Efficient, Verified Checking of Propositional Proofs

Marijn Heule[1(✉)], Warren Hunt Jr.[1], Matt Kaufmann[1], and Nathan Wetzler[2]

[1] The University of Texas at Austin, Austin, TX, USA
[2] Intel Corporation, Hillsboro, OR, USA
{marijn,hunt,kaufmann}@cs.utexas.edu, nathan.wetzler@gmail.com

Abstract. Satisfiability (SAT) solvers—and software in general—sometimes have serious bugs. We mitigate these effects by validating the results. Today's SAT solvers emit proofs that can be checked with reasonable efficiency. However, these checkers are not trivial and can have bugs as well. We propose to check proofs using a formally verified program that adds little overhead to the overall process of proof validation. We have implemented a sequence of increasingly efficient, verified checkers using the ACL2 theorem proving system, and we discuss lessons from this effort. This work is already being used in industry and is slated for use in the next SAT competition.

1 Introduction

This paper presents a formally verified application, a SAT proof-checker, that has sufficient efficiency to support its practical use. Our checker, developed using the ACL2 theorem-proving system [12,15], validates the results of SAT solvers by checking the emitted proofs. Our intention here is to provide some useful lessons from the development of an efficient, formally verified application using ACL2. We therefore avoid lower-level details of algorithms, mathematics, and proof development.

The Problem. Boolean satisfiability (SAT) solving has become a key technology for formal verification. Users of SAT solvers increasingly seek confidence in claims that given formulas are unsatisfiable[1]. Contemporary SAT solvers therefore emit proofs [10] that can be validated by *SAT proof-checkers* [9,27]. Such a proof is a sequence of *steps*, each of which is interpreted as transforming a formula to a new formula. Checking the proof is just the result of iterating through the steps; for each step, the checker performs a validation intended to guarantee that if the current formula (initially the input formula) is satisfiable, then so

This work was supported by NSF under Grant No. CCF-1526760. We thank the reviewers for useful feedback.

[1] Checking a claim of satisfiability is easy.

M. Ayala-Rincón and C.A. Muñoz (Eds.): ITP 2017, LNCS 10499, pp. 269–284, 2017.
DOI: 10.1007/978-3-319-66107-0_18

is the transformed formula. Typically the final formula is clearly unsatisfiable; then the validation process guarantees that the input formula is unsatisfiable.

How can we trust SAT proof-checkers? Although they are usually much simpler than SAT solvers, they are not trivial, and any software is susceptible to bugs. We implemented a verified SAT proof-checker in ACL2 [26], but this checker was not intended be *efficient*. For example, a specific proof that was validated in about 1.5 s by the unverified checker `DRAT-trim` [27] took about a week to validate using this verified checker. Several reasons explain the slowdown. The verified checker used list-based data structures, providing linear-time accesses, while the unverified checker used arrays and various low-level optimizations. Additionally, proofs of unsatisfiability usually contain many *deletion steps*, while deletion is not supported by that verified checker. The size of the formula is important because a key procedure, the *RAT* check [11], may need to consider every clause in the formula. Finally, RAT checking is based on a procedure, *unit propagation*, that can require expensive search. (These two aspects of RAT checks—some checks that are linear in the size of the formula, and search—are all we need to know about RAT checks for this paper.)

Alternatively one could verify the correctness of the solver in a theorem prover. That approach does not require proof logging and validation. However, SAT solvers are complicated and frequently improved, thereby making the verification task hard. Moreover, verified SAT solvers tend to be orders of magnitude slower compared to unverified solvers [1]. That said, verification of SAT solvers has been studied by various authors in the last decade. The DPLL [4,5] algorithm, which was the core algorithm of solvers until the late 90's, has been formalized and verified by Lescuyer and Conchon [17] in Coq and by Shankar and Vaucher [23] in PVS. The conflict driven clause-learning paradigm of modern SAT solvers [20] was verified by Marić [18,19] in Isabelle/HOL (2010), by Oe et al. [22] in Guru (2012), and by Blanchette et al. [1] in Isabelle/HOL (2016).

