; (f)ormula, (a)ssignment
  (declare (xargs :guard (and (formulap f)
                              (assignmentp a))))
  (if (atom f)                          ; if f is empty
      (true)                            ; return true
    (let* ((c (car f))                  ; let c be the first clause
           (cv (evaluate-clause c a))) ; and cv be the eval of c
      (if (falsep cv)                   ; if c evals false
          (false)                       ; then f is false
        (let* ((rf (cdr f))             ; let rf be f without c
               (rfv (evaluate-formula  ; and rfv be eval of rf
                     rf a)))
          (cond
           ((falsep rfv) (false))       ; if rfv false, false
           ((undefp cv) (undef))        ; if cv undef, undef
           (t rfv))))))                 ; else, return rfv
```

A clause is evaluated with respect to an assignment by the ACL2 function `evaluate-clause` and returns `true` if some literal in the clause evaluates to `true`, `false` if all literals in the clause evaluate to `false`, and `undef` otherwise.

```
(defun evaluate-clause (c a)            ; (c)lause, (a)ssignment
  (declare (xargs :guard (and (clausep c)
                              (assignmentp a))))
  (if (atom c)                          ; if c is empty
      (false)                           ; return false
    (let* ((l (car c))                  ; let l be first literal
           (lv (evaluate-literal l a))) ; and lv eval of l
      (if (truep lv)                    ; if l evals true
          (true)                        ; then c is true
        (let* ((rc (cdr c))             ; let rc be c without l
               (rcv (evaluate-clause    ; and rcv be eval of rc
                     rc a)))
          (cond
           ((truep rcv) (true))         ; if rcv true, true
           ((undefp lv) (undef))        ; if lv undef, undef
           (t rcv)))))))                ; else, return rcv
```

Finally, a literal `l` is evaluated with respect to assignment `a` by the function `evaluate-literal` and returns `true` if `(member l a)`, `false` if `(member (negate l) a)`, and `undef` otherwise.

```
(defun evaluate-literal (l a)      ; (l)iteral, (a)ssignment
  (declare (xargs :guard (and (literalp l)
                              (assignmentp a))))
  (cond
   ((member l a) (true))           ; if $l \in a$, true
   ((member (negate l) a) (false)) ; if $\bar{l} \in a$, false
   (t (undef))))                   ; else, undef
```

4.3 Formulas and Clauses

A formula is a conjuction of clauses. In this specification, formulas are finite lists (an implied conjunction) of clauses and are recognized by the predicate `formulap`.

```
(defun formulap (f)            ; (f)ormula
  (declare (xargs :guard t))
  (if (atom f)                 ; if f is not a pair
      (null f)                 ; then f must be nil
    (and (clausep (car c))     ; else the first elem is a clause
         (formulap (cdr f))))) ; and the rest are a formula
```

Clauses may not be unique in this representation of a formula; detecting the uniqueness of clauses could be supported by implementing a canonical form or hashing routine. The set of all literals occuring in a formula is computed by the function `all-literals`.

Clauses are specified as finite lists (an implied disjuntion) of unique and non-conflicting literals and are recognized by the predicate `clausep`. This definition disallows tautologies.

```
(defun clausep (c)                     ; (c)lause
  (declare (xargs :guard t))
  (and (literal-listp c)               ; a proper list of literals
       (unique-literalsp c)            ; each literal appears once
       (no-conflicting-literalsp c))) ; cannot contain l and l̄
```

Note that the definition of `clausep` is actually the same as that of `assignmentp`, but the interpretations are different.

4.4 Variables and Literals

There are many different approaches to representing variables and literals, and the specification changed many times throughout this work. A full description of the benefits and detriments of literal encoding schemes is presented in Section 7.3.

Variables are specified as positive integers less than $2^{58} - 1$ and are recognized by the predicate `variablep`. The integer n corresponds to a variable x_n.

```
(defun variablep (v)          ; object (x)
  (declare (xargs :guard t))
  (and (integerp v)           ; v is an integer,
       (< 0 v)                ; greater than 0,
       (< v (1- *2^58*))))    ; and less than 2^58 - 1
```

Literals are positive and negative representations of a variable. In some solvers and literature, literals are expressed as positive and negative integers where the integer n corresponds to the literal x_n and the integer $-n$ corresponds to the literal $\bar{x}_n$. The function `abs` then converts literals to variables and the function `unary--` negates literals. This specification was used for the majority of the early work in this disseratation.

Currently, literals are specified as integers greater than 1 and less than 2^{59} and are recognized by the predicate `literalp`. The integers $2, 3, 4, 5, ...$ correspond to the literals $x_1, \bar{x}_1, x_2, \bar{x}_2, ...$

```
(defun literalp (l)          ; (l)iteral
  (declare (xargs :guard t))
  (and (integerp l)          ; l is an integer,
       (< 1 l)               ; greater than 1,
       (< l *2^59*)))        ; and less than 2^59

(defun negate (l)                           ; (l)iteral
  (declare (xargs :guard (literalp l)))
  (logxor l 1))                             ; l ⊕ 1
```

The *arithmetic shift* function `(ash l -1)` converts literals to variables and the *logical exclusive or* function `(logxor l 1)` negates literals.

Chapter 5

RAT Algorithm

In this chapter, an executable algorithm for RAT validation is presented in a top-down fashion. This algorithm is designed to be as simple as possible while still being capable of checking any RAT proof.

Note that many of the lower-level functions employ evaluation functions from the specification. It would have been possible to create separate functions for the specification and implementation (e.g. `evaluate-literal-spec` for the specification and `evaluate-literal-impl` for the implementation) and then prove corresponding functions equivalent, but there was no real benefit to doing so in such a high-level algorithm.

5.1 Validation

The purpose of the algorithm is to validate an unsatisfiability proof for a given formula. A proof is a series of lemmas (or clauses) that have the RAT redundancy property with respect to an extended formula that contains the original formula and all preceding lemmas.

At the highest level, a `verify-proof` function should accept a formula (satisfied by the `formulap` predicate from the specification) and a proof stored as a list of clauses, defined as a `clause-listp` object. This function will return `t` when the clause list has been validated as a proof and `nil` if the clause list contains some lemma that is not redundant (according to the RAT property).

```
(defun verify-proof (cl f)          ; (cl) clause list, (f)ormula
  (declare (xargs :guard (and (formulap f)
                              (clause-listp cl))))
  (if (atom cl)                              ; if cl empty,
      t                                      ; success
    (let ((c (car cl)))                      ; let c be first clause
      (if (verify-clause (car cl) f)         ; if c is redundant,
          (verify-proof (cdr cl)             ; verify rest of cl
                        (cons (car cl) f)) ; with c added to f
        nil))))                              ; else, failure
```

At every step, one clause of the clause-list is validated with respect to the (extended) formula. If the clause can be validated, then it is removed from the proof and the formula is extended with the validated clause.

Clauses are individually validated by the `verify-clause` function. This function first checks if the clause has the RUP property, and if this fails, the function checks for the RAT property with respect to the first literal in the clause.

```
(defun verify-clause (c f)       ; (c)lause, (f)ormula
  (declare (xargs :guard (and (clausep c)
                              (formulap f))))
  (or (RUPp f c)                 ; c has RUP wrt f
      (and (not (atom c))        ; c is not empty, and
           (RATp f c (car c))))) ; c has RAT on first lit wrt f
```

If the clause `c` above does not have the RUP property with respect to the formula `f`, then there must be at least one literal in the clause in order to check the RAT property. Thus, the test for `atom` is included in the definition. This also is necessary to relieve the guard of `consp` on the call to `car`.

5.2 Redundancy Properties

The `verify-clause` function checks for both the RUP property and the RAT property, so both of these redundancy properties must be modeled in ACL2. The definition of RUP from Section 2.2 uses a process called reverse unit propagation where a clause is negated, unit propagation is performed, and tested for a conflict. Note that this is an operation on a clause and a formula and the result is a Boolean.

The RUP property is modeled by the predicate `RUPp` in ACL2.

```
(defun RUPp (f c)                  ; (f)ormula, (c)lause
  (declare (xargs :guard (and (formulap f)
                              (clausep c))))
  (falsep (evaluate-formula        ; false evaluation of
           f                       ; f after
           (unit-propagation       ; performing unit propagation
            f                      ; wrt f on the assignment
            (negate-clause c))))) ; obtained from negating c
```

The tested clause `c` is first negated to create an assignment with the `negate-clause` function, which simply negates every literal in `c`. This assignment is extended by unit propagation with respect to the formula `f` using the `unit-propagation` function. Then, a conflict is detected by checking if evaluation of the formula `f` on the extended assignment is `false`.[1]

The `negate-clause` function above is a simple recursive function that takes a

[1]Note that one could model the ALA function and the tautology predicate in ACL2 and prove the equivalence of the AT property with the RUP property, but this is left to the reader as it does not enhance the execution of this algorithm.

list, representing a clause, and applies the `negate` function to every element of the list, producing a new list that represents an assignment.

```
(defun negate-clause (c)        ; (c)lause
  (declare (xargs :guard (clausep c)))
  (if (atom c)                  ; if c empty
      nil                       ; then return nil
    (cons                       ; return the pair of
     (negate (car c))           ; negation of first of c
     (negate-clause (cdr c))))) ; with recursion on rest of c
```

If the tested clause does satisfy the `RUPp` predicate, then the clause is checked for the RAT property with respect to the first literal in the clause. This follows the convention of the RAT [40] and DRAT [80] proof formats. The RAT property is modeled by the predicate `RATp` which uses a helper function `RATp1`.

```
(defun RATp (f c l) ; (f)ormula, (c)lause, (l)iteral
  (declare (xargs :guard (and (formulap f)
                              (clausep c)
                              (literalp l))))
  (RATp1 f f c l))  ; call the helper function with duplicate f
```

The `RATp` function saves a reference to the formula by adding it as an additional argument. This allows the first argument of the `RATp1` helper function to be used in the recursion over the clauses in the formula, and the second argument remains untouched so that the RUP property can be checked on each non-tautological resolvent.

```
(defun RATp1 (cl f c l)  ; (cl) clause list, (f)ormula,
                         ; (c)lause, (l)iteral
  (declare (xargs :guard (and (clause-listp cl)
                              (formulap f)
                              (clausep c)
                              (literalp l))))
  (if (atom cl)                         ; if cl is empty,
      t                                 ; then success (all checked)
    (if (not (member (negate l)         ; else if l̄ ∉ first clause
                     (car cl)))         ; of cl,
        (RATp1 (cdr cl) f c l)          ; then check rest of cl
      (let ((r                          ; else, let r be
             (resolution l              ; resolution on l
                         c              ; between c
                         (car cl))))    ; and first in cl
        (if (tautologyp r)              ; if r is tautology,
            (RATp1 (cdr cl) f c l)      ; then check rest of cl
          (and (RUPp f r)               ; else, r has RUP wrt f
               (RATp1                   ; and check RAT on
                (cdr cl) f c l)))))))   ; rest of cl
```

The `RATp1` predicate first checks if the current clause in the `clause-list` contains the resolution variable. If it does, then the resolvent is computed and tested to see if it is a tautology. If not, then the RUP property is tested on the resolvent and the rest of the clauses in the `clause-list` are validated.

5.3 Resolution

Resolution is computed by the `resolution` function that takes a literal and two clauses as input and returns a new pseudo-clause as output. The literal must be a member of the first clause. The literal and its negation are removed from the clauses and then a union is performed to merge two clauses and remove duplicates. The result is a pseudo-clause because the union operation only guarantees that the result will be a unique list of literals. It does not guarantee that the result is non-tautological.

```
(defun resolution (l A B)          ; (l)iteral, (A) (B) clauses
  (declare (xargs :guard (and (literalp l)
                              (clausep A)
                              (clausep B))))
  (union                           ; union the results
   (remove-literal l A)            ; of l removed from A
   (remove-literal (negate l) B))) ; and l̄ removed from B
```

5.4 Unit Propagation

Unit propagation is the core component of the RAT proof checking algorithm and is perhaps the most difficult to reason about. The version of unit propagation presented here is naïve and does not make use of watched-literal data structures [60] to improve efficiency. The function `unit-propagation` takes a formula and an assignment as input and returns a (new) assignment.

