Mechanical Verification of SAT Refutations
with Extended Resolution

Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt Jr. [*]

The University of Texas at Austin

Abstract. We present a mechanically-verified proof checker developed with the ACL2 theorem-proving system that is general enough to support the growing variety of increasingly complex satisfiability (SAT) solver techniques, including those based on extended resolution. A common approach to assure the correctness of SAT solvers is to emit a proof of unsatisfiability when no solution is reported to exist. Contemporary proof checkers only check logical equivalence using resolution-style inference. However, some state-of-the-art, conflict-driven clause-learning SAT solvers use preprocessing, inprocessing, and learning techniques, that cannot be checked solely by resolution-style inference. We have developed a mechanically-verified proof checker that assures refutation clauses preserve satisfiability. We believe our approach is sufficiently expressive to validate all known SAT-solver techniques.

1 Introduction

Satisfiability (SAT) solvers are becoming commonplace for a variety of applications, including model checking [1], equivalence checking, hardware verification, software verification, and debugging. These tools are often used not only to find a solution for a Boolean formula, but also to make the claim that no solution exists. 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, we want to 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 to maximize efficiency. Errors may be introduced at a conceptual level as well as a implementation level. Formal verification, then, is a reasonable approach to detect errors or to assure that 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. The benefit of such a direct approach is that the solver would only need to run once for a given input. There are many problems with this approach, however. SAT solvers are constantly evolving, and each new implementation would require a new proof. Furthermore, it is hard to balance verification requirements with efficiency. Lescuyer and Conchon [2] formalized and verified the basic Davis-Putnam-Logemann-Loveland (DPLL) [3,4] algorithm

using Coq [5]. Shankar and Vaucher [6] verified a DPLL solver using PVS. Marić [7,8] verified pseudocode fragments of a conflict-driven clause-learning (CDCL) [9] solver in 2009 and verified a CDCL-style solver using Isabelle/HOL [10] in 2010. Oe et al. [11] provided an verified CDCL-style solver in Guru. Several key techniques, such as clause minimization during conflict analysis, have yet to be mechanically verified.

Another approach is to validate the output of a SAT solver. 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 program called a proof checker. A refutation is a verified proof trace containing the empty clause. Ideally, a proof trace should be compact, easy to obtain, efficient to verify, and should facilitate a simple checker implementation. Moreover, we "only" need to formally verify the proof checker. By focusing verification efforts on a proof checker, we gain assurance while avoiding the need to verify a variety of solvers with differing implementations.

Proof traces have traditionally established redundancy in the form of resolution chains [12,13,14]. In resolution-style proofs, clauses are iteratively added to a formula provided that they can be derived from a sequence of applications of the resolution rule. Resolution proof traces are simple to express and can be efficiently validated, but they tend to be enormous and difficult to obtain from a solver. Weber [15,16] demonstrated the first mechanically-verified resolution-based proof checker using Isabelle/HOL. Darbari et al. [17] verified a resolution-based proof checker in Coq which is able to execute outside of the theorem-prover environment. Armand et al. [18] extended a SAT resolution-based proof checker to include SMT proofs using Coq.

Alternatively, one can use unit propagation, one of the basic SAT simplification techniques, to check proof traces. Each proof clause is shown to be redundant by adding the complement of the clause as unit clauses, performing unit propagation with respect to the conjunction of the original formula and all verified proof clauses so far, and then checking for a conflict. This process is known as reverse unit propagation (RUP) [13,19]. RUP proofs are compact and easy to obtain; however, RUP checkers are somewhat inefficient and more complicated than their resolution-based counterparts. Oe and Stump [20] implemented a non-verified RUP proof checker in C++ and proposed a verified RUP proof checker in Guru.

However, both resolution and 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, and some of these techniques cannot be expressed by resolution-style inference such as bounded-variable addition [21], blocked-clause addition [22], and extended learning [23]. These techniques can be expressed, however, by extended resolution (ER) [24] or a generalization of ER [22]. Järvisalo et al. [25] demonstrated a hierarchy of redundancy properties, the most expressive of which is Resolution Asymmetric Tautology (RAT), which is a generalization of RUP. All 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 RUP (or resolution) is that RAT checks satisfiability equivalence instead of logical equivalence.

In [26], we proposed a new proof format based on the RAT redundancy property and described a fast implementation of a proof checker for this format written in C. In this paper, we present a mechanically-verified SAT proof checker using ACL2 [27] based on the RAT redundancy property. This includes a mechanical proof of the redundancy (via satisfiability equivalence) of RAT clauses. Our implemented proof checker is the most expressive proof checker to date and it is mechanically verified.

In Section 2 we will introduce ACL2 and formalize key SAT concepts including unit propagation and resolution. We will also discuss a redundancy hierarchy and provide our implementation of RAT and our RAT proof checker. We will give a specification for our RAT proof checker in Section 3, and present our mechanical proof of correctness in Section 4. Finally, we conclude in Section 5.

2 Formalization

2.1 ACL2

We used the ACL2 system [27] to develop our formalization, specification, and proof. ACL2 is a freely-available system that provides a theorem prover and a programming language, both of which are based on a first-order logic of recursive functions. The logic is compatible with Common Lisp—indeed, "ACL2" is an acronym that might be written as "ACL^2" and stands for "A Computational Logic for Applicative Common Lisp"— and an executable image can be built on a number of Common Lisp implementations. ACL2 provides efficient execution by way of Common Lisp compilers.