Towards a Solution. At least three parallel efforts have attempted to produce efficient, formally verified SAT checkers [3,16]. A key idea was to avoid all search (all of which results from unit propagation) by adding certain "hints" to each proof step, resulting in a new proof format, LRAT (Linear RAT) [3]. In this paper we discuss one of those three efforts: an LRAT proof-checker developed in ACL2 (the others being checkers in Coq [3] and Isabelle/HOL [16]). The SAT proof mentioned above that took a week to check now takes under 3 s to check with the new ACL2-based checker. As suggested by some data reported below, our checker may run sufficiently fast so that it adds relatively little overhead beyond using a fast C-based checker. This work is already used in industry at Centaur Technology [25], and we expect it to be used in the 2017 SAT competition. In this system, one does not need to reason about the original proof produced by a solver or the proof conversion process of DRAT-trim: if our verified checker validates the final optimized proof, then the input formula is unsatisfiable.

This paper is not intended to provide proof details, but rather, to extract some lessons in the effective use of one proof assistant (ACL2). This paper

assumes no knowledge of ACL2, SAT solving, or SAT proof-checking (such as RAT and LRAT); all necessary background is provided above or as needed below.

We begin with a few ACL2 preliminaries. Then in Sect. 3 we describe a sequence of increasingly efficient checkers. That description provides background for discussion in Sect. 4 of the ACL2 soundness proofs done for each of these checkers. Section 5 concludes with remarks that summarize our findings.

2 ACL2 Preliminaries

The ACL2[2] theorem-proving system [12,15] includes a programming language based on an applicative subset of Common Lisp [24]. Lisp is one of the oldest programming languages [21] and is supported by several efficient compilers, both commercial and free. Moreover, ACL2 was designed with efficient execution in mind [7,28]; indeed, efficiency is important since the ACL2 theorem prover is mostly written in its own language. Thus, ACL2 provides a platform where one can write programs that execute efficiently and also prove programs correct.

We focus below on three ACL2 features that support efficient execution of our SAT proof-checker: *guards*, *stobjs*, and *fast-alists*. Then we close this section by explaining ACL2 notions used in the rest of this paper.

Guards. The ACL2 logic is an untyped first-order logic of total functions. The expression `(first 3)` denotes the application of a function, `first` to a single parameter, `3`. Thus, even a "bad" expression like `(first 3)`—`first` is intended to be applied to a list (to return its first element), not a number—are logically well-formed. Indeed, ACL2 can prove that `(first 3)` is equal to `(first 4)`, and ACL2 provides a way to evaluate `(first 3)` without error. On the other hand, Common Lisp signals an error when evaluating this expression. It would be wrong for ACL2 to use Common Lisp to do all of its evaluation, while taking advantage of modern Common Lisp compilers is exactly what we want to do.

A solution is provided by *guards*. The ACL2 guard for a function is an expression whose variables are all formal parameters of that function. Guards can be viewed as analogous of types, in that they are preconditions on the arguments of a function. In contrast with most type systems, however, a guard can be any expression involving any subset of the formal parameters of a function. For example, the guard for `first`, with formal parameter `x`, is that `x` is a list.[3]

ACL2 relies on Common Lisp to evaluate using the definitions provided, but only after *guard verification* is performed on those definitions: proving formulas guaranteeing that for every function call during evaluation, the arguments of that call satisfy its function's guard. Guard verification was an important part of our proof effort (see Sect. 4.3), resulting in a verified checker that executes efficiently in Common Lisp.

[2] "A Computational Logic for Applicative Common Lisp".

[3] More accurately, `first` is a macro expanding to a corresponding call of the function `car`, whose guard specifies that the argument is a pair or the empty list.

Stobjs. Single-threaded objects, or *stobjs* [2], are mutable objects that support fast execution in ACL2.[4] A stobj s is introduced as a record with fields, some of which may be arrays. Henceforth, s may be an argument to a function, but ACL2 enforces syntactic requirements, in particular: if s is modified by a function then it must be returned, and its use must be single-threaded. Such restrictions guarantee that there is only one instance of s present at any time during evaluation, and therefore it is sound to modify s in place, which can boost speed significantly since it avoids allocating new structures.