```
(defun unit-propagation (f a)      ; (f)ormula, (a)ssignment
  (declare (xargs :guard (and (formulap f)
                              (assignmentp a))
                  :measure (num-undef f a)))   ; termination measure
  (mv-let (ul                      ; let ul be unit literal and
           uc)                     ; uc be unit clause returned by
          (find-unit-clause f a)   ; find-unit-clause on f and a
          (declare (ignorable uc)) ; ignore uc below
          (if (not ul)             ; if no unit literal,
              a                    ; then return a
            (unit-propagation      ; else, recur
             f                     ; with formula and
             (cons ul a)))))       ; a extended with ul
```

The `unit-propagation` function is recursive and uses the number of clauses that evaluate to undefined in the formula, computed by `num-undef`, as a termination measure. Each recursive call attempts to find a unit clause with the `find-unit-clause` function and extend the formula with the unit literal, removing a unit clause from the formula in the process. Thus, while the size of the assignment increases at each step, the number of clauses evaluating to undefined decreases because the addition of the unit literal to the assignment will cause the unit clause to evaluate to true.

Aside. In some of the initial work on the verification of SAT solvers, the number of unassigned variables was used as a measure. This measure is a little simpler conceptually but seemed to necessitate reasoning about canonical assignments, and much of the canonical assignment reasoning used quantification to represent the weakening and strengthening of assignments. This complicated induction schemes for unit propagation and was abandoned. The `num-undef` approach supports a cleaner induction scheme.

The `num-undef` function is a basic function that recurs over a formula and counts the number of clauses that evaluate to undefined with respect to a given assignment.

```
(defun num-undef (f a)              ; (f)ormula, (a)ssignment
  (declare (xargs :guard (and (formulap f)
                              (assignmentp a))))
  (if (atom f)                      ; if f empty,
      0                             ; then return 0
    (if (undefp (evaluate-clause    ; else, if first of f evals to
                 (car f) a))        ; undef wrt a
        (1+ (num-undef (cdr f) a)) ; increment the recursive result
      (num-undef (cdr f) a))))      ; else, recur on rest of f
```

The `find-unit-clause` function recurs over the formula and tests each clause with the `is-unit-clause` function. If a unit clause is found, a multiple-value (`mv`) pair is returned containing the unit literal and unit clause. If a unit clause is not found, then the pair (mv nil nil) is returned indicating that no unit literal and unit clause could be found.

```
(defun find-unit-clause (f a)     ; (f)ormula, (a)ssignment
  (declare (xargs :guard (and (formulap f)
                              (assignmentp a))))
  (b* (((if (atom f))             ; if f empty,
        (mv nil nil))             ; return nil (no unit clause)
       (c (car f))                ; let c be first of f
       (rf (cdr f))               ; let rf be rest of f
       (ul (is-unit-clause c a)) ; is-unit-clause on c returns ul
       ((if ul) (mv ul c)))       ; if ul exists, return ul and c
      (find-unit-clause rf a)))   ; else, recur with rf
```

Note that in the `unit-propagation` function, the second return value `uc` (short for unit clause) of `find-unit-clause` is ignored, which might indicate that only the unit literal should be returned. However, returning the unit clause greatly simplifies reasoning about `find-unit-clause`. Namely, the unit literal is a member of the unit clause, and the unit clause is a member of the formula. Without this return value, an existential might be needed.

The `is-unit-clause` function checks if all but one literal in the clause evaluates to false and no literal evaluates to true. If the clause is unit, then the unit literal is returned. Else, `nil` is returned indicating that the clause is not unit.

```
(defun is-unit-clause (c a)      ; (c)lause, (a)ssignment
  (declare (xargs :guard (and (clausep c)
                              (assignmentp a))))
  (cond
   ((atom c) nil)                ; if c is empty, then return nil
   ((truep (evaluate-literal     ; if first of c evals to
           (car c) a))           ; true wrt a,
    nil)                         ; then return nil
   ((undefp (evaluate-literal    ; if first of c evals to
            (car c) a))          ; undef wrt a,
    (if (falsep (evaluate-clause ; then if rest of c evals to
                (cdr c) a))      ; false wrt a,
        (car c)                  ; then return first of c
      nil))                      ; else, return nil
   ((falsep (evaluate-literal    ; if first of c evals to
            (car c) a))          ; false wrt a
    (is-unit-clause (cdr c) a))  ; recur on rest of c
   (t nil)))                     ; otherwise nil, should not reach
```

The function is Boolean, but the return value has meaning in the non-`nil` case (similar to the built-in `member` function). This function is designed to detect the existence of exactly one undefined literal where the rest of the literals evaluate to `false`. Recursive calls of `is-unit-clause` happen only when the current literal evaluates to `false`. Once a literal evaluates to `undef`, the `evaluate-clause` function finishes testing the rest of the clause, guaranteeing only `false` literals.

Note that `unit-propagation` will continue to search for unit clauses until none remain and a maximal assignment is obtained. This makes it relatively easy to reason about the resulting assignment and makes no assumptions about the use of the `unit-propagation` function. However, in proof validation, there is no reason to futher extend an assignment if it falsifies a clause and thereby falsifies a formula. This implementation could be improved by adding an additional return value from `is-unit-clause` that also detects if the clause is falsified. Then `find-unit-clause` and `unit-propagation` could stop early with an incomplete, but falsifying assignment, which is all that is necessary for `RUPp` and `RATp`.

Chapter 6

RAT Checker Proof of Correctness

The RAT validation algorithm described in Chapter 5 has been mechanically verified using ACL2 [79] by proving a soundness theorem detailed in this section. First, a specification for the algorithm is presented (that makes use of the unsatisfiability specification from Chapter 4), and then the mechanical proof process is described.

6.1 Specification

A proof of correctness for a proof-validation algorithm can be expressed as a soundness theorem; the theorem states that if a refutation can be validated with respect to a formula by the algorithm, then the formula is unsatisfiable. Before a soundness statement can be constructed, however, one must first define what it means for an object to be a refutation. A predicate `proofp` recognizes a list of clauses that is validated by the `verify-proof` function.

```
(defun proofp (p f)          ; (p)roof, (f)ormula
  (declare (xargs :guard (formulap f)))
  (and (clause-listp p)      ; p is a list of clauses and
       (verify-proof p f))) ; is verified wrt f
```

A refutation is a `proofp` object and contains the empty clause. Refutations are recognized by the `refutationp` predicate.

```
(defconst *ec* nil) ; (e)mpty (c)lause is nil

(defun refutationp (r f) ; (r)efutation, (f)ormula
  (declare (xargs :guard (formulap f)))
  (and (proofp r f)      ; r is a verified proof and
       (member *ec* r))) ; contains the empty clause
```

The soundness theorem that represents the proof of correctness for the algorithm is constructed from the definition of a refutation and the specification for unsatisfiability (Chapter 4). The algorithm is sound if a valid refutation for a formula implies that the formula is unsatisfiable. This statement is labeled `main-theorem`.

```
(defthm main-theorem
  (implies (and (formulap f)           ; given a formula and
                (refutationp r f))     ; a valid refutation
           (not (exists-solution f)))) ; then f is UNSAT
```

6.2 Proof Overview

An overview for the proof of `main-theorem` is described here in a top-down fashion. The goal is to prove that a refutation `r` for a formula `f` implies that `f` is unsatisfiable. The proof is constructed as follows:

1. Prove the contrapositive—if there exists a solution `s` for `f`, then `r` is not a valid refutation.
2. Prove that the empty clause is not redundant with respect to `f` if `f` is satisfiable and has a solution `s`.
3. Show that every clause `c` in `r` is redundant: this is a contradiction with (2)

because the empty clause is a member of `r`. Proof by structural induction on `r`.

(a) Clauses satisfying `RUPp` are redundant.

(b) Clauses satisfying `RATp` are redundant. Case split based on the return value of `(evaluate-clause c s)`, and show that a solution exists for both `f` and `c`.

   i. If `(evaluate-clause c s)` is `true`, then `s` is a solution for `c`.

   ii. If `(evaluate-clause c s)` is `undef`, then construct a new solution `s+` that consists of `s` with an `undef` literal in `c`.

   iii. If `(evaluate-clause c s)` is `false`, then construct a new solution `s*` that is `s` with the exception that one literal in `s` has been negated.

An expansion of the predicate `refutationp` and then the predicate `proofp` in `main-theorem` allows the term containing `verify-proof` to be contraposed with the `exists-solution` term. Now that the refutation `r` is in the conclusion, the theorem is a candidate for structural induction on `r`.

```
(defthm verify-proof-induction
  (implies (and (clause-listp r)      ; given a clause list r,
                (formulap f)          ; formula f,
                (exists-solution f)   ; f is satisfiable,
                (member *ec* r))      ; and empty clause in r,
           (not (verify-proof r f)))) ; then, r cannot be validated
```

For the conclusion `(not (verify-proof r f))` to be true, some step of the recursion must fail. Each step of `verify-proof` calls `verify-clause` on the next clause in the refutation. Each successful step of the `verify-proof` function shrinks the refutation by one clause and adds the redundant clause from the refutation to the formula.

If each clause added is redundant, then there exists a solution for the extended formula in the next step. The empty clause is guaranteed to be a member of the refutation (from the hypotheses of `verify-proof-induction`) and cannot be redundant if a solution exists for the (extended) formula.

```
(defthm *ec*-lemma
  (implies (solutionp s f)        ; if s is a solution for f, then
           (not (RUPp f *ec*)))) ; empty clause does not have RUP
```

The proof of this lemma is constructed with set reasoning. If an assignment falsifies a given formula, then a superset of that assignment will also falsify the formula. Any solution must be a superset of the assignment constructed by performing unit propagation on the empty clause. Furthermore, the empty clause does not have `RATp` because there is no literal with which to perform resolution. This case is excluded by performing a `(not (atom c))` check in `verify-clause`.

Return to the induction step of `verify-proof-induction`. This is a rather odd induction step because it needs to be expressed in terms of existentials. One can prove that if there exists a solution for the formula, then there exists a solution for the formula extended with a clause from the refutation. In other words, show that the extended formula is satisfiability equivalent to the original formula.

Here the proof diverges based on whether a proof clause has `RUPp` or `RATp`. The `RUPp` case is considered in Section 6.3 and the `RATp` case in Section 6.4.

6.3 RUPp

If a clause `c` has the property `RUPp` with respect to a formula `f`, then the addition of `c` to `f`, written as `(cons c f)`, is logically equivalent to `f` and has a solution.

```
(defthm RUPp-lemma
  (implies (and (RUPp f c)                    ; if c has RUP wrt f,
                (exists-solution f)           ; f has solution,
                (formulap f)                  ; formula f,
                (clausep c))                  ; and clause c, then
           (exists-solution (cons c f))))     ; c ∧ f has solution
```

To prove this, first expand `(exists-solution f)` to obtain a witness solution. This solution can serve as a witness for the existential `(exists-solution (cons c f))` in the conclusion. The witness solution satisfies every clause in the original formula by definition, so it is sufficient to show that the witness satisfies the clause `c`.

Recall the definition of `RUPp`. The clause `c` can be replaced with an abstraction, namely the clause obtained by negating an assignment `a`, written `(negate-assignment a)`. The function `negate-assignment` is the complement of the `negate-clause` function presented in Section 5.2, so `(negate-clause (negate-assignment a))` simplifies to `a`. This gives a proper target for induction on `a`.

```
(defthm RUPp-lemma-induction
  (implies (and (falsep                            ; f is falsified
                  (evaluate-formula                ; when evaluating
                   f                               ; f wrt result of
                   (unit-propagation f a)))        ; unit prop on a
                (truep (evaluate-formula f s))     ; s satisfies f
                (formulap f)                       ; formula f
                (assignmentp a)                    ; assignment a
                (assignmentp s))                   ; assignment s
           (truep (evaluate-clause                 ; clause from
                   (negate-assignment a)           ; negating a
                   s))))                           ; satisfied by s
```

One must prove that there is a literal `l` such that `l` is a member of solution `s` and `(negate l)` is a member of `a`. Let assignment `up-a = (unit-propagation f a)`. Because `up-a` falsifies `f`, there must be a clause `c*` that is falsified by `up-a`. Because `s` satisfies `f`, `s` also satisfies `c*`. Let `l*` be the literal that is a member of both `c*` and `s`. Notice that `(negate l*)` is a member of `up-a`.

Induct on the extended assignment `up-a`. In the base case, `up-a` is equal to `a`. Therefore, `(negate l*)` is a member of `a` and `l*` is a member of `s`. In the induction step, `up-a` is `(cons ul a)` for some unit clause `uc` with unit literal `ul`. Again, there is a clause `c*` that is falsified by `up-a` but satisfied by `s`. Let `l*` be the literal that is a member of `c*` and `s`. Either `(negate l*)` is equal to `ul` or `(negate l*)` is in `a`. If `(negate l*)` is in `a`, then the proof is finished. Otherwise, `ul` is equal to `(negate l*)`, i.e. `(negate ul)` is in `s`. Consequently, `uc` was not satisfied by `ul`. All literals in `uc` not equal to `ul` are falsified by `a` from the definition of unit clause. Let `l**` be the literal in `s` that satisfies `uc`. Since `l**` cannot be `ul`, `(negate l**)` is in `a` and `l**` is in `s`, which completes the proof.

Aside. The induction for `RUPp-lemma-induction` is the most difficult part of the proof of `RUPp-lemma`. First, the induction variable would be blocked by the term `(negate-clause c)`. Several abstractions were considered, all of which negatively affected the goal `(truep (evaluate-clause c s))`. The `negate-assignment` abstraction lets one perform the correct induction without significantly changing the goal. Second, the induction itself is rather subtle because of the custom measure provided to `unit-propagation` (Section 5.4). The assignment grows during every recursive call of `unit-propagation`, but the number of `undef` clauses decreases.

6.4 RATp

One must prove that if there exists a solution `s` for formula `f` and a clause `c` has the `RATp` property with respect to `f` and a literal `l` in `c`, then there is a solution for the formula `(cons c f)`.