The initial theory for ACL2 contains axioms for primitive functions such as `cons` (the constructor for an ordered pair), `car` (the head of a list or first component of a pair), and `cdr` (the tail of a list or second component of a pair). It also contains axioms for Common Lisp functions, such as `member`, and it introduces axioms for user-supplied definitions. ACL2 provides a top-level read-eval-print loop. Arbitrary ACL2 expressions may be submitted for evaluation. Of special interest are events, including definitions and theorems; these modify the the theorem prover's logical database for subsequent proof and evaluation. Function definitions are typically expressed using the `defun` event and theorems using the `defthm` event. We call an ACL2 function a *predicate* if it returns a Boolean value.

As is the case for Lisp, the syntax of ACL2 is generally case-insensitive and is based on prefix notation: (`function argument1 ... argumentk`). For example, the term denoting the sum of x and y is (`+ x y`). A semicolon ";" starts a comment. ACL2 also supports the functions `let` and and `let*` for parallel and sequential bindings, respectively. Functions may return mulitple values using the constructor `mv` which stands for "multiple value". Elements of `mv` may be accessed with the function `mv-nth` which accesses the nth value of an `mv`. The function `mv-let` takes three arguments: a list of bindings, a function that returns an `mv` (with the same number of values as the bindings), and a body. For example, suppose that a function `f` returns two values and we wish to compute their sum. We can compute this with the following term:

```
(mv-let (x y)
        (f ...)
        (+ x y))
```

ACL2 does not have native support for quantification in the logic; however, ACL2 allows a user to define *Skolemized functions* using the `defun-sk` event. This event introduces a witness function that will return a witness object if such an object exists. For example, if we wanted to express the mathematical statement, "there exists an x such that $x < y$", we could do so with the event:

```
(defun-sk exists-x-<-y (y) (exists x (< x y)))
```

This event defines a non-executable, one-argument function `exists-x-<-y-witness` that will return an `x` if one exists. The non-executable function `exists-x-<-y` returns true if `exists-x-<-y-witness` finds such an object.

Links to papers that apply ACL2, as well as detailed hypertext documentation and installation instructions, may be found on the ACL2 home page.[1]

2.2   Satisfiability Basics

We will now begin introducing some SAT concepts. We will forego providing the traditional SAT notation and will instead use ACL2 notation. In this way, we can define key SAT terminology while describing our formalization.

We model Boolean variables with positive integers. For a Boolean variable `v`, there are two *literals*, the positive literal `l` and the negative literal computed by `(negate l)`. We represent positive and negative literals as positive and negative integers, and we recognize them with the predicate `literalp`. A *clause* is a finite disjunction of literals, and a clause is a *tautology* if it contains the conflicting literals `l` and `(negate l)` for some `l`. We introduce the predicate `no-conflicting-literalsp` to recognize non-tautological lists, and we define the predicate `clausep` to recognize non-tautological ACL2 lists of unique literals.[2] A *conjunctive normal form* (CNF) *formula*, recognized by predicate `formulap`, is a finite conjunction of clauses which we represent as an ACL2 list of non-tautological clauses. We do not require clauses to be unique within a formula. In the rest of this paper, we will assume all formulas to be in CNF. The set of literals occurring in a formula `f` is computed by the function `all-literals`.

A truth *assignment* for a formula `f` is a partial function that maps literals `l` in `(all-literals f)` to Boolean values. We model an assignment, with predicate `assignmentp`, as a ACL2 list of unique non-conflicting literals (note that the implementations of `assignmentp` and a `clausep` are the same). In our ACL2 representation, we define special values `true`, `false`, and `undef` with corresponding predicates `truep`, `falsep`, `undefp`. The evaluation of literal `l` with respect to assignment `ta` is computed by the function `evaluate-literal` which returns `true` if `(member l ta)`, `false` if `(member (negate l) ta)`, and `undef` otherwise. The evaluation of a clause `c` with respect to assignment `ta` is computed by the function `evaluate-clause` which returns `true` if `(evaluate-literal l ta)` is true for some literal `l` in `c`, `false` if `(evaluate-literal l ta)` is `false` for all literals `l` in `c`, and `undef` otherwise. The evaluation of a formula `f` with respect to assignment `ta` is computed by the function `evaluate-formula` which returns `true` if `(evaluate-clause c ta)` is true for all clauses `c` in `f`, `false` if `(evaluate-clause c ta)` is `false` for some clause `c` in `f`, and `undef` otherwise.

---

[1] `www.cs.utexas.edu/users/moore/acl2/`

[2] This is the same representation as the SAT competition DIMACS format.

We say a clause `c`, or a formula `f`, is *satisfied* by an assignment `ta` if evaluation of `c`, or `f`, with respect to `ta` is `true`. An assignment that satisfies a formula is called a *solution*. We say a clause `c`, or a formula `f`, is *falsified* by an assignment `ta` if evaluation of `c`, or `f`, with respect to `ta` is `false`. A formula `f` is *satisfiable* if there exists a solution for `f` and *unsatisfiable* if there does not exist a solution for `f`. Two formulas are *logically equivalent* if and only if they have the same set of satisfying assignments. Two formulas are *satisfiability equivalent* if and only if they are both satisfiable or both unsatisfiable.