Fast-Alists. In Lisp parlance, an *alist* (or *association list*) is a representation of a finite function as a list of ordered pairs $\langle i, j \rangle$ for which the *key*, i, is mapped to the *value*, j. ACL2 supports so-called *fast-alists*, sometimes called *applicative hash tables*. For any fast-alist, the implementation provides a corresponding hash table so that the function `hons-get` obtains the value for a given key in essentially constant time—provided a certain single-threaded discipline is maintained. Unlike stobjs, the discipline is not enforced at definition time; instead, a runtime warning is printed when it is violated, in which case the alist is searched linearly until a pair $\langle i, j \rangle$ is found for a given key, i. In practice, it is straightforward for ACL2 programmers to use fast-alists so that the discipline is maintained.

Other Preliminaries. We mention a few other aspects of ACL2, towards making this paper self-contained. The ACL2 prover is extensively discussed in its documentation[5] and in other places [12,15]. While automated induction is certainly helpful, the "workhorse" of the prover is rewriting. Definitions and (by default) theorems are stored as rewrite rules. It is often helpful to *disable* (turn off) some rules either to speed up the prover or to implement some rewriting strategy. A *book* is an ACL2 input file, typically containing definitions and theorems. Finally, symbols are case-insensitive and in particular, Boolean values are represented by the symbols `T` (true) and `NIL` (false).

3 SAT Proof-Checker Code

Our most efficient SAT proof-checker is the last in a sequence of verified SAT proof-checkers developed in ACL2. Section 3.1 enumerates these checkers, providing a name and some explanation for each. The statistics provided in Sect. 3.2 demonstrate improved performance offered by each successive checker. All supporting materials for the checkers listed below, including proofs, may be found in the `projects/sat/lrat/` directory within the ACL2 community books[6]; see its README file.

[4] Thus, stobjs play a role somewhat like monads in higher-order functional languages.
[5] http://www.cs.utexas.edu/users/moore/acl2/current/manual/.
[6] https://github.com/acl2/acl2/tree/master/books/.

3.1 A Sequence of Checkers

[**rat**] **A Verified RAT Checker** [26]**.** A formula is a list of clauses, implicitly conjoined (hence, in what is typically called *conjunctive normal form*). A proof designates an ordered sequence of clauses, each of which is to be added to the formula, in order. The RAT check is intended to ensure that when a clause C in the proof is added to the current formula F: if F is satisfiable, then F remains satisfiable after adding C. The RAT check is proved sound: if the proof passes that check and contains the empty clause, then the original formula is unsatisfiable.

[**drat**] **A Verified DRAT Checker.** Our first proof effort was to extend the verified RAT checker to handle deletion—the "D" in "DRAT"—of clauses from a formula. Thus a proof step became a pair consisting of a Boolean flag and a clause, where: a flag of `T` indicates that the clause is to be added, as before; but a flag of `NIL` indicates that the clause is to be removed. Since deletion obviously preserves satisfiability, we quite easily modified the [rat] soundness proof to accommodate this enhanced notion of SAT proof.

Only modest benefit might accrue from extending the initial checker in a straightforward way with deletion: on the easiest problem in our test suite, [rat] requires 20 s, while [drat] took 9 s. All [lrat-*] checkers can verify the proof of the same problem in a fraction of a second. However, it is well established that without deletion, high-performance checkers will suffer greatly [9]. Thus, incorporating deletion was an important first step.

[**lrat-1**] **A Verified LRAT Checker Using Fast-Alists.** In order to speed up SAT proof-checking, we wanted to exploit proof hints recently provided by the *LRAT* format [3], which facilitate fast lookup of clauses in formulas. So we developed an ACL2 checker that represents formulas using fast-alists, which provide a Lisp hash-table for nearly constant-time lookup. Our fast-alists contain pairs of the form $\langle i, c\rangle$, where the key, i, is a positive integer that denotes the index of the associated clause, c. But a formula can also contain pairs $\langle i, \texttt{D}\rangle$ where `D` is a special deletion indicator, meaning that the clause with index i has been deleted from the formula. A deletion proof step provides an index i to delete, and is processed by updating the formula's fast-alist with a new pair $\langle i, \texttt{D}\rangle$.

For this checker, a formula is actually an ordered pair $\langle m, a\rangle$, where a is a fast-alist as described above and m is the maximum index in that alist. That value is passed to the function that may be called to perform a *full RAT check*, which recurs through the entire formula starting with index $i = m$. Each step in that recursion looks up i in the fast-alist to find either a clause that is checked, or the deletion indicator, `D`. The repeated use of the lookup function, `hons-get`, on clause indices turned out to be somewhat expensive, in spite of its use of a Lisp hash-table. That expense is addressed with improvements discussed below.