```
(defthm RATp-lemma
  (implies (and (formulap f)              ; for formula f
                (clausep c)               ; and clause c,
                (member l c)              ; if l ∈ c,
                (exists-solution f)       ; f has solution,
                (RATp f c l))             ; and c has RAT on l,
           (exists-solution (cons c f)))) ; c ∧ f has solution
```

Let assignment `s` satisfy `f`. Case split on `(evaluate-clause c s)`.

- `true`: There exists a solution for `(cons c f)`, namely `s`.
- `undef`: Choose a literal `l+` in `c` such that `(evaluate-literal l+ s)` evaluates to `undef`. Let the assignment `s+` be `(cons l+ s)`. The evaluation of `(evaluate-clause c s+)` will return `true` because `(evaluate-literal l+ s+)`

returns `true`. Consider some clause `c1` in `f`. One can show `(evaluate-clause c1 s+)` returns `true`, because `(evaluate-clause c1 s)` returns `true`. Therefore, `f` is satisfied by `s+` and there exists a solution for `(cons c f)`, namely `s+`.

- `false`: Create a new assignment `s*` such that `(evaluate-literal l s*)` is `true` by removing the literal `(negate l)` from `s` and adding `l` to `s`. By construction, the term `(evaluate-clause c s*)` is `true`. One must then prove that `(evaluate-clause c1 s*)` is `true` for all `c1` in `f`. This follows from the solution reconstruction method (Section 2.2.2).

Consider a clause `c1` in `f`. If literal `(negate l)` is not a member of `c1`, then one can show that `(evaluate-clause c1 s*)` is still true (because `l` is the only literal that changed in `s`). Recall `c` has `RATp` so the resolvent `r` computed by `(resolution l c c1)` has the property `RUPp` with respect to `f`. By the lemma `RUPp-lemma`, `r` is also satisfied by `s`. Therfore, there exists a literal `lr` in `r` such that `(evaluate-literal lr s)` is `true`. Now, `lr` cannot be in `c` because `(evaluate-clause c s)` is `false`. Since `lr` is in `r` and not in `c`, then `lr` is in `c1`. Furthermore, `lr` cannot be equal to `(negate l)` because `(negate l)` is not in `r` by the definition of resolution. Therefore, `(evaluate-clause c1 s*)` is `true`, and there exists a solution for `(cons c f)`, namely `s*`.

Aside. One key observation during the development of the proof for `RATp-lemma` was the need for a case split on `(evaluate-clause c s)`. Previous attempts tried to use an induction on the clause list from `RATp1` with `(not (truep (evaluate-clause` `c s)))` as a hypothesis. This was insufficient. Feedback from ACL2 indicated that a full three-way case split should be performed. This strengthened the condition to `(falsep (evaluate-clause c s))`.

Another subtle part of the proof is the case of `tautologyp` for the resolvent during an induction of the clause list in `RATp1`. In this proof, one must find a conflicting literal in the resolvent and then show that the existence of a conflicting literal implies that a clause from the formula is satisfied by the modified solution.

Chapter 7

Implementation

When viewed as an executable model, the algorithm described in Chapter 5 performs very poorly. Values (such as literals) are unbounded and "bignum" arithmetic (for values not known to fit in 64 bits) is performed during execution; all accesses and updates to the list-based data structures require linear time to traverse and reconstruct the list structures; the unit propagation routine searches all clauses in the database; and unit clauses in the original formula are rediscovered on every redundancy check. If the goal is to achieve execution times similar to DRAT-trim (Chapter 3), then all of these problems need to be addressed. Every time a modification is made to the algorithm, a new proof of correctness must be established. Alternatively, a proof of equivalence to the original algorithm is sufficient. In this chapter, a first step towards a more efficient implementation is described. In the next chapter, a proof of equivalence relating the new implementation to the old is presented.

The development of a new implementation is completed in two parts. First, a new ACL2 data structure called `farray` is designed to combine efficient execution and a convenient theory. Second, the `assignmentp` structure from the previously defined RAT validation algorithm is modeled using `farray`.

7.1 Field-addressable Arrays

Much of the reasoning that is done in ACL2 is constructive. That is, a model is defined by predicates that recognize an instance of the model. The instance of the model is called the "state" and operations describe valid modifications to the state. ACL2 has many different built-in and user-defined systems for representing a state-based model. Each of these systems have strengths and weaknesses in relation to the execution efficiency and ease of use of the supporting theory, but none are scalable enough to represent a problem of this magnitude. A new data structure, called `farray`, was designed to accomplish this task. This section provides motivation for the `farray` data structure by examining existing list and STOBJ models of state, and it details the `farray` model and supporting theory.

7.1.1 Motivation

An example of state-based modeling using `cons`-lists and single-threaded objects with arrays is detailed below. Then, the execution efficiency and reasoning support for both of these methods is explored.

7.1.1.1 Example List-based Model

Consider an example state-based model constructed with lists (note that the functions `nth` and `update-nth` use zero-based numbering while the fields use one-based numbering):

```
(defun foo-model-listp (x)          ; (x) object
  (declare (xargs :guard t))
  (and (true-listp x)               ; true-list to represent state
       (equal (len x) 3)            ; 3 fields total
       (integer-listp (nth 0 x))    ; field 1 is list of integers,
       (equal (len (nth 0 x)) 10)   ; and has length 10
       (integerp (nth 1 x))         ; field 2 is scalar integer
       (integer-listp (nth 2 x))    ; field 3 is list of integers,
       (equal (len (nth 2 x)) 5)))  ; and has length 5
```

The function `foo-model-listp` recognizes an object that contains one sub-list of length 10, a scalar, and a sub-list of length 5. In other words, given an object, `foo-model-listp` will verify that the object is a valid representation of state. This model chooses to unify the various fields into one state object, as opposed to the model of the RAT validation algorithm which separates assignments, formulas, and proofs.

Basic operations on a state object should be well-defined, namely reading values from and writing values to a state. An example write operation for the model above might be defined as:

```
(defun field3-write-list (i v foo)        ; (i)ndex, (v)alue,
                                          ; (foo) object
  (declare (xargs :guard
                  (and (foo-model-listp foo)
                       (integerp i)
                       (<= 0 i)
                       (< i 5)
                       (integerp v))))
  (let* ((field3                          ; let field3 be
          (nth 2 foo))                    ; foo[2]
         (new-field3                      ; let new-field3 be
          (update-nth i v field3))        ; ( field3[i] = v )
         (new-foo                         ; let new-foo
          (update-nth 2 new-field3 foo))) ; foo[2] = new-field3
    new-foo))                             ; return new-foo
```

Note that the `nth` and `update-nth` functions used in the definition of the function `field3-write-list` each require a linear amount of time: they recur until an index `i` is reduced to zero. In the case of `update-nth`, the list is reconstructed once the appropriate modification has been made. Thus, a write to a list-based model will need to rebuild the field *and* the state. This is an extremely expensive operation to simply change one value in a field.

7.1.1.2 Example STOBJ-based Model

ACL2 STOBJs (Single-Threaded OBJects) have the ability to interface with Lisp arrays from ACL2, which improves the execution efficiency of ACL2 programs using this data structure. This provides constant-time lookup for STOBJ fields and arrays, as opposed to lists which require linear-time lookup for `nth` and `update-nth` operations.

Reasoning about STOBJs is relatively straightforward because the logical model is constructed of lists. The cost of the this approach is that STOBJs are syntactically limited so that updates to the STOBJ data structure are serialized (hence "single-threaded").

Consider this STOBJ definition designed to represent the same model described by `foo-model-listp`:

```
(defstobj foo-model-stobj
  (field1 :type (array integer (10)) ; field1 is 10 integer array,
          :initially 0)              ; values initialized to 0
  (field2 :type integer              ; field2 is scalar integer,
          :initially 0)              ; initialized to 0
  (field3 :type (array integer (5))  ; field3 is 5 integer array,
          :initially 0))             ; values initialized to 0
```

This STOBJ definition introduces a recognizer and creator for the STOBJ and introduces a recognizer, an accessor, an updater, a length operator, and optionally a resizer for each field in the STOBJ (13 function definitions in total). During execution, these functions interact with the underlying LISP representation, providing constant-time reads from and writes to the state. For reasoning, however, the definitions are equal to those of a list-based model. In fact, the logical definition of `foo-model-stobjp` is equal to the definition of `foo-model-listp` above. ACL2 guarantees this abstraction is sound.

7.1.1.3 Proof Efficiency and Proof Convenience

A theory needs to be constructed in both the list-based and STOBJ-based models describing how each of these functions interacts with each other function. This effort grows quadratically in the number of fields. If the model is known ahead of time and will remain static, perhaps this is a reasonable approach.

Suppose that `foo-model-stobj` is under active development and a new field, called `field4`, is added to the STOBJ. This small change adds 4 new functions associated with `field4`, but it also creates a need for theorems relating `field4` operations to every other field *and* theorems relating every other field to `field4`. This approach is not feasible when a state-based model is under active development. This should be a relatively simple modification to the data structure, but the developer has to manage a quadratic increase in both the amount of theorems that need to be written and the certification time of the book containing this data structure.

In other words, these list-based and STOBJ-based approaches lead to poor "proof convenience", i.e. the ease of expressing, maintaining, and using the theorems in a mechanical proof; and poor "proof efficiency", i.e. the time it takes the theorem prover to replay a proof or certify a book.

In the list-based example, an ACL2 user-contributed library (called a "book") introduces a data structure called `defaggregate` that defines state-based models containing various fields. The defaggregate event defines creators, accessors, updaters, etc. and automatically proves some theorems about the data structure. Execution efficiency is still limited by the list-based implementation, however.

A macro-based approach is also possible to automatically generate theorems associated with list or STOBJ fields. This method can benefit from the execution efficiency of STOBJs and improves proof conveniences of the theory; but the proof efficiency is still limited, especially for larger, more complicated models.

7.1.2 farray Definition

Instead of the previously mentioned approaches, a STOBJ can be defined with just one field: an array for memory. This field can be subdivided further into other arrays and scalars. In this way, a single array in a STOBJ can simulate a STOBJ with multiple fields. This approach mimics C-style programs where there is no notion of memory other than as a single, flat array. Pointers may indicate the start of a particular structure (like a C struct or array), but the underlying memory is still flat. This approach is transparent and relies on the user to perform memory management.[1]

Furthermore, the notion of a field can be abstracted during proof. By creating parameterized functions that read and write to any field (called "uniform access"), proofs can be completed just once for all combinations of reads and writes to different fields, thereby improving proof convenience and proof efficiency.[2] The addition of a field to the structure does not create any extra proof effort. This solves the problem of quadratic increase of theorems for STOBJ data structures.

A new data structure called `farray` was developed for this purpose. A STOBJ named `st` (short for "state") contains one array field of 60-bit signed integers named `mem` (short for "memory"). The `farray` is also defined by a `start` index into the array where the data structure begins, so that there can be multiple `farray` data structures in the same STOBJ. The number of fields in the `farray` is stored at the

[1]The single-array STOBJ approach was suggested by Warren A. Hunt, Jr. in order to create ACL2 programs that more closely resemble C programs.

[2]The "uniform access" approach is a generalization of an abstraction J S. Moore used in the M1 model.

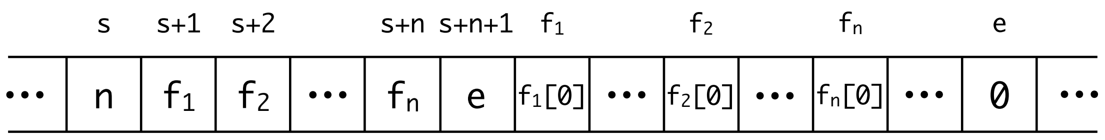

Figure 7.1: This diagram represents a generic `farray` object residing in a section of the `mem` array. The value above a box represents an address of the `mem` array and the value inside a box represents the value stored at a `mem` array address. The `farray` begins at address `s` which contains the number of fields `n`. The field table occupies the next `n+1` locations which store the addresses $\texttt{f}_1$ through $\texttt{f}_n$ where each field begins. The last field table address is `s+n+1` and contains the address `e` indicating the end of the `farray` structure. The data for each field is stored at the address $\texttt{f}_1$ through $\texttt{f}_n$. The length of each field is defined to be the difference between the low address of the field and the low address of the subsequent field.

`start` location and is accessible with a function `num-fields`. The next $n+1$ locations are monotonic offsets into the array where each field $1, ..., n$ begins. The last location is where the `farray` structure ends. This part of the structure is called the *field table*. Fields are referenced by integer values, but constants may be defined to create *field identifiers* (e.g., a `*field1*` constant in place of the integer 1). Each field begins at the offset in the field table and spans the indices up to the subsequent entry in the field table; the field length can be retrieved with the `flength` function. A scalar field has length 1. Fields and offsets within fields are recognized by `fieldp` and `field-offsetp` predicates, and the number of fields can be retrieved with the `num-fields` function. A diagram of a generic farray is displayed in Figure 7.1.

A field table for an `farray` of `n` fields starting at address `s` is recognized by the recursive predicate `field-tablep` which enforces the following invariants for an address `i` from addresses `s+1` to `s+n+1`:

1. the value at `i` is an integer,
2. the value at `i` is greater than or equal to 0,
3. the value at `i` is less than the length of array `mem`,
4. the value at `i` is greater than the value at address `s+n+1`, and
5. the value at `i` is less than the value at address `i+1`.

With the definition of `field-tablep`, one can define the predicate `farrayp` that recognizes an `farray` in a state `st`. The start `s` of the `farray` is also required as input, as multiple `farray` objects can exist within `st`.