The negation of a clause is an assignment computed by `negate-clause`. For example, (`negate-clause` '(1 -2 3)) returns the assignment (-1 2 -3). The negation of an assignment is a clause computed by `negate-assignment`. Both functions share the same implementation and are complements of each other.

2.3 Proof Traces

Conflict-driven clause learning (CDCL) [9] is the leading paradigm of modern SAT solvers. A core aspect of CDCL solvers is the addition and removal of clauses. The main form of CDCL reasoning is known as conflict analysis, which adds *conflict clauses* encountered during search. Additionally, state-of-the-art CDCL solvers use preprocessing and inprocessing techniques that both add and remove clauses.

A clause `c` is *redundant* with respect to a formula `f` if (`cons c f`) is satisfiability equivalent to `f`. A formula is a set of clauses, and we write (`cons c f`) to extend a formula. We say the addition of a redundant clause `c` to `f` *preserves satisfiability*.

A *proof trace* is a sequence of clauses that are redundant with respect to an input formula. Note that a proof trace is a sequence because the order of the clauses in a proof trace is essential. As an example, two clauses `c1` and `c2` may both be redundant with respect to a formula `f`, but `c2` may not be redundant with respect to the extended formula (`cons c1 f`). A proof trace can be validated by a *proof checker* tool that iteratively (or recursively) removes the first clause `c` from a proof trace, checks the clause `c` for redundancy with respect to the current formula `f`, and then extends the set `f` with `c`. A validated proof trace that contains the empty clause, which cannot be satisfied, is called a *refutation*.

2.4 Resolution and Resolution Proofs

The early approaches to verify proof traces were based on resolution [12]. The *resolution rule* states that given a clause `c1` containing literal `l` and a clause `c2` containing literal (`negate l`) that the *resolvent* is the union of `c1` without `l` and `c2` without (`negate l`). The resolvent is logically implied by `c1` and `c2`. We compute resolvents with the function (`resolution l c1 c2`). Note that one can compute resolution without the use of a resolving literal, i.e. `l`. We chose this form in our model because it is more explicit and eases proof burdens.

Example 1. The clauses (1 -2) and (2 -3) contain a conflicting literal, i.e, one clause contains `l` and the other contains (`negate l`). Therefore, we can apply resolution on them. (`resolution -2 '(1 -2) '(2 -3)`) results in (1 -3).

Resolution proof traces are a sequence of clauses whose redundancy can be established by a sequence of resolutions on clauses from an input formula. Clauses added by CDCL solvers can be simulated by a sequence of resolutions [28]. Because resolution is such an elementary operation, simple and fast checking algorithms exist [12] that assure that a trace of resolution applications is correct. However, resolution proofs tend to be very large, and it may be hard to modify a SAT solver to emit a resolution refutation; for instance, one must determine the clauses on which to apply resolution, and specifying the order of resolutions can be difficult.

2.5 Extended Resolution

For a given formula `f`, the *extension rule* [24] allows one to iteratively add clauses to `f` encoding the logical AND of two existing literals as a new Boolean variable. More specifically, given variables `1`, `2` that appear in `f` and a variable `3` which does not appear in `f`, the clauses `((3 -1 -2) (-3 1) (-3 2))` can be added to `f`. *Extended Resolution* (ER) [24] is a proof system in which the extension rule is repeatedly applied to a formula `f`, followed by applications of the resolution rule. This proof system surpasses what can be expressed using only resolution.

ER [24] is the basis for some techniques used during learning [23] and preprocessing [21] in state-of-the-art SAT solvers. Refutations using ER can be exponentially smaller than refutations based solely on resolution. Examples include the pigeon-hole problems where Haken [29] showed that all resolution proofs are exponential in size, while Cook [30] demonstrated that some ER proofs can be polynomial in size. Our proof checker supports techniques that are based on ER.

2.6 Unit Propagation and Clausal Proofs

Goldberg and Novikov [19] proposed an alternative to resolution-based proofs. They observed that each clause added by CDCL conflict analysis can be checked using *unit propagation*, also known as *Boolean constraint propagation*. This technique simplifies a formula `f` based on unit clauses. A clause of any length is *unit* if all literals in the clause `c` evaluate to `false` under an assignment `ta` except for one, which evaluates to `undef`; this literal is called a *unit literal* and is added to `ta`. Adding the unassigned literal to `ta` makes `c` evaluate to `true` under the extended assignment. This procedure continues until a unit clause cannot be found.

Unit propagation can be used to check if a clause `c` is logically implied by a formula `f`. Start with the assignment `(negate-clause c)`. Apply unit propagation until a *conflict* arises, i.e., some clause in `f` is falsified. If a conflict occurs, then adding `c` to `f` preserves logical equivalence [19]. Clauses that can be checked using this procedure are also known as *reverse unit propagation* (RUP) clauses [13].

Example 2. Consider the formula `f = ((-1 2) (-2 -3) (3 4))` and the assignment `ta = (1 -4)`. Formula `f` under `ta` contains two unit clauses: `(-1 2)` with unit literal `2` and `(3 4)` with unit literal `3`. Extending `ta` with the unit literals results in the extended assignment `(1 2 3 -4)`. The extended assignment falsifies clause `(-2 -3)`, so unit propagation results in a conflict. Unit propagation on `f` under `(1 -4)` results in a conflict, which shows that clause `(-1 4)` is redundant with respect to `f`.