[**lrat-2**] **A Faster Verified LRAT Checker that Shrinks Fast-Alists.** ACL2 supports profiling, which we used on the [lrat-1] proof-checker. We found that 69% of the time was spent performing lookup with `hons-get`. On reflection,

this was not a surprise: the full RAT check walks through the entire (fast-)alist, which grows with every proof step that adds a clause. This quadratic behavior would not be present if fast-alists were nothing more than mutable hash tables; but in ACL2 they are also alists, which grow with each update. Note the [lrat-1] checker applies `hons-get` at every step of the full RAT check: the ordered pair $\langle i, c\rangle$ seems to suggest that a suitable check needs to be done on the clause c, but this pair may be overridden by a pair $\langle i, \texttt{D}\rangle$ in the formula indicating that the clause c has actually been deleted, and thus should not be checked.

This checker (also those that follow) heuristically chooses when to shrink the formula's fast-alist, by removing from it all traces of deleted clauses. This happens immediately before checking any proof step's addition of a clause, whenever the number of deleted clauses in the formula exceeds the number of *active* (not deleted) clauses by at least a certain factor. Based on some experimentation, that factor is set to 1/3 when about to do the full RAT check, which as mentioned above must consider every clause in the formula; otherwise, the factor is set to 10. The function `shrink-formula-fal` creates a smaller formula, equivalent to its input, by removing pairs that represent deletion. It does this by first using an ACL2 primitive that exploits the underlying hash table to remove, very efficiently, all pairs $\langle i, c\rangle$ that are overridden by deletion pairs $\langle i, \texttt{D}\rangle$; a linear walk removing all deletion pairs, followed by creation of a new fast-alist, then finishes the job.

[**lrat-3**] **A Verified LRAT Checker with a Simpler Representation of Formulas.** The previous version still represents a formula as a pair $\langle m, a\rangle$, where a is a fast-alist and m is its maximum index. The [lrat-3] checker represents a formula simply as a fast-alist, since starting with [lrat-2], the full RAT check recurs through the fast-alist without needing the maximum index in advance. Other improvements (all small) include better error messages.

[**lrat-4**] **A Verified LRAT Checker with Assignments Based on Single-Threaded Objects.** The previous versions all represent an assignment as a list of (true) literals. Our next change was to represent assignments using single-threaded objects in order to improve performance. Profiling showed that most of the time in [lrat-3] was being spent evaluating clauses and literals. Evidently, the linear lookup into a long assignment (list of literals) can be expensive. Using a stobj avoids memory allocation for assignments, but probably much more important, it supports constant-time evaluation of literals.

Our stobj, `a$`, contains the three fields below. It uses standard representations: of propositional variables as natural numbers, of literals as non-zero integers, and of logical negation as arithmetic negation (-5 represents "not 5").

- `a$arr`: an array whose ith value is `T`, `NIL`, or 0, according to whether variable i is true, false, or of unknown value
- `a$stk`: a stack of variables, implemented as an array
- `a$ptr`: a natural number indicating the top of the stack

Returning to the `a$` stobj, we observe that the `a$arr` field alone does not provide direct support for reverting an assignment after having extended it.

We use a standard "trail" [6] approach to address this, by creating a stack of variables that have been assigned, such that whenever a Boolean value is written at position V of `a$arr`, V is also pushed onto the stack, by writing V at position `a$ptr` of `a$stk` and then incrementing `a$ptr`. That extension is undone by way of an inverse operation: the variable at the top of the stack serves as an index into `a$arr` at which to write 0 ("unassigned"), and then the stack pointer `a$ptr` is decremented.

[**lrat-5**] **Compression and Incremental Reading.** SAT proofs have grown to the point where the proof files that need to be certified are gigabytes in size. To help manage the sheer size of these proofs, we developed a lightweight procedure to compress LRAT files into CLRAT (Compressed LRAT) files, using techniques similar to those used for compression of DRAT files [8]. Our compression results in files about 40% the size of the original. Our CLRAT proof-file reader is guard-verified, both to support efficient execution and to increase confidence that we are parsing the input in a manner consistent with its specified syntax.