```
(defun farrayp (s st)                       ; (s)tart, (st)ate
  (declare (xargs :guard t
                  :stobjs (st)))
  (and (stp st)                             ; valid STOBJ st
       (integerp s)                         ; s is integer
       (<= 0 s)                             ; s >= 0
       (< s (mem-len st))                   ; s in range of mem
       (integerp (memi s st))               ; num fields is int
       (<= 0 (memi s st))                   ; num fields s >= 0
       (< (+ s (memi s st) 1)               ; end in range
          (mem-len st))                     ; of mem
       (< (memi (+ s (memi s st) 1) st)     ; value at end
          (mem-len st))                     ; in range of mem
       (sb60p (mem-len st))                 ; length of mem < 2^59
       (field-tablep (+ s 1)                ; field table from s+1
                     (+ s (memi s st) 1)    ; to end position
                     st)))                  ; in st
```

The functions `fread` and `fwrite` are (at the time of this publication) the only method of accessing and updating values in the `farray`. The `fread` function takes a field, a field offset, the start of the `farray`, and the state as input and returns the value at that offset within the field.

```
(defun fread (f o s st)       ; (f)ield, (o)ffset, (s)tart, (st)ate
  (declare (xargs :guard (and (farrayp s st)
                              (fieldp f s st)
                              (field-op o f s st))
                  :stobjs (st)))
  (memi (+ (memi (+ s f) st) ; read field table entry for f,
           o)                ; add offset o for new address,
        st))                 ; and read the address
```

Note that this function is parameterized on the field in order to provide uniform access. The downside of this process is that each `fread` operation from the state now requires two reads from the array (the `memi` calls) and two additions. When compared to a multiple-field STOBJ-based approach, this may seem excessive, but it is still much more efficient than the list-based approach.

The `fwrite` function accepts a field, a field offset, a value, the start of the `farray`, and the state as input and returns a new state.

```
(defun fwrite (f o v s st)    ; (f)ield, (o)ffset, (v)alue,
                              ; (s)tart, (st)ate
  (declare (xargs :guard (and (farrayp s st)
                              (fieldp f s st)
                              (field-offsetp o f s st)
                              (sb60p v))
                  :stobjs (st)))
  (!memi (+ (memi (+ s f) st) ; read field table entry for f,
            o)                ; add offset o for new address,
         v st))               ; and write value v to address
```

To provide this abstraction, `fwrite` pays the extra cost of an array read and two additions on top of the cost of the write.

Proper theorems establish (rewrite) rules in ACL2 that help automate reasoning about implementations using `farray`. The collection of rules is often called a *theory*. The major points of the `farray` theory are listed here:

1. `fread` always returns a signed-byte 60-bit integer,
2. `fwrite` always returns an `farrayp`,
3. `fwrite` does not affect `flength`,
4. an `fwrite` followed by an `fread` results in the value written if the fields and offsets are equal,
5. two `fwrite` operations to the same field and offset are the same as the last write,
6. an `fwrite` of the value already at the field and offset is the same as no `fwrite`,
7. two `fwrite` operations to distinct fields can be ordered,
8. two `fwrite` operations to the same field with distinct offsets can be ordered,
9. the number of fields is not affected by `fwrite`,
10. `fieldp` is not affected by `fwrite`, and
11. `field-offsetp` is not affected by `fwrite`.

One of the notable theorems is that an `fwrite` followed by an `fread` results in the value written only if the fields and offsets are equal (number 4 in the list above).