Because unit propagation will play a key role in our proof checker, we formalize this technique below. The function (`is-unit-clause c ta`) returns the unit literal if one exists or `nil` if `c` is not unit. The function (`find-unit-clause f ta`) recursively checks if each clause in `f` is unit, returns the multiple-value pair containing the unit literal and unit clause if a unit clause exists, and returns the multiple-value pair (`mv nil nil`) otherwise. Unit propagation for a formula `f` with respect to assignment `ta` is defined as follows:

```
(defun unit-propagation (f ta)            ;; Formula f, assignment ta
  (declare (xargs :measure (num-undef f ta))) ;; Termination measure
  (mv-let (ul uc)                         ;; Unit literal, unit clause
          (find-unit-clause f ta)         ;; Found by find-unit-clause
          (declare (ignorable uc))        ;; Unit clause not needed
          (if (not ul)                    ;; No unit literal?
              ta                          ;; Then, return assignment
            (unit-propagation f (cons ul ta))))) ;; Recur with new ta
```

The `mv-let` (Section 2.1) calls `find-unit-clause`, binds the results to `ul` and `uc`, declares that `uc` will be ignored, and executes the body which tests for a unit literal and recurs with an extended assignment if a unit literal is found.

ACL2 proves the termination of each function admitted to the logic. In most instances, ACL2 will be able to prove the termination of a function without additional help, but sometimes one might need to explicitly provide a measure. A measure is a function which computes a value that must decrease (with respect to a well-defined relation) during every recursive call. We provide a measure for `unit-propagation` in the definition above called `num-undef` that computes the number of clauses in `f` that evaluate to `undef` under `ta`. While `ta` is recursively *extended* during `unit-propagation`, the measure will *decrease* because each unit literal added to `ta` will make one `undef` clause become `true`.

2.7 Redundancy Hierarchy

There are many properties that can establish the redundancy of a clause. Järvisalo et al. [25] offers a hierarchy of fifteen redundancy properties that can be computed in polynomial time with respect to a formula. If a clause has one of those fifteen properties with respect to a given formula, then adding the clause to the formula preserves satisfiability. A discussion of all fifteen properties goes beyond the scope of this paper, so we will focus on the four properties that are related to our proof checker. A reduced hierarchy is shown in Fig 1.

We have already presented two redundancy properties. A clause has T (*tautology*) if and only if it contains the literals `l` and (`negate l`) for some `l`. A clause has AT (*asymmetric tautology*) if and only if reverse unit propagation results in a conflict. Note that any clause with property T trivially has the property AT. Many techniques used in SAT solvers can be expressed as a trace of clauses with AT including CDCL learning [31], variable elimination (DP resolution) [3,32] and subsumption. Adding clauses with T or AT to a formula preserves logical equivalence.

The other two redundancy properties are related to resolution. For a given literal `l` and a formula `f`, let `f-neg-l` denote the subset of clauses in `f` that contain the literal (`negate l`). First, a clause `c` has RT (*resolution tautology*) if and only if `c` has T or it contains a literal `l` such that all resolvents between `c` and a clause in `f-neg-l` have T. Second, a clause `c` has RAT (*resolution asymmetric tautology*) if and only if `c` has AT or it contains a literal `l` such that all resolvents between `c` and a clause in `f-neg-l` have AT. If a clause has any redundancy property in the hierarchy, then it also has RAT [25]. Techniques that can be expressed using RAT but not with AT include blocked clause addition [22], bound variable addition [21], extended resolution [24], and extended learning [23].

Example 3. Let formula `f` be (`(1 2) (2 3) (-2 -3)`).

- The clause (`1 -1`) is a tautology (has T) because it contains `1` and (`negate 1`).
- The clause (`1 -3`) does not have T. However, it has RT (and RAT) with respect to `f` and literal `1`, because `f` contains no clauses with literal `-1`. Furthermore, it also has AT because unit propagation with the assignment (`-1 3`) results in a conflict.
- The clause (`-1 3`) has RAT, but not T, AT, or RT. Unit propagation under the assignment (`1 -3`) does not result in a conflict, so (`-1 3`) does not have AT. Also, (`-1 3`) does not have RT, because there are non-tautological resolvents with (`1 2`) and (`-2 -3`). Finally, the only resolvent on literal `-1` is computed by resolving (`-1 3`) with (`1 2`) to obtain (`2 3`), which is already in `f`. So, unit propagation on the negation of the resolvent, (`-2 -3`), results in a conflict. Hence, (`-1 3`) has RAT.
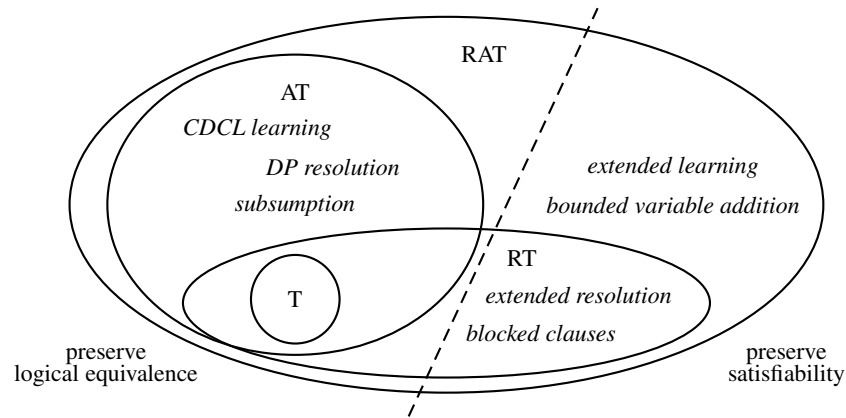
Fig. 1. Relationships between clause redundancy properties that can be computed in polynomial time with respect to the size of a formula. Techniques, shown in italics, are positioned based on the most efficient check to verify that technique. All techniques used in state-of-the-art SAT solvers can expressed as a sequence of RAT clauses [25]. The dashed line separates techniques that preserve logical equivalence and techniques that preserve satisfiability.