```
(defthm fread-fwrite
  (implies                              ; if there is
   (and (farrayp s st)                  ; farray at s in state st,
        (fieldp f1 s st)                ; field f1,
        (field-offsetp o1 f1 s st)      ; field offset o1,
        (fieldp f2 s st)                ; field f2, and
        (field-offsetp o2 f2 s st))     ; field offset o2, then
   (equal (fread f2 o2 s                ; reading f2 at o2 after
                 (fwrite f1 o1 v s st)) ; writing v to o1 at f1,
          (if (and (equal f1 f2)        ; is equal to v if f1=f2
                   (equal o1 o2))       ; and o1=o2
              v
            (fread f2 o2 s st)))))      ; and f2 at o2 otherwise
```

This theorem is often called a "read over write" theorem and allows the ACL2 rewriter to simplify a term where an `fread` surrounds an `fwrite`.

A predicate `field-memberp` is included in the definition of `farray` and behaves similarly to the ACL2 built-in function `member`. This function checks a range of a field for a given value. This is useful when a certain range of a field corresponds to a list (Chapter 8).

The `farray` data structure is not specific to the work on mechanical verification of SAT validation algorithms. It is designed to be useful for anyone interested in fast, executable, mechanically-verified code or anyone that is modeling C-like programs in ACL2.

7.2 Literal Encodings

The move to an array-based implementation requires that values be bounded. The decision was made to use signed 60-bit integers as the basis for values in the new implemenation. Clozure Common Lisp (CCL) is the target LISP compiler for this project and uses 3 bits of every 64-bit integer to record typing information. The 60-bit signed integer type is sufficient for the size of variables in a SAT solver or proof checker and stays under the limit of what CCL can store in a 64-bit value, eliminating the problem of "bignum" arithmetic in compiled code.

Another consideration is the encoding of literals in the array. In previous verification efforts, a straightforward encoding of literals $\bar{x}_2, \bar{x}_1, x_1, x_2$ to integers $-2, -1, 1, 2$ was acceptable because efficiency was not a priority. However, the move to a more efficient system for assignments necessitates a change in encoding. Four encoding methods were considered for the new array-based assignment structure; the purpose of each encoding is to determine one of three states for a literal (usually obtained from a clause) given an assignment. Encodings must maintain the uniqueness of literals in the assignment and disallow conflicting literals. Below, each assignment encoding is evaluated based on the access/update cost, negation cost, and invariant.

Method 1 A compact assignment encoding is shown in Figure 7.2. In this method, the literals lit and $-lit$ share the same location in the assignment. The values at the location are in the set $\{0, 1, -1\}$ and correspond to the cases where neither positive nor negative literal is assigned, the positive literal is true, or the negative literal is true, respectively. Given a literal, the absolute value of the literal is the index into the assignment. The value at that location must then be interpreted based on the positive/negative status of the literal. This is important because these are expensive operations that must be performed every time a literal is accessed or updated in the assignment array. The benefit of this representation is that it is compact (only requires space equal to the number of variables) and there is no extra invariant on the assignment other than the values are in the set $\{0, 1, -1\}$.

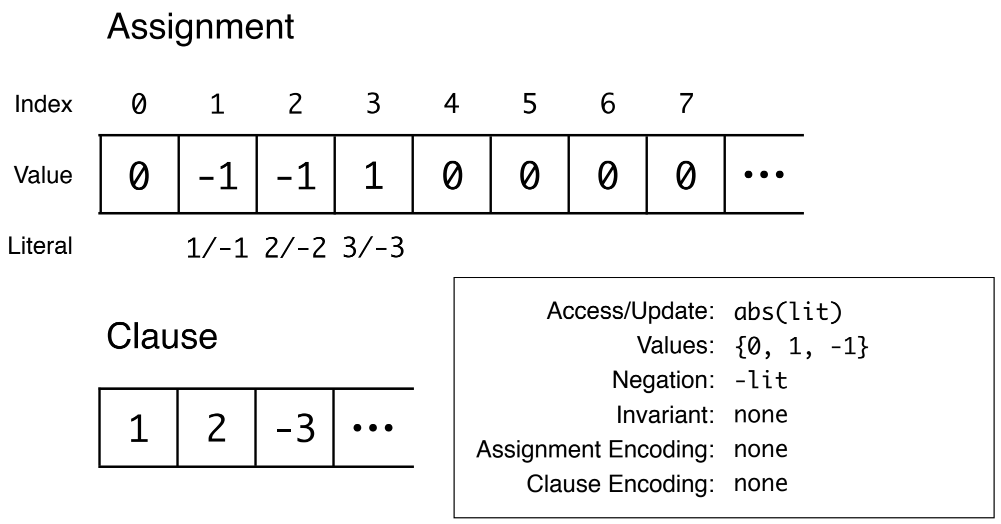

Figure 7.2: **Method 1** is a compact encoding where positive and negative forms of a literal share the same location in the assignment. The assignment `(-1 -2 3)` and the clause `(1 2 -3)` are displayed in the figure.

Method 2 One can encode positive and negative literals separately into positive indices by using the assignment encoding in Figure 7.3. This method doubles the space requirements for an assignment but removes the interpretation cost associated with Method 1. Given a literal (from a clause), the literal must be encoded to obtain an index into the assignment. Values stored in the assignment are either 0 or 1 corresponding to the cases where the literal is assigned to true or not assigned to true. This introduces a new invariant in that the value of the location associated with the positive literal and the negative literal cannot both be equal to 1 (assigned to true). Accesses and updates are even more expensive than in the previous method because of the complexity of the encoding function.

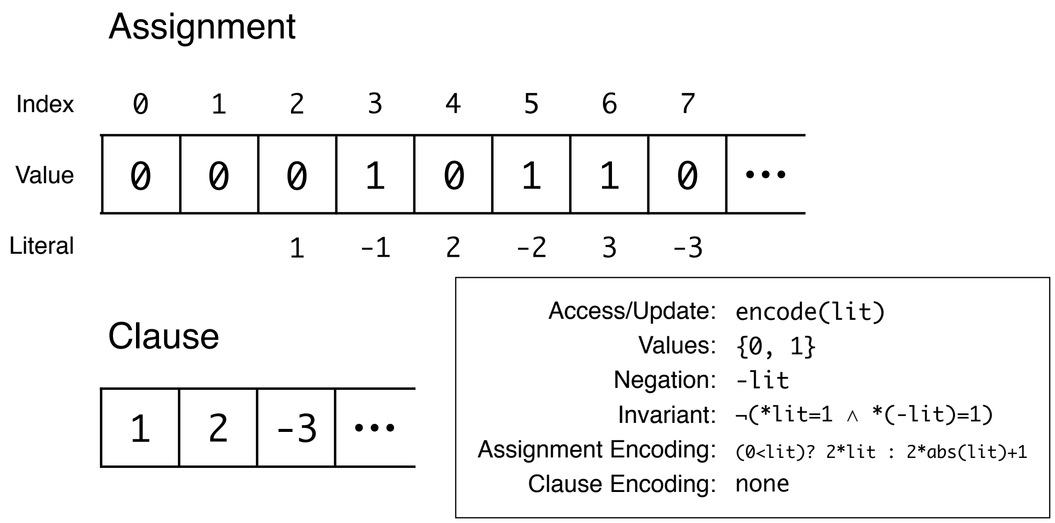
Assignment
Index 0 1 2 3 4 5 6 7
Value 0 0 0 1 0 1 1 0 •••
Literal 1 -1 2 -2 3 -3
Clause
1 2 -3 •••
Access/Update: encode(lit)
Values: {0, 1}
Negation: -lit
Invariant: ¬(*lit=1 ∧ *(-lit)=1)
Assignment Encoding: (0<lit)? 2*lit : 2*abs(lit)+1
Clause Encoding: none

Figure 7.3: **Method2** separates the locations for positive and negative literals to avoid the interpretation cost associated with Method 1. This comes at the cost of (double) space and a more complicated access/update function. The assignment `(-1 -2 3)` and the clause `(1 2 -3)` are displayed in the figure.

Method 3 The need to encode a literal during runtime can be avoided if all clauses have been encoded using the same scheme as shown in Figure 7.4. In this method, clauses contain encoded literals, eliminating the need to encode literals into indices. This is the most efficient method of accessing and updating the assignment, but it comes at the cost of changing the internal representation of literals in the formula. Negation of literals in the clause is now performed by bitwise exclusive-or with the constant 1. This is the method that is used in the implementation described in Section 7.3.

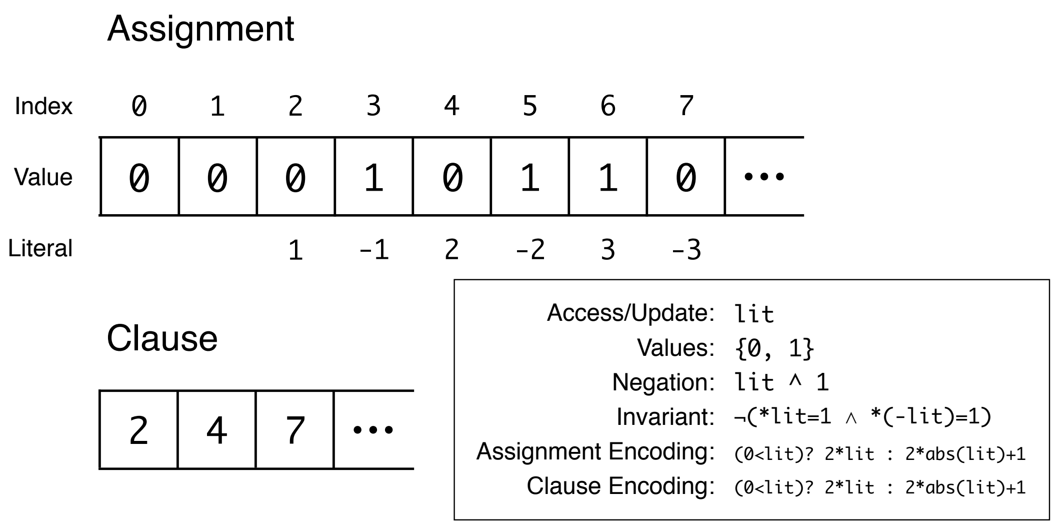
Assignment
Index
0
1
2
3
4
5
6
7
Value
0
0
0
1
0
1
1
0
•••
Literal
1
-1
2
-2
3
-3
Clause
2
4
7
•••
Access/Update: lit
Values: {0, 1}
Negation: lit ^ 1
Invariant: ¬(*lit=1 ∧ *(-lit)=1)
Assignment Encoding: (0<lit)? 2*lit : 2*abs(lit)+1
Clause Encoding: (0<lit)? 2*lit : 2*abs(lit)+1

Figure 7.4: **Method 3** encodes the literals in each clause to reduce the access and update time for the assignment. This greatly improves performance at the cost of changing the way literals are stored elsewhere in the program. The assignment `(-1 -2 3)` and the clause `(1 2 -3)` are displayed in the figure.

Method 4 Finally, given an index i in the center of the assignment, literals can be kept unencoded with relatively little cost for accesses and updates (Figure 7.5). This method rivals the efficiency of Method 3 and allows the internal representation of literals in the formula to be the same as the external representation in problem files. It requires the knowledge of a midpoint i within the array; this is easy for a language like C that has access to pointers, but it is much harder to represent in ACL2. This is the method used in the implementation of DRAT-trim from Chapter 3.

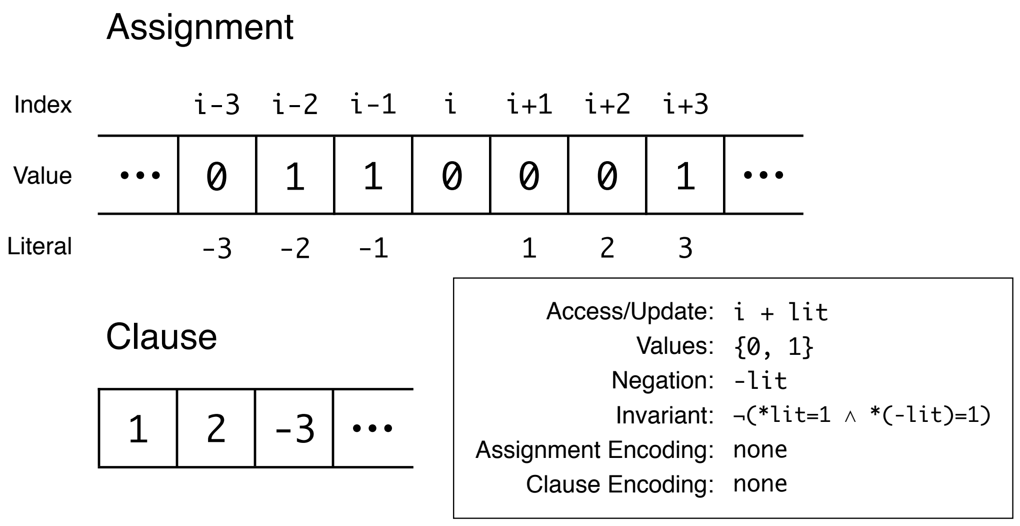

Figure 7.5: **Method 4** uses a midpoint into the assignment array i to provide efficient accesses and updates with unencoded literals. This approach is more suited to C (where i can be a pointer) than it is to ACL2. The assignment `(-1 -2 3)` and the clause `(1 2 -3)` are displayed in the figure.

7.3 Array-based Assignments

In the satisfiability specification from Chapter 4, assignments are defined to be lists of unique, non-conflicting literals, recognized by the predicate `assignmentp`. In Chapter 5, the RAT proof validation algorithm has two forms of updates to data structures: creation/modification of assignments and moving clauses from the proof to the formula. The former represents far more updates that the latter. As a first step to improving the efficiency of the algorithm, assignments are transitioned to an array-based data structure for constant-time access and update. This change is implemented using `farray`.

In this section, the definition for an `assignment-st` structure is described. Four basic operations on the state are then defined. Finally, an example of the proof work necessary to maintain one of the recursive invariants is presented.

7.3.1 Definition

The array-based model of an assignment is called an `assignment-st` (short for "assignment state"), and it is defined using an `farray` with four fields. The first field is addressed by the field identifier `*num-vars*` and references a scalar that records the number of variables. This field is used to define the maximum variable that will be encountered during verification and is important when defining the length of the lookup table that maps literals to truth assignments. The second and third fields construct a stack and are addressed by the field identifiers `*stack-end*` and `*stack*`. The `*stack*` field is an array of length `(1+ *num-vars*)` and the `*stack-end*` field is a scalar that contains an offset to the next empty position in the `*stack*`. The fourth field in `assignment-st` uses the field identifier `*lookup*` and is an array that acts as a lookup table. All literals that are less than or equal to `(+ 2 (* 2 *num-vars*))` are valid field offsets into the `*lookup*` field. Figure 7.6 diagrams the layout of a `assignment-st` structure for 3 variables.

The `assignment-st` structure represents assignments as both a stack and a lookup table. These two substructures are synchronized: any state update must be recorded in both substructures. While the lookup table is the only structure necessary for constant time accesses and updates, the stack adds the ability to encode temporal information. The stack not only contains the literals that are assigned to true, but also contains the order in which they were assigned to true. This is important for reasoning about routines like unit propagation. The order that unit literals are added to the assignment plays a major part in the equality of two different unit propagation routines.

There are 19 high-level invariants maintained by the `assignment-stp` recognizer predicate for `assignment-st` structures, 7 of which are recursive invariants. The full predicate is displayed in Figure 7.7 and the recursive invariants are:

1. The values in the stack are literals.
2. The values in the stack are unique.
3. The values in the stack are non-conflicting.
4. The values in the lookup table correspond with the values in the stack.
5. The values in the stack are within the range specified by the number of variables.

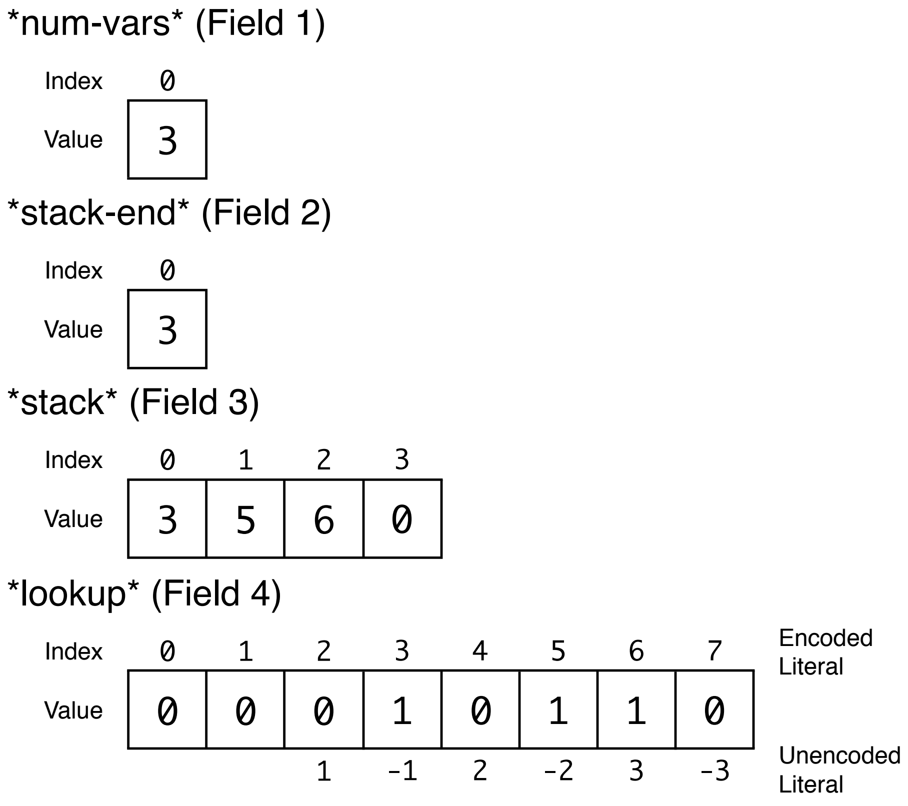

Figure 7.6: The assignment `(-1 -2 3)` (encoded as `(3 5 6)`) is displayed in the `assignment-st` model. The underlying representation for this model is an `farray` of four fields, each of which are identified by constants. The `*num-vars*` field contains the largest variable that can exist in the model. The `*stack-end*` field contains a valid offset into the `*stack*` field. This offset indicates the next open location in the stack. In this example, the stack is full because the offset is the last position in the stack. The `*stack*` field contains all (encoded) literals that have been assigned to true. In this example, the literals `3` (the encoded literal `-1`), `5` (`-2`), and `6` (`3`) are true. Finally, the `*lookup*` field is an array that is indexed by encoded literals. If the value at an index is equal to one, then the literal has been assigned to true.

6. The values in the stack correspond with the values in the lookup table.
7. The number of literals assigned in the lookup table is the same as the number of literals in the stack.

7.3.2 Operations

Recall the four basic operations on list-based assignments (see Section 4.2). Given an assignment `a`, the operation `(member l a)` tests if literal `l` is assigned to true with respect to `a`. The operation `(cons l a)` assigns the literal `a` to true. The operation `(cdr a)` unassigns the most recently assigned literal. The assignment `nil` unassigns all literals and starts with an empty assignment.

These four basic operations are implemented for state-based assignments as `assignedp`, `assign-lit`, `unassign-one`, and `unassign-all`. These functions are detailed below.

Reads from the `assignment-st` structure are performed by the `assignedp` predicate. The function reads the value in the `*lookup*` field at the offset equal to the literal. If the value is equal to `1`, then the literal has been assigned to true.

```
(defun assignedp (l st)            ; (l)iteral, (st)ate
  (declare (xargs :guard (and (assignment-stp st)
                              (literalp l)
                              (lit-in-rangep l st))
                  :stobjs (st)))
  (equal (fread *lookup* l *s* st) ; read from lookup table
         1))                       ; at offset l, compare to 1
```

Note that only one `fread` from the lookup table is necessary. No recusion is necessary and this operation is perfomed in consant time.

```
(defun assignment-stp (st)                  ; (st)ate
  (declare (xargs :guard t
                  :stobjs (st)))
  (and
   (farrayp *s* st)                         ; farray
   (equal (num-fields *s* st)               ; four fields in farray
          4)
   (fieldp *num-vars* *s* st)               ; num-vars is field 1
   (equal (flength *num-vars* *s* st)       ; num-vars is length 1
          1)
   (let ((n (fread *num-vars* 0 *s* st)))   ; let n be num-vars
     (and
      (<= 0 n)                              ; n is natural
      (fieldp *stack-end* *s* st)           ; stack-end is field 2
      (equal (flength *stack-end* *s* st)   ; stack-end is length 1
             1)
      (fieldp *stack* *s* st)               ; stack is field 3
      (equal (flength *stack* *s* st)       ; stack is length n+1
             (1+ n))
      (fieldp *lookup* *s* st)              ; lookup is field 4
      (equal (flength *lookup* *s* st)      ; lookup is length 2n+2
             (+ 2 (* 2 n)))
      (let ((se (fread *stack-end*          ; let se be stack-end
                       0 *s* st)))
        (and
         (field-offsetp se *stack* *s* st) ; se is field offset
         (stack-contains-literalsp          ; stack values are
          (1- se) st)                       ; literals
         (stack-uniquep (1- se) st)         ; stack values unique
         (stack-not-conflictingp            ; stack values are
          (1- se) st)                       ; not conflicting
         (lookup-corresponds-with-stackp    ; lookup matches
          2 st)                             ; stack
         (stack-in-rangep (1- se) st)       ; stack values bounded
         (stack-corresponds-with-lookup-p   ; stack matches
          (1- se) st)                       ; lookup
         (equal (count-assigned 2 st)       ; number assigned in
                se)                         ; lookup equals se
         ))))))
```

Figure 7.7: Definition of the `assignment-stp` predicate.

The `assign-lit` function accepts a state and a literal and returns a new state where the lookup table entry for the literal has been set, the literal has been added to the top of the stack, and the offset to the top of the stack has been incremented.

```
(defun assign-lit (l st)                 ; (l)iteral, (st)ate
  (declare (xargs :guard ...
                  :stobjs (st)))
  (let* ((se (fread *stack-end* 0        ; let se = first empty
                     *s* st))            ; stack location
         (st (fwrite *lookup* l 1        ; write 1 at offset l
                      *s* st))           ; in lookup table
         (st (fwrite *stack* se l        ; write l at offset se
                      *s* st))           ; in stack
         (st (fwrite *stack-end* 0 (1+ se) ; write 1+se to first
                      *s* st)))          ; empty stack location
    st))                                 ; return state
```

This operation requires one read from and three writes to the assignment state. Again, each of these reads and writes take contant time. This is perhaps the most important of the four basic operations on an `assignment-st` structure.

The third operation on a state-based assignment is to unassign the most recently assigned literal. This is performed by the `unassign-one` function that accepts a state and returns a new state where the literal on the top of the stack is cleared from the lookup table and the offset to the top of the stack has been decremented.

```
(defun unassign-one (st)                ; (st)ate
  (declare (xargs :guard ...            ; guards omitted
                  :stobjs (st)))
  (let* ((1-se (1- (fread *stack-end* 0 ; let 1-se = one below
                          *s* st)))     ; first empty stack loc
         (l (fread *stack* 1-se         ; let l be last lit in
                   *s* st))             ; stack
         (st (fwrite *lookup* l 0       ; write 0 at offset l
                     *s* st))           ; in lookup table
         (st (fwrite *stack-end* 0 1-se ; reduce stack by writing
                     *s* st)))          ; 1-se to stack end,
    st))                                ; then return state
```

This operation requires two reads from and two writes to the assignment state. The `unassign-one` function is never used on its own in practice; it represents one step in the process of removing all literals from the assignment.

The `unassign-all` function accepts a state and an offset to the top of the stack and returns a new state where all literals have been unassigned. This is perfomed by recursively unassigning a single literal until the stack is empty.

```
(defun unassign-all (i st)                  ; (i)ndex, (st)ate
  (declare (xargs :guard ...                ; guards omitted
                  :stobjs (st)
                  :measure (nfix i)
                  ))
  ;; all of the following (m)ust (b)e (t)rue to continue
  (if (not (mbt (and (assignment-stp st)
                     (equal (fread *stack-end* 0
                                   *s* st)
                            i)
                     (or (field-offsetp i *stack*
                                        *s* st)
                         (equal i 0)))))
      st
    (if (equal i 0)                         ; if i=0, nothing to
        st                                  ; unassign, done
      (let* ((st (unassign-one st))         ; unassign one lit
             (st (unassign-all (1- i) st))) ; then unassign rest
        st))))                              ; and return state
```

This operation is linear in the number of assigned literals. Note that this operation is less efficient than the list-based counterpart, where an assignment is replaced by the symbol `nil`.

These four operations each require the same number of reads and writes to the state as the basic operations in the C-based utility DRAT-trim.

7.3.3 Maintaining Invariants

The `assignment-stp` predicate describes a well-formed assignment in the new, state-based model and includes several different invariants, some of which are defined recursively. One of the most challenging parts of creating a more efficient implementation for assignments is proving that these invariants are maintained during the basic operations on the state.

Consider the `assign-lit` operation that accepts a literal and a state and returns a new state where the literal has been assigned to true. One must prove that this operation returns a valid state that satisfies the `assignment-stp` predicate. This is only true if the literal is not already assigned to true. This statement is represented by the theorem:

```
(defthm assignment-stp-assign-lit
  (implies                               ; if
   (and (assignment-stp st)             ; valid assignment,
        (literalp l)                    ; literal l,
        (field-offsetp l *lookup*       ; l is valid offset into
                       *s* st)          ; lookup table,
        (not (field-memberp             ; l is not a member
              l                         ; of the stack,
              (1- (fread *stack-end* 0
                         *s* st))
              0 *stack* *s* st))
        (not (field-memberp             ; and negation of l
              (negate l)                ; is not a member of
              (1- (fread *stack-end* 0 ; stack, then
                         *s* st))
              0 *stack* *s* st)))
   (assignment-stp (assign-lit l st))) ; valid assignment after
  ...)                                  ; assigning l to true
```

In order to prove this theorem, each of the 19 high-level invariants (including 7 recursive invariants) must be true after the `assign-lit` operation has completed.

For the purposes of this example, consider the invariant that states that elements of the stack must be literals. This invariant is checked recursively by the predicate `stack-contains-literalsp`. This invariant holds after writing a literal to the stack and increasing the offset of the next empty stack location.

```
(defthm stack-contains-literalsp-assign-lit
  (implies                                     ; if
   (and (farrayp *s* st)                       ; farray at start,
        (fieldp *stack-end* *s* st)            ; stack-end is field,
        (fieldp *stack* *s* st)                ; stack is field,
        (fieldp *lookup* *s* st)               ; lookup is field,
        (field-offsetp                         ; value at stack-end
         (fread *stack-end* 0 *s* st)          ; is offset into
         *stack* *s* st)                       ; stack,
        (field-offsetp l *lookup* *s* st)      ; l offset for lookup,
        (literalp l)                           ; l is literal
        (stack-contains-literalsp              ; and stack contains
         (1- (fread *stack-end* 0 *s* st))     ; only literals before
         st))                                  ; assignment, then
   (stack-contains-literalsp                   ; stack contains only
    (fread *stack-end* 0 *s* st)               ; literals after
    (assign-lit l st))))                       ; assignment
```

The last literal in the stack before the assign-lit operation is at one less than the *stack-end* value, and the last literal in the stack after the assign-lit operation is at the *stack-end* value.

To prove this theorem, one must reason about how each of the three writes to the state in the assign-lit operation affects this predicate. This is relatively easy for this pairing of predicate and operation. The only fields that matter are *stack* and *stack-end*; any other field is irrelevant.

```
(defthm stack-contains-literalsp-fwrite-diff-field
  (implies                                  ; if
   (and (not (equal f *stack*))             ; field f is not stack,
        (stack-contains-literalsp i st)     ; stack has only lits,
        (fieldp f *s* st)                   ; f is a field,
        (field-offsetp o f *s* st)          ; o is an offset for f,
        (sb60p v))                          ; and v is a value, then
   (stack-contains-literalsp                ; stack has only lits
    i                                       ; after writing v to
    (fwrite f o v *s* st))))                ; field other than stack
```

If the *stack* field is modified, then the value written must a literal.

```
(defthm stack-contains-literalsp-fwrite-literal
  (implies                                  ; if
   (and (literalp l)                        ; l is literal,
        (stack-contains-literalsp i st)     ; stack has only lits,
        (field-offsetp o *stack*            ; and o is offset into
                       *s* st))             ; stack, then
   (stack-contains-literalsp                ; stack has only lits
    i                                       ; after writing l to
    (fwrite *stack* o l *s* st))))          ; stack field
```

These two theorems are sufficient to show that the stack contains only literals after assigning a literal to true. There are many other combinations of invariants and operations on the state, all of which are more complicated than this example. The most difficult invariants to maintain are the correspondence between lookup table and stack and vice versa because these invariants rely on all of the writes as a whole: it is harder to case split on the field that is being written.

Once the assignment-stp predicate has been established for each basic operation, one can begin the process of relating the new array-based model for assignments to the old list-based model.

Chapter 8

Equivalence

Instead of proving the soundness of an implementation that uses the new `assignment-st` structure from Chapter 7, it is possible to prove that the new implementation is equivalent to the algorithm presented in Chapter 5. This is achieved by a process called bisimulation. For each function, one can show that the result of the new state-based function is the same as the result of the original list-based function. This relies on the ability to *project* the `assignment-st` objects to assignments as lists. Executing a function in the new model and then projecting the result should be the same as projecting the initial assignment and executing the corresponding function in the old model. Figure 8.1 depicts these relationships.

First, a projection function is defined that takes an `assignment-st` structure as input and outputs a list-based assignment satisfying the `assignmentp` predicate. This is accomplished by reading the stack from high to low and constructing a list from each element. The non-recursive function `project` accepts a state as input and returns a list; this function uses an auxilliary recursive function called `project1`. Note that `project1` is generalized to work on any field.

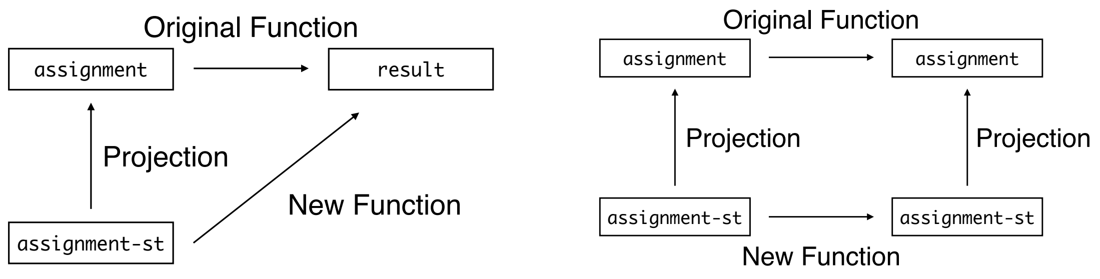

Figure 8.1: Two forms of bisimulation used in the equivalence proof relating an implementation using state-based assignments to an implementation using list-based assignments. Some functions return a ternary value, literal, clause, formula, etc. The equivalence for these functions is represented by the diagram on the left. Other functions return an assignment and the equivalence is represented by the diagram on the right.

```
(defun project1 (i f s st)        ; (i)ndex, (f)ield, (s)tart,
                                  ; (st)ate
  (declare (xargs :guard (and (farrayp s st)
                              (fieldp f s st)
                              (or (field-offsetp i f s st)
                                  (equal i -1)))
                  :stobjs (st)
                  :measure (nfix (1+ i))))
  ;; the following (m)ust (b)e (t)rue to contine
  (if (not (mbt (and (farrayp s st)
                     (fieldp f s st)
                     (or (field-offsetp i f s st)
                         (equal i -1)))))
      nil
    (if (equal i -1)              ; if i = -1, then f is
        nil                       ; empty, return empty list
      (cons (fread f i s st)      ; else, return value at i
            (project1 (1- i)      ; in f cons onto the rest
                      f s st)))))  ; of the projection

(defun project (st)                  ; (st)ate
  (declare (xargs :guard (assignment-stp st)
                  :stobjs (st)))
  (project1                          ; recursively project
   (1- (fread *stack-end* 0 *s* st)) ; starting at end of stack
   *stack* *s* st))                  ; in stack field
```

The first part of the equivalence proof for the RAT validation utility involves proving that a state satisfying the `assignment-stp` predicate implies that the projection satisfies the `assignmentp` predicate. In other words, the `project` function should return a valid list-based assignment.

```
(defthm assignmentp-project
  (implies                       ; if
   (assignment-stp st)           ; st is valid assignment state,
   (assignmentp (project st))))  ; then project returns assignment
```

Theorems are then established showing the correspondence for each operation on the state: `assignedp`, `assign-lit`, `unassign-one`, and `unassign-all`. For example, projecting after calling `assign-lit` is equal to `cons`-ing a literal after projection.

```
(defthm assign-lit-cons-project
  (implies                            ; if
   (and (assignment-stp st)           ; valid assignment state st,
        (literalp l)                  ; literal l,
        (lit-in-rangep l st))         ; and l fits in assignment,
   (equal (project                    ; then, projecting after
           (assign-lit l st))         ; assigning l in state is
          (cons l                     ; the same as adding l
                (project st)))) ...)  ; after projecting to a list
```

Next, each RAT validation function defined in Chapters 4 and 5 is defined for the `assignment-st` implementation. These function names are similar except they have the `-st` suffix. Some functions are almost identical and the only change is that they call state-based operations instead of list-based operations. Consider the function `evaluate-literal` function from Section 4.2. This function accepts a literal and an assignment and returns a ternary value. In the state-based implementation, the function `evaluate-literal-st` accepts a literal and state and returns a ternary value.