2.8 RAT

RAT is the strongest redundancy property in the hierarchy of [25] that preserves satisfiability. All preprocessing, inprocessing, and solving techniques in state-of-the-art SAT

solvers can be expressed in terms of addition and removal of RAT clauses [25]. Recall that a clause `c` has RAT if and only if `c` has AT or it contains a literal `l` such that all resolvents between `c` and a clause in `f-neg-l` have AT. A clause `c` has AT if unit propagation on the assignment (`negate c`) results in a conflict.

We define *resolution asymmetric tautology* (RAT) as follows:

```
(defun ATp (f c)   ;; Given formula f, clause c
  (falsep (evaluate-formula f
                            (unit-propagation f (negate-clause c)))))


;; Given clause list cl, formula f, clause c, and literal l
(defun RATp1 (cl f c l)
  (if (atom cl)                            ;; End of clause list?
      t                                    ;; Then, success
    (if (not (member (negate l) (car cl)))  ;; No resolution?
        (RATp1 (cdr cl) f c l)             ;; Then, continue
      (let ((r (resolution l c (car cl))))  ;; Make resolvent
        (if (tautologyp r))                ;; Resolvent has T?
            (RATp1 (cdr cl) f c l)         ;; Then, continue
          (and (ATp f r)                   ;; Resolvent has AT?
               (RATp1 (cdr cl) f c l)))))))  ;; Then recur, else fail


(defun RATp (f c l)   ;; Given formula f, clause c, and literal l
  (RATp1 f f c l))    ;; Copy f for recursion in helper function
```

The function `RATp` destructively recurs over a formula but needs a copy of the formula to compute asymmetric tautologies. Therefore, we begin by calling a helper function `RATp1` which has two copies of formula `f`. The first copy will be used for recursion and bound as `cl` while the second copy will remain untouched. If we have checked all clauses in `cl`, then we return `t` because `c` has RAT. If (`negate l`) is not a member of the current clause (`car cl`), then we recur. We next perform the resolution of `c` and (`car cl`) on `l` and bind the result to `r`. Finally, we check if `r` is a tautology or if `r` has AT. If either of these is true, we recur, and we return `nil` otherwise.

2.9 Proof Checker

We now present the implemenation of our proof checker. Our checker works by ensuring that each clause `c` in a proof trace `pt` has `ATp` with respect to formula `f` or `RATp` with respect to formula `f` on the first literal of the clause. If `c` can be verified, then `c` is added to `f` before recurring.

```
(defun verify-clause (c f)       ;; Given clause c, formula f
  (or (ATp f c)                  ;; Verify by AT, OR
      (and (not (atom c))        ;; Check for non-empty clause, AND
           (RATp f c (car c))))) ;; Verify by RAT w.r.t. 1st literal


(defun verify-proof (pt f)        ;; Proof trace pt, formula f
  (if (atom pt)                   ;; End of proof trace?
      t                           ;; Then, success
```

```
        (if (verify-clause (car pt) f)  ;; First clause in pt verified?
            (verify-proof (cdr pt)              ;; Then, recur with
                          (cons (car pt) f))  ;; extended formula
          nil)))                              ;; Else, fail
```

To be clear, we assume that the first literal of the clause (car c) is the literal on which to perform resolution during the RATp check. This is a design choice and the efficiency of this method is described in [26]. Note that the empty clause () will fail the (not (atom c)) case in verify-clause. We do this because () does not have an explicit resolution literal.

One can simply redefine verify-clause to only check clauses for ATp, removing the call to RATp. This creates a traditional RUP proof checker.

Example 4. Let formula f = ((1 2 -3) (-1 -2 3) (2 3 -4) (-2 -3 4) (-1 -3 -4) (1 3 4) (-1 2 4) (1 -2 -4)). A refutation for f is ((1) (2) ()).

3 Specification

We will introduce a few new concepts and then state our main theorem. The function clause-listp recognizes lists of clauses, similar to formulap. We define a proofp object to be a clause list that has been checked by our proof checker. A proof is a refutationp object if it also contains the empty clause. The predicate solutionp recognizes assignments that satisfy a given formula.

```
(defun proofp (pt f)            ;; A proof is a clause sequence
  (and (clause-listp pt)        ;; that has been verified with
       (verify-proof pt f)))    ;; respect to a formula

(defconst *empty-clause* nil)   ;; The empty clause

(defun refutationp (p f)        ;; A refutation is a proof that
  (and (proofp p f)             ;; contains the empty clause
       (member *empty-clause* p)))

(defun solutionp (ta f)         ;; A solution is an assignment
  (and (assignmentp ta)         ;; that satisfies a formula
       (truep (evaluate-formula f ta))))
```

We use the defun-sk event (Section 2.1) to define the notion that there exists a solution for a formula.

```
(defun-sk exists-solution (f)
  (exists ta (solutionp ta f)))
```

With those definitions, we can state our main theorem.

```
(defthm main-theorem
  (implies (and (formulap f)            ;; Given a formula f
                (refutationp r f))      ;; And refutation r
           (not (exists-solution f))))  ;; Then f is unsatisfiable
```

This theorem reads that if given a refutation `r` and a formula `f`, then there does not exist a solution for `f`. In other words, a refutation that has been verified by our RAT checker implies that a formula is unsatisfiable. Note that this is only a specification for correctness of our proof checker as defined above.