```
(defun evaluate-literal-st (l st)      ; (l)iteral, (st)ate
  (declare (xargs :guard (and (assignment-stp st)
                              (literalp l)
                              (lit-in-rangep l st))
                  :stobjs (st)))
  (cond
   ((assignedp l st) (true))           ; if l assigned, true
   ((assignedp (negate l) st) (false)) ; if l̄ assigned, false
   (t (undef))))                       ; else, undef
```

Two of the more interesting function definitions in the state-based model are `unit-propagation-st` and `RUPp-st`. In the case of `unit-propagation-st`, it is much easier to provide a measure based on the projection rather than define the measure based on a property of the state. Recall from Section 5.4 that the list-based `unit-propagation` has a custom measure that describes a property that always decreases on each recursive call: the number of clauses that evaluate to `undef`. The same property is true of the state-based assignment and the *projection* of the state-based assignment.

```
(defun unit-propagation-st (f st)        ; (f)ormula, (st)ate
  (declare (xargs :guard (and (assignment-stp st)
                              (formulap f)
                              (formula-in-rangep f st))
                  :verify-guards nil     ; guard proof delayed
                  :stobjs (st)
                  :measure (num-undef    ; note measure in
                            f            ; terms of projection
                            (project st))))
  ;; the following (m)ust (b)e (t)rue to contine
  (if (not (mbt (and (assignment-stp st)
                     (formulap f)
                     (formula-in-rangep f st))))
      st
     (mv-let                             ; let results of
      (ul uc)                            ; finding unit clause be
      (find-unit-clause-st f st)         ; bound to ul and uc
      (declare (ignorable uc))           ; not using uc below
      (if (not ul)                       ; if no unit literal
          st                             ; found, return state
        (let ((st (assign-lit ul st)))   ; else, assign ul
          (unit-propagation-st f st)))))) ; and recur with new st
```

In the case of `RUPp-st`, the assignment needs to be cleared by calling `unassign-all` and then recursive assigning the negation of each literal in a clause via a function called `clause-to-assignment`, which is not defined here. This is perhaps the most difficult function to reason about in the state-based implementation because it is so different from the definition in the list-based implementation.

```
(defun RUPp-st (f c st)                   ; (f)ormula, (c)lause
                                          ; (st)ate
  (declare (xargs :guard (and (assignment-stp st)
                              (formulap f)
                              (formula-in-rangep f st)
                              (clausep c)
                              (clause-in-rangep c st))
                  :stobjs (st)))
  (let* ((st (unassign-all                ; unassign all lits
               (fread *stack-end* 0 *s* st)  ; starting at top of
               st))                       ; stack
         (st (clause-to-assignment c st))   ; convert clause to
         (st (unit-propagation-st f st)))   ; assignment and prop
    (mv (falsep (evaluate-formula-st f st)) ; return if eval of f
        st)))                             ; is false and state
```

The list-based version of this function is defined in Section 5.2.

Once each function is defined, an equivalence proof is established that relates state-based functions to list-based functions. Many of these theorems are easy to prove because of the equivalence of assignment operations discussed above. For example, the `evaluate-literal-st` and `evaluate-literal` functions share a nearly identical body. The theorem relating these two functions is stated as:

```
(defthm evaluate-literal-equiv
  (implies                       ; if
   (and (assignment-stp st)      ; valid assignment state st,
        (literalp l)             ; literal l,
        (lit-in-rangep l st))    ; and l fits in assignment, then
   (equal (evaluate-literal-st   ; evaluating l in assignment state
           l st)                 ; is equal to
          (evaluate-literal      ; evaluating literal after
           l (project st)))))    ; projecting state to list
```

Some of the theorems are much more difficult. Consider the equivalence theorem for unit propagation.

```
(defthm unit-propagation-equiv
  (implies                                ; if
   (and (assignment-stp st)               ; valid assignment state,
        (formulap f)                      ; formula f, and
        (formula-in-rangep f st))         ; all l ∈ f fit in st,
   (equal (project                        ; then projecting state
           (unit-propagation-st f st))    ; after unit prop
          (unit-propagation               ; is equal to unit prop
           f (project st)))))             ; after projecting state
```

This theorem is the reason a stack is necessary in the array-based assignment. By reading the stack to a list, one can show that an array-based assignment and list-based assignment are *equal* instead of *set equal*. An initial implementation of state-based assignments only used a lookup table. The projection function in this implemenation would scan the lookup table (in numerical order) and construct a list from all literals with a value `1`. This produced an assignment that was set equal to the list-based assignment. The former was ordered by literal value and the latter was ordered by the time each literal was assigned to true. This made it extremely difficult to prove the equivalence for the `unit-propagation` routine, which incrementally assigns literals. A (partial) proof for the equivalence of `unit-propagation` was constructed, but it relied heavily on quantification.

Finally, it is possible to establish a version of the `main-theorem` from Section 6.1 about the new RAT validation implementation.