4 Proof

We will now describe the mechanical proof of `main-theorem`. ACL2-style proofs are generally a sequence of `defthm` and `defun` events. While ACL2 processes events in a bottom-up fashion, we provide a top-down description of our ACL2 proof. We want to prove that a refutation `r` for a formula `f` implies that the formula is unsatisfiable. An outline of the proof is as follows:

1. We will prove the contrapositive—if there exists a solution `s` for `f`, then we should not be able to verify the refutation.
2. We prove that the empty clause cannot be redundant with respect to `f` provided `s` is a solution for `f`.
3. We show that every clause `c` in `r` is redundant. This contradicts (2) because the empty clause is a member of `r`. We use structural induction on `r` to prove this.
   (a) We show that clauses with `ATp` are redundant.
   (b) We show that clauses with `RATp` are redundant. We case split based on the result of (`evaluate-clause c s`).
      i. If (`evaluate-clause c s`) is `true`, then `s` is a solution for `c`.
      ii. If (`evaluate-clause c s`) is `undef`, then we construct a new solution `s+` that consists of `s` with an `undef` literal in `c`.
      iii. If (`evaluate-clause c s`) is `false`, then we construct a new solution `s*` that is `s` with the exception that one literal in `s` has been negated.

In order to prove `main-theorem`, we first expand the definition of `refutationp` and contrapose the call of `verify-proof` with `exists-solution`. We will now use structural induction on the proof trace `pt`.

```
(defthm verify-proof-induction
  (implies (and (clause-listp pt)
                (formulap f)
                (exists-solution f)
                (member *empty-clause* pt))
           (not (verify-proof pt f))))
```

Recall that at each step `verify-proof` adds a clause from the proof trace `pt` to the formula. In our induction step, we will show that clauses in the proof with `ATp` or `RATp` are redundant. This allows us to derive a contradiction because `*empty-clause*` is a member of the refutation, does not have `ATp` or `RATp` with respect to a satisfiable formula, and is therefore not satisfiability equivalent.

```
(defthm *empty-clause*-lemma
  (implies (solutionp s f)
           (not (ATp f *empty-clause*))))
```

We prove this lemma with set reasoning. Specifically, if an assignment falsifies a formula, then a superset of that assignment will falsify a 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. We exclude this case by performing a `(not (atom c))` check in `verify-clause`.

We now 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. We will prove that if there exists a solution for the formula, then there exists a solution for the formula extended with a clause from the proof trace. In other words, we want to show that the extended formula is satisfiability equivalent to the original formula.

Here the proof diverges based on whether a proof clause has `ATp` or `RATp`. We consider the `ATp` case in Section 4.1 and the `RATp` case in Section 4.2.

4.1 ATp

If a clause `c` has `ATp` with respect to a formula `f`, then we will show that `(cons c f)` is logically equivalent to `f` (and therefore satisfiability equivalent to `f`).

```
(defthm ATp-lemma
  (implies (and (ATp f c)
                (exists-solution f)
                (formulap f)
                (clausep c))
           (exists-solution (cons c f))))
```

We expand `(exists-solution f)` to obtain a witness solution `s`. We will use `s` as a witness for the term `(exists-solution (cons c f))` in the conclusion. We know that `s` satisfies every clause in the original formula, so it is sufficient to show that `s` satisfies the `ATp` clause. Recall the definition of `ATp`. We replace the clause `c` with an abstraction `(negate-assignment ta)`. As previously stated, `negate-assignment` and `negate-clause` are complements of each other; `(negate-clause (negate-assignment ta))` simplifies to `ta`. We are then left with the following theorem.

```
(defthm ATp-lemma-induction
  (implies (and (falsep (evaluate-formula f (unit-propagation f ta)))
                (truep (evaluate-formula f s))
                (formulap f)
                (assignmentp ta)
                (assignmentp s))
           (truep (evaluate-clause (negate-assignment ta) s))))
```

We want to show that there is an `l` such that `l` is a member of `s` and `(negate l)` is a member of `ta`. Let assignment `up-ta` be the result of `(unit-propagation f ta)`. Because `up-ta` falsifies `f`, there must be a clause `c*` that is falsified by `up-ta`. Since `s` satisfies `f`, `s` also satisfies `c*`. Let `l*` be the literal that is a member of `c*` and `s`. Notice that `(negate l*)` is a member of `up-ta`.

We will induct on the extended assignment `up-ta`. In the base case, `up-ta` is equal to `ta`. Thus, `(negate l*)` is a member of `ta` and `l*` is a member of `s`. In the induction

step, `up-ta` is (`cons ul ta`) for some unit clause `uc` with unit literal `ul`. Again, there must be a clause `c*` that is falsified by `up-ta` 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 `ta`. If (`negate l*`) is in `ta`, then we are done. Otherwise, `ul` is equal to (`negate l*`), i.e. (`negate ul`) is in `s`. Consquently, `uc` was not satisfied by `ul`. All literals in `uc` not equal to `ul` are falsified by `ta` 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 `ta` and `l**` is in `s`.