```
(defthm main-theorem-st
  (implies                              ; if
   (and (assignment-stp st)             ; valid assignment state,
        (formulap f)                    ; formula f,
        (formula-in-rangep f st)        ; all l ∈ f fit in st,
        (clause-listp cl)               ; clause list cl,
        (clause-list-in-rangep cl st)   ; all l ∈ cl fit in st,
        (mv-nth 0 (refutationp-st       ; and cl is refutation for
                   cl f st)))           ; f using state, then
   (not (exists-solution f))))          ; f is unsatisfiable
```

This theorem states that a refutation that is checked by the state-based implementation implies that the formula is unsatisfiable.

Chapter 9

Future Work

The DRAT proof format is powerful enough to express all contemporary SAT solving and preprocessing techniques, but this does not imply that all of these techniques are naturally expressed in the format. Some techniques, such as Gaussian elminiation and handling of XOR constraints, may require extra thought. Ultimately, it is up to solver authors to determine how to best express the reasoning behind their techniques in a format that is checkable.

DRAT-trim has already seen success outside of its validation capabilities. TraceCheck$^+$ dependency graphs, one possible output of DRAT-trim, are already being used as input to minimal unsatisfiable subset (MUS) [62, 7] and to compute Craig interpolants [76, 36]. There are most likely other uses for DRAT-trim that the satisfiability community will discover.

There is still much to do with respect to the mechanically-verified implementation. While the representation of assignments is completed, there are other features that must be implemented and shown equivalent in order to approach the efficiency of the C-based implementation of DRAT-trim:

1. **Clauses integrated into array-based data structure.** In C, clauses are stored in an array, and access to each clause can be obtained by an index into

the array. On its own, this does not provide much benefit, but it is necessary when representing dynamic sets of clauses like watched-literal data structures and more efficient RAT checks. The current array-based implementation for assignments should be extended to contain another large field for the clauses in the formula and refutation. A proof index field will also need to be added to record the separation between the (extended) formula and the (remaining) proof. Operations on formulas and clauses will need to be expressed in terms of the new fields and these operations should be shown equivalent to list-based counterparts. No modification to the original proof of correctness is expected.

2. **Base unit propagation.** Many formulas contain unit clauses, and a significant performance improvement can be obtained by recording these unit clauses and their implications in a "base assignment". Right now, the mechanically-verified algorithm (re)discovers all of the implications of the original formula every time it checks redundancy. To avoid this, before any redundancy checks are performed, unit propagation should be applied, resulting in a base assignment that is implied by the original formula. This base assignment will be a subset of all other assignments during validation and should be used as a base for future redundancy checks. To implement this, the array-based data structure needs an additional pointer into the assignment stack which records the base assignment. After each redundancy check, instead of unassigning all literals, the function should only unassign back to the base assignment pointer. This will require not only a change of the data structure, but also a change in the algorithm. A proof will need to be constructed that shows that a base

assignment is a subset of any assignment produced during redundancy checks.

3. **Watched literal data structure.** Watched literals greatly increase the performance of unit propagation. This data structure and associated algorithmic improvements still need to be integrated into the ACL2 implementation. There are two common ways of representing watched literal lists. The first is to maintain a separate list clause indices for each literal that point to clauses where the literal is watched. The second is to incorporate these lists into the clause database by including two indices at the beginning of the clause. While the first option may be easier to reason about simply because it is separated from the clause database, it is harder to implement without dynamic allocation. The total amount of space for all watched literal lists is constant, but individual lists will shrink and grow during computation. Thus, the second option is easier to implement but requires additional invariants over the clause database. Regardless of the data structure used, the clauses of the formula will be rearranged during validation as certain literals become "watched". From a proof perspective, a formula at the end of validation could be quite different to the formula before validation. Permutation of the clauses in the formula must be shown irrelevant to evaluation of clauses and formulas.

4. **Deletion information.** Deletion information still needs to be included in the mechanically-verified proof checker. This is more of a problem from an implementation perspective than a proof perspective. Deletion instructions must be matched with clauses in the formula. In DRAT-trim, this matching is performed by hashing the clauses to obtain "timestamps" that describe when

a certain clause is active. It is not immediately clear what method should be used to obtain these timestamps, but it may be necessary to perform this kind of operation outside of the verification process. An algorithm for an earlier proof format, called IORUP, included temporal information of clauses and was mechanically verified using ACL2. In this experiment, it was not difficult to show that it was sound to ignore clauses during the validation process. In the main theorem that states validation implies unsatisfiability, removal of clauses during validation can only falsify the hypothesis, making the theorem trivially true.

5. **Reading directly into STOBJ.** Currently, formulas and refutations are read and parsed into a list structure. The entire list structure is then checked by a predicate to determine if the formula is valid. It should be possible to read the formula and refutation directly into the array-based implementation and check any necessary predicates during parsing.

The `farray` data structure is (at the time of this publication) under active development. Some of the areas for improvement are:

1. The length of each field is constant once an `farray` is defined. An `fresize` function can be defined that would resize a field by modifying the field table and shifting data to new positions. This work has partially been accomplished. The field table adjustment function has been defined and proven to maintain the `farray` invariant.

2. The location of the `farray` in is constant. An `farray-move` could be defined that would shift the entire `farray` to a new start location.

3. All `fread` and `fwrite` operations require an extra `memi` access and some arithmetic. A `relative-to-absolute` function (and `absolute-to-relative` function) can be defined that would convert a `field-offsetp` to an absolute index in `mem` (and vice versa). Accesses and updates to `mem` with an absolute address should preserve the `farray` structure. Users could "drop" to absolute indices for important functions to increase performance.

4. The `farray` structure is limited to the STOBJ name `st` and the field name `mem`. One should be able to define an `farray` in such a way that they can be used with other STOBJ names.

Chapter 10

Conclusions

As a direct result of this work, all state-of-the-art satisfiability solvers can be validated efficiently and the core part of the validation process has been mechanically verified. These contributions fundamentally change the relationship between solver developers and solver users in that users have a reasonable method for checking important SAT results.

The DRAT proof format is the basis for such a radical change. With the addition of expressibility and clause management to existing clausal proof formats, the DRAT format allows solver developers to easily emit all steps a solver or preprocesser takes including the deletion of clauses from a working database. This format is accessible enough that it was used by all top-tier solvers in the UNSAT track of the 2014 SAT Competition. The widespread adoption of DRAT in the SAT community is remarkable.

The DRAT format would be uninteresting if it could not be efficiently evaluated. The DRAT-trim validation and trimming utility is evidence that such proofs can be validated in a similar time to solving. All DRAT proofs emitted by solvers in the 2014 SAT competition were validated within the competition time constraints using DRAT-trim. Furthermore, DRAT-trim optionally produces trimmed formulas, optimized DRAT proofs, and dependency graphs. These outputs are already being used as input to MUS extraction or interpolation utilities.

The key advancement of the DRAT format is the addition of the RAT check to traditional clausal proofs. Validation of proofs containing RAT clauses was mechanically verified using the ACL2 theorem proving system. This involves specifying unsatisfiability for a formula, developing an algorithm to validate RAT proofs, and then formally proving that validation implies unsatisfiability. This is evidence that validated RAT proofs guarantee unsatisfiability of the original formula.

A new ACL2 data structure and associated theory enable users to model C-style memory management by introducing multiple fields in a single array. This is an good first step towards applying mechanical verification to fast formal models and providing users with an efficient and convenient theory. This method of modeling is already being adopted by other ACL2 users who want to mechanically verify code without sacrificing performance.

Another iteration of a mechanically-verified RAT checker demonstrates that an efficient implementation is possible within ACL2. This was accomplished by changing the data structure of the only mutable structure in the original algorithm, the assignment. Reads and writes were reduced from linear time operations to constant time operations by means of arrays in ACL2 STOBJs. The array-based implementation is proven equivalent to the original algorithm and, thus, benefits from the previous proof of correctness. Although this implementation does not contain all of the features and techniques of DRAT-trim, it is a significant step towards efficient, mechanically-verified validation of SAT solvers.

In a more general sense, this work points the way towards mechanical verification of C-style code. High-level, abstract algorithms (such as the one shown in Chapter 5) can be proven equivalent to a state-based, efficient model (an example of which is given in Chapter 8). The work presented here documents this process for unsatisfiability proof validation but can be applied to other programs as well. In this respect, our approach provides a starting point for transforming and verifying low-level code.

Bibliography

[1] Michaël Armand, Benjamin Grégoire, Arnaud Spiwack, and Laurent Théry. Extending Coq with imperative features and its application to SAT verification. In *Interactive Theorem Proving (ITP)*, volume 6172 of *LNCS*, pages 83–98. Springer, 2010.

[2] Mickaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Wener. Verifying SAT and SMT in Coq for a fully automated decision procedure. In *International Workshop on Proof-Search in Axiomatic Theories and Type Theories (PSATTT)*, 2011.

[3] Gilles Audemard, George Katsirelos, and Laurent Simon. A restriction of extended resolution for clause learning SAT solvers. In Maria Fox and David Poole, editors, *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI)*. AAAI Press, 2010.

[4] Ken E. Batcher. Sorting networks and their applications. In *Proceedings of the Spring Joint Computer Conference*, AFIPS 1968 (Spring), pages 307–314. ACM, 1968.

[5] Jason Baumgartner, Hari Mony, Viresh Paruthi, Robert Kanzelman, and Geert Janssen. Scalable sequential equivalence checking across arbitrary design trans-

formations. In *International Conference on Computer Design (ICCD)*, pages 259–266. IEEE, 2007.

[6] Paul Beame, Henry Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research (JAIR)*, 22:319–351, 2004.

[7] Anton Belov, Marijn J. H. Heule, and João P. Marques-Silva. MUS extraction using clausal proofs. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 8561 of *LNCS*, pages 48–57. Springer, 2014.

[8] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.

[9] Armin Biere. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 4:75–97, 2008.

[10] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Masahiro Fujita, and Yunshan Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Design Automation Conference (DAC)*, pages 317–320. ACM, 1999.

[11] Armin Biere, Marijn J. H. Heule, and Hans van Maaren. *Handbook of Satisfiability*, volume 185. IOS Press, 2009.

[12] Niklas Sörensson Armin Biere. Minimizing learned clauses. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 5584 of *LNCS*, pages 237–243. Springer, 2009.

[13] Sascha Böhme and Tjark Weber. Fast LCF-style proof reconstruction for z3. In *Interactive Theorem Proving (ITP)*, volume 6172 of *LNCS*, pages 179–194. Springer, 2010.

[14] R. C. Bose and R. J. Nelson. A sorting problem. *Journal of the ACM (JACM)*, 9(2):282–296, April 1962.

[15] Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of SAT and QBF solvers. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 6175 of *LNCS*, pages 44–57. Springer, 2010.

[16] Yibin Chen, Sean Safarpour, João P. Marques-Silva, and Andreas Veneris. Automated design debugging with maximum satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(11):1804–1817, 2010.

[17] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

[18] Michael Codish, Luís Cruz-Filipe, Michael Frank, and Peter Schneider-Kamp. Twenty-five comparators is optimal when sorting nine inputs (and twenty-nine for ten). In *International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 186–193. IEEE, 2014.

[19] John Conway. The game of life. *Scientific American*, 223(4):4, 1970.

[20] Stephen A. Cook. A short proof of the pigeon hole principle using extended resolution. *SIGACT News*, 8(4):28–32, 1976.

[21] Stephen A. Cook and Robert A. Reckhow. The relative efficiency of propositional proof systems. *The Journal of Symbolic Logic*, 44(01):36–50, 1979.

[22] Ashish Darbari, Bernd Fischer, and João P. Marques-Silva. Industrial-strength certified SAT solving through verified SAT proof checking. In *Theoretical Aspects of Computing (ICTAC)*, volume 6255 of *LNCS*, pages 260–274. Springer, 2010.

[23] Jared Davis. *A Self-Verifying Theorem Prover*. PhD thesis, The University of Texas at Austin, December 2009.

[24] Jared Davis and Sol Swords. Bit-blasting ACL2 theorems. In *International Workshop on the ACL2 Theorem Prover and its Applications (ACL2)*, volume 70 of *EPTCS*, pages 84–102, 2011.

[25] Jared Davis and Sol Swords. Verified AIG algorithms in ACL2. In *International Workshop on the ACL2 Theorem Prover and its Applications (ACL2)*, volume 114 of *EPTCS*, pages 95–110, 2013.

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

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

[28] Leonardo M. de Moura and Nikolaj Bjørner. Proofs and refutations, and Z3. In *Logic for Programming, Artificial Intelligence and Reasoning (LPAR) Workshops*, volume 418 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[29] Michael R. Dransfield, Victor W. Marek, and Mirosław Truszczyński. Satisfiability and computing van der Waerden numbers. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 1–13. Springer, 2004.

[30] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 3569 of *LNCS*, pages 61–75. Springer, 2005.

[31] Paul Erdős. Some unsolved problems. *The Michigan Mathematical Journal*, 4(3):291–300, 1957.

[32] Hiroshi Fujita, Miyuki Koshimura, and Ryuzo Hasegawa. SCSat: a soft constraint guided SAT solver. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 7962 of *LNCS*, pages 415–421. Springer, 2013.

[33] Evguenii I. Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 10886–10891. IEEE, 2003.

[34] Evguenii I. Goldberg, Mukul R. Prasad, and Robert K. Brayton. Using SAT for combinational equivalence checking. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 114–121. IEEE, 2001.

[35] Ronald L. Graham, Bruce L. Rothschild, and Joel H. Spencer. *Ramsey theory*. John Wiley and Sons, New York, second edition, 1990.

[36] Arie Gurfinkel and Yakir Vizel. DRUPing for interpolants. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 99–106. FMCAD Inc, 2014.

[37] Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985.

[38] Christiaan Hartman, Marijn J. H. Heule, Kees Kwekkeboom, and Alain Noels. Symmetry in gardens of eden. *Electronic Journal of Combinatorics*, 20(3):16, 2013.

[39] Marijn J. H. Heule, Warren A. Hunt, Jr., and Nathan Wetzler. Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 181–188. IEEE, 2013.

[40] Marijn J. H. Heule, Warren A. Hunt, Jr., and Nathan Wetzler. Verifying refutations with extended resolution. In *International Conference on Automated Deduction (CADE)*, volume 7898 of *LNAI*, pages 345–359. Springer, 2013.

[41] Marijn J. H. Heule, Warren A. Hunt, Jr., and Nathan Wetzler. Bridging the gap between easy generation and efficient verification of unsatisfiability proofs. *Software Testing, Verification, and Reliability (STVR)*, 24(8):593–607, 2014.

[42] Marijn J. H. Heule, Warren A. Hunt, Jr., and Nathan Wetzler. Expressing symmetry breaking in DRAT proofs. In *International Conference on Automated Deduction (CADE)*, 2015. Under submission.

[43] Marijn J. H. Heule, Matti Järvisalo, and Armin Biere. Clause elimination procedures for CNF formulas. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 6937 of *LNCS*, pages 357–371. Springer, 2010.

[44] Marijn J. H. Heule and Hans van Maaren. *Look-Ahead Based SAT Solvers*, chapter 5, pages 155–184. IOS Press, 2009.

[45] Matti Järvisalo, Armin Biere, and Marijn J. H. Heule. Simulating circuit-level simplifications on CNF. *Journal of Automated Reasoning*, 49(4):583–619, 2012.

[46] Matti Järvisalo, Marijn J. H. Heule, and Armin Biere. Inprocessing rules. In *International Joint Conference on Automated Reasoning (IJCAR)*, volume 7364 of *LNCS*, pages 355–370. Springer, 2012.

[47] Daher Kaiss, Marcelo Skaba, Ziyad Hanna, and Zurab Khasidashvili. Industrial strength SAT-based alignability algorithm for hardware equivalence verification. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 20–26. IEEE, 2007.

[48] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, Boston, June 2000.

[49] Chantal Keller. Extended resolution as certificates for propositional logic. In *International Workshop on Proof Exchange for Theorem Proving (PxTP)*, volume 14 of *EPiC Series*, pages 96–109. EasyChair, 2013.

[50] Donald E. Knuth. *The art of computer programming: sorting and searching*, volume 3. Pearson Education, 1998.

[51] Boris Konev and Alexei Lisitsa. A SAT attack on the Erdős Discrepancy Conjecture. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 8561 of *LNCS*, pages 219–226. Springer, 2014.

[52] Michal Kouril and Jerome L. Paul. The van der Waerden number W(2,6) is 1132. *Experimental Mathematics*, 17(1):53–61, 2008.

[53] Oliver Kullmann. On a generalization of extended resolution. *Discrete Applied Mathematics*, 96-97:149–176, 1999.

[54] Stéphane Lescuyer and Sylvain Conchon. A reflexive formalization of a SAT solver in Coq. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, 2008.

[55] Norbert Manthey, Marijn J. H. Heule, and Armin Biere. Automated reencoding of boolean formulas. In *Proceedings of Haifa Verification Conference (HVC)*, volume 6397 of *LNCS*, pages 102–117. Springer, 2012.

[56] Filip Marić. Formalization and implementation of modern SAT solvers. *Journal of Automated Reasoning*, 43(1):81–119, 2009.

[57] Filip Marić. Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. *Theoretical Computer Science*, 411(50):4333–4356, 2010.

[58] João P. Marques-Silva, Ines Lynce, and Sharad Malik. *Conflict-Driven Clause Learning SAT Solvers*, chapter 4, pages 131–153. IOS Press, 2009.

[59] Alan Mishchenko, Satrajit Chatterjee, Robert Brayton, and Niklas Een. Improvements to combinational equivalence checking. In *International Conference on Computer-Aided Design (ICCAD)*, pages 836–843. IEEE, 2006.

[60] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference (DAC)*, pages 530–535. ACM, 2001.

[61] Magnus O. Myreen and Jared Davis. The reflective Milawa theorem prover is sound. In *Interactive Theorem Proving (ITP)*, volume 8558 of *LNCS*, pages 421–436. Springer, 2014.

[62] Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Efficient MUS extraction with resolution. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 197–200. IEEE, 2013.

[63] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[64] Duckki Oe and Aaron Stump. Combining a logical framework with an RUP checker for SMT proofs. In *Satisfiability Modulo Theories (SMT)*, page 40, 2011.

[65] Duckki Oe, Aaron Stump, Corey Oliver, and Kevin Clancy. versat: a verified modern SAT solver. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 363–378. Springer, 2012.

[66] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In *International Conference on Automated Deduction (CADE)*, volume 607 of *LNCS*, pages 748–752. Springer, 1992.

[67] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.

[68] Vadim Ryvchin and Ofer Strichman. Faster extraction of high-level minimal unsatisfiable cores. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 6695 of *LNCS*, pages 174–187. Springer, 2011.

[69] Natarajan Shankar and Marc Vaucher. The mechanical verification of a DPLL-based satisfiability solver. *Electronic Notes in Theoretical Computer Science*, 269:3–17, 2011.

[70] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 127–144. Springer, 2000.

[71] Laurent Simon. Post mortem analysis of SAT solver proofs. In *Pragmatics of SAT (POS) Workshop*, volume 27 of *EPiC Series*, pages 26–40. EasyChair, 2014.

[72] Anna Slobodová, Jared Davis, Sol Swords, and Warren A. Hunt, Jr. A flexible formal verification framework for industrial scale validation. In *International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 89–97. IEEE, 2011.

[73] Aaron Stump, Morgan Deters, Adam Petcher, Todd Schiller, and Timothy Simpson. Verified programming in Guru. In *Workshop on Programming Languages Meets Program Verification (PLPV)*, pages 49–58. ACM, 2009.

[74] Grigori S. Tseitin. On the complexity of derivation in propositional calculus. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning 2*, pages 466–483. Springer-Verlag, 1983.

[75] Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *International Symposium on Artificial Intelligence and Mathematics (ISAIM)*, 2008.

[76] Yakir Vizel, Vadim Ryvchin, and Alexander Nadel. Efficient generation of small interpolants in CNF. In *Computer Aided Verification (CAV)*, volume 8044 of *LNCS*, pages 330–346. Springer, 2013.

[77] Tjark Weber. Efficiently checking propositional resolution proofs in Isabelle/HOL. In *International Workshop on the Implementation of Logics (IWIL)*, volume 212, pages 44–62, 2006.

[78] Tjark Weber and Hasan Amjad. Efficiently checking propositional refutations in HOL theorem provers. *Journal of Applied Logic*, 7(1):26–40, 2009.

[79] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt, Jr. Mechanical verification of SAT refutations with extended resolution. In *Interactive Theorem Proving (ITP)*, volume 7998 of *LNCS*, pages 229–244. Springer, 2013.

[80] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt, Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 8561 of *LNCS*, pages 422–429. Springer, 2014.

[81] Lintao Zhang and Sharad Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 880–885. IEEE, 2003.