Commentary. The induction for `ATp-lemma-induction` is the most difficult part of the proof of `ATp-lemma`. First, the induction is blocked by (`negate-clause c`). We tried several abstractions, all of which affected the goal (`truep (evaluate-clause c s)`), before settling on the use of `negate-assignment`. This abstraction lets us perform the correct induction without significantly changing the goal. Second, the induction itself is subtle because of the custom measure provided to `unit-propagation`. The assignment continues to grow during every recursive call of `unit-propagation`, but the number of `undef` clauses decreases.

4.2   RATp

We wish to show that if there exists a solution `s` for formula `f` and a clause `c` has `RATp` with respect to `f` and literal `l` in `c`, then there is a solution for the set (`cons c f`).

```
(defthm RATp-lemma
  (implies (and (formulap f)
                (clausep c)
                (member l c)
                (exists-solution f)
                (RATp f c l))
           (exists-solution (cons c f))))
```

Let assignment `s` satisfy `f`. Consider the cases for (`evaluate-clause c s`).

- `true`: There exists a solution for (`cons c f`), namely `s`.
- `undef`: Choose literal `l+` in `c` such that (`evaluate-literal l+ s`) returns `undef`. Let assignment `s+` be the result of (`cons l+ s+`). (`evaluate-clause c s+`) returns `true` because (`evaluate-literal l+ s+`) returns `true`. Consider any clause `c1` in `f`. We know (`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`: Make a new assignment `s*` such that (`evaluate-literal l s*`) is `true` by removing (`negate l`) from `s` and then adding `l` to `s`. By construction, we have that (`evaluate-clause c s*`) is `true`. We now need to show (`evaluate-clause c1 s*`) is `true` for all `c1` in `f`.

Consider a clause `c1` in `f`. If literal (`negate l`) is not a member of `c1`, then we know 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 `ATp` with respect to `f`. By `ATp-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`. Because `lr` is in `r` and not in `c`, we know `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*`.

Commentary. One key observation during the proof of `RATp-lemma` was the need for a case split on `(evaluate-clause c s)`. We previously tried to use an induction on clause list from `RATp1` with `(not (truep (evaluate-clause c s)))` as a hypothesis, which was insufficient. Feedback from ACL2 led us to a full three-way case split, which strengthened the condition to `(falsep (evaluate-clause c s))`.

Another subtle part of the proof was the case of `tautologyp` for the resolvent during an induction of the clause list in `RATp1`. In this proof, we needed to 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.

4.3 Statistics

Our RAT proof checker formalization, specification, and mechanical proof of correctness[3] contain 93 ACL2 `defun` events and 282 ACL2 `defthm` events and certifies in approximately 45 seconds. Of those, 32 `defun` and 140 `defthm` events are specific to the RAT proof checker while the other 61 `defun` and 142 `defthm` events are part of our "library" of SAT concepts. This library includes code about sets, literals, clauses, evaluation, unit propagation, and parsing. The RAT proof checker contains 1088 lines of uncommented non-blank lines of ACL2 source and 2080 lines total; the associated SAT library contains 1121 uncommented non-blank lines of ACL2 and 1836 lines total. We can extract only the definitions that are necessary to create an executable version of the RAT proof checker; this can be expressed in just 26 `defun` events. ACL2 allows the user to supply custom hints for conjectures that are not proved automatically; hints are used to guide the theorem prover towards a proof. We had to provide custom hints for 33 of the 140 `defthm` events that are specific to the RAT proof checker and 12 of the 142 `defthm` events in the SAT library.

5 Conclusion

We presented the formalization, specification, and proof of correctness for a SAT proof checker in ACL2. Our proof checker is based the strong redundancy property RAT that preserves satisfiability. This allows us to validate refutations generated by state-of-the-art SAT solvers that make use of techniques based on extended resolution. We describe the first mechanically-verified proof checker to be complete with respect to all contemporary SAT-solving techniques.

We are developing a faster checker that will employ watched-literal data structures. Our current checker, although proven to be correct, is still too slow to use to check large

---

[3] The implementation and proof presented in this paper are available at the address `http://cs.utexas.edu/~nwetzler/itp13/`.

proofs. We do not expect it to be too difficult to map clauses from list data structures to memory arrays represented in a heap since the ACL2 formalization of arrays is actually given with a list-based semantics. The inclusion of pointers to implement watched-literals will require that we prove an invariant assuring that the watched-literal data structure is always properly maintained. A fast, verified RAT proof checker could be used improve ACL2 by way of a verified clause processor. In other words, ACL2 could construct a SAT encoding for a given subgoal and then call any off-the-shelf SAT solver that produces a solution or a proof trace, that could then be checked by our tool to admit a theorem of unsatisfiablity into the logic. A framework for this proof strategy is explored by Davis et al. in [33].

We submit that all SAT solvers should be able to emit UNSAT proofs that can be checked. Experimentation has shown us that UNSAT proofs can generally be checked using a C-based program with watched literals in a time similar to that required by contemporary solvers. In the future, we encourage all SAT-solver developers to make provisions for emitting RAT proof traces that can be verified using a checker, like the one presented here. Furthermore, it should be the focus of future research to devise elegant and efficient ways of producing RAT proof traces.

References

1. Goldberg, E.I., Prasad, M.R., Brayton, R.K.: Using SAT for combinational equivalence checking. In: Design, Automation and Test in Europe Conference and Exhibition (DATE), IEEE (2001) 114–121
2. Lescuyer, S., Conchon, S.: A reflexive formalization of a SAT solver in Coq. In: International Conference on Theorem Proving in Higher Order Logics (TPHOLs). (2008)
3. Davis, M., Putnam, H.: A computing procedure for quantification theory. Journal of the ACM (JACM) **7**(3) (July 1960) 201–215
4. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Communications of the ACM **5**(7) (July 1962) 394–397
5. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer-Verlag (2004)
6. Shankar, N., Vaucher, M.: The mechanical verification of a DPLL-based satisfiability solver. Electronic Notes in Theoretical Computer Science **269** (2011) 3–17
7. Marić, F.: Formalization and implementation of modern SAT solvers. Journal of Automated Reasoning **43**(1) (April 2009) 81–119
8. Marić, F.: Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. Theoretical Computer Science **411**(50) (November 2010) 4333–4356
9. Marques-Silva, J.P., Lynce, I., Malik, S.: 4. In: Conflict-Driven Clause Learning SAT Solvers. Handbook of Satisfiability, IOS Press (February 2009) 131–153
10. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)
11. Oe, D., Stump, A., Oliver, C., Clancy, K.: versat: a verified modern SAT solver. In: Verification, Model Checking, and Abstract Interpretation (VMCAI), Springer-Verlag (January 2012) 363–378
12. Zhang, L., Malik, S.: 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), IEEE (2003) 880–885

13. Van Gelder, A.: Verifying RUP proofs of propositional unsatisfiability. In: International Symposium on Artificial Intelligence and Mathematics (ISAIM). (2008)
14. Biere, A.: PicoSAT essentials. Journal on Satisfiability, Boolean Modeling and Computation **4**(75-97) (2008) 45
15. Weber, T.: Efficiently checking propositional resolution proofs in Isabelle/HOL. In: International Workshop on the Implementation of Logics (IWIL). Volume 212. (2006) 44–62
16. Weber, T., Amjad, H.: Efficiently checking propositional refutations in HOL theorem provers. Journal of Applied Logic **7**(1) (2009) 26–40
17. Darbari, A., Fischer, B., Marques-Silva, J.: Industrial-strength certified SAT solving through verified SAT proof checking. In: Theoretical Aspects of Computing (ICTAC), Springer-Verlag (September 2010) 260–274
18. Armand, M., Faure, G., Grégoire, B., Keller, C., Théry, L., Wener, B.: Verifying SAT and SMT in Coq for a fully automated decision procedure. In: PSATTT'11: International Workshop on Proof-Search in Axiomatic Theories and Type Theories. (2011)
19. Goldberg, E.I., Novikov, Y.: Verification of proofs of unsatisfiability for CNF formulas. In: Design, Automation and Test in Europe Conference and Exhibition (DATE), IEEE (2003) 10886–10891
20. Oe, D., Stump, A.: Combining a logical framework with an RUP checker for SMT proofs. In: Satisfiability Modulo Theories (SMT). (2011) 40
21. Manthey, N., Heule, M.J.H., Biere, A.: Automated reencoding of boolean formulas. In: Proceedings of Haifa Verification Conference. Volume 7364 of LNCS. (2013) 102–117
22. Kullmann, O.: On a generalization of extended resolution. Discrete Applied Mathematics **96-97** (1999) 149–176
23. Audemard, G., Katsirelos, G., Simon, L.: A restriction of extended resolution for clause learning SAT solvers. In Fox, M., Poole, D., eds.: Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI), AAAI Press (2010)
24. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In Siekmann, J., Wrightson, G., eds.: Automation of Reasoning 2. Springer-Verlag (1983) 466–483
25. Järvisalo, M., Heule, M.J.H., Biere, A.: Inprocessing rules. In: International Joint Conference on Automated Reasoning. Volume 7364 of LNCS., Springer (2012) 355–370
26. Heule, M.J.H., Hunt, Jr., W.A., Wetzler, N.: Verifying refutations with extended resolution. In: International Conference on Automated Deduction (CADE). Volume 7898 of LNAI., Springer (2013) 345–359
27. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, Boston (June 2000)
28. Beame, P., Kautz, H., Sabharwal, A.: Towards understanding and harnessing the potential of clause learning. Journal of Artificial Intelligence Research (JAIR) **22** (2004) 319–351
29. Haken, A.: The intractability of resolution. Theoretical Computer Science **39** (1985) 297–308
30. Cook, S.A.: A short proof of the pigeon hole principle using extended resolution. SIGACT News **8**(4) (October 1976) 28–32
31. Marques Silva, J.P., Sakallah, K.A.: Grasp: A search algorithm for propositional satisfiability. IEEE Trans. Computers **48**(5) (1999) 506–521
32. Eén, N., Biere, A.: Effective preprocessing in sat through variable and clause elimination. In Bacchus, F., Walsh, T., eds.: Theory and Applications of Satisfiability Testing. Volume 3569 of Lecture Notes in Computer Science., Springer (2005) 61–75
33. Davis, J., Swords, S.: Verified AIG algorithms in ACL2. In Gamboa, R., Davis, J., eds.: ACL2. Volume 114 of EPTCS. (2013) 95–110