Compressed files reduce the size of proof files, but they do not reduce the number of proof steps that must be processed. Our earlier SAT proof-checkers read an entire proof file (into memory) before checking the veracity of every proof step, but given the ever increasing size of proof files, this approach is no longer tenable. We can now read SAT proofs in sections, for example of a few megabytes each; thus, we read (some of a proof file), then check (part of a proof), then read some more, then check some more, and so on, thus supporting proof files of arbitrary length. This checker has the highest performance of all of our verified SAT proof-checkers.

To provide for incrementally reading a large file, we extended the ACL2 function `read-file-into-string` so that it could read successive segments of the file, as specified by the user. Our correctness proof confirms that performing the interleaved file-reading and proof-checking is sound. The main advantage of interleaving proof reading and proof validation is that we can avoid having the entire proof in memory, which significantly reduces the memory footprint of the checker.

3.2 Performance

Table 1 compares performance for the checkers discussed above[7]. All runtimes are in seconds and include both parsing and checking time, and each is labeled by the proof file for the run. Each column header indicates one of the checkers discussed above, with a reminder of how it differs from the preceding checker. The [lrat-5] times do not include the use of `diff` described in Sect. 4.4, although that was done and was measured at under 1/50 s in each case. We omit columns for [lrat-2] (which is similar to [lrat-3] and for the early checkers that were much less efficient (for example, roughly one week for [rat] on R_4_4_18).

[7] We used ACL2 GitHub commit `639ef8760d30a63e2f21e160cdf02b75e1154fcc` and SBCL Version 1.3.15, on a 3.5 GHz Intel(R) Xeon(R) with 32 GB of memory running on Ubuntu Linux.

Notice that an improvement can make much more of a difference for some tests than for others. In particular, as we move down through the last two columns we see that the list-based [lrat-3] checker compares well with the stobj-based [lrat-4] checker, until we get to a hard benchmark from the SAT 2016 competition, "Schur_161_5_d43.cnf", with a 5.6 GB proof (a rather typical size). Profiling showed that most of the time for [lrat-2] and [lrat-3] is in evaluating clauses and literals with respect to assignments. Since an [lrat-3] assignment is a linear list (of all true literals), it makes sense that the constant-time array access provided by an [lrat-4] stobj can reduce the time considerably. The [lrat-5] time of just over 4 min adds less than 25% to the 20 min it takes for the DRAT-trim checker to process a DRAT proof into an LRAT proof, which bodes well for using [lrat-5] in SAT competitions.

Table 1. Times in seconds when running checkers on various inputs

benchmark	[lrat-1] *(fast-alist)*	[lrat-3] *(shrink)*	[lrat-4] *(stobjs)*	[lrat-5] *(incremental)*
uuf-100-3	0.09	0.03	0.05	0.01
tph6[-dd]	3.08	0.57	0.33	0.33
R_4_4_18	164.74	5.13	2.23	2.24
transform	25.63	6.16	5.81	5.82
Schur_161_5_d43	5341.69	2355.26	840.04	259.82

We also produced RAT proofs of all application benchmarks of the SAT 2016 Competition that CryptoMinisat 5.0[8] could solve in 5000 s. We choose CryptoMinisat as it produces proofs with the most RAT clauses among those solvers that participated in the SAT 2016 Competition. CryptoMinisat solved 95 unsatisfiable benchmarks within the time limit. On 5 problems we ran into memory issues when converting the DRAT proof produced by CryptoMinisat into CLRAT proofs. One benchmark used duplicate literals, which is not allowed in our formalization. Figure 1 shows the results on the 89 validated proofs. For benchmarks that can be solved within 20 s, solving, DRAT to CLRAT conversion, and verified CLRAT checking are similar. For hard problems, solving takes about one third the time compared to DRAT to CLRAT conversion, while verified CLRAT checking takes about one third the time compared to solving. Hence, verified CLRAT checking adds relatively small overhead to the tool chain.

4 Correctness Proofs

We next consider, in order, each checker of the preceding section except the first, [rat], explaining some key high-level approaches to its correctness proof. Our focus is not on proving the basic algorithm correct, as this was done previously for the [rat] checker [26], including an analogue of the key inductive step (called

[8] https://github.com/msoos/cryptominisat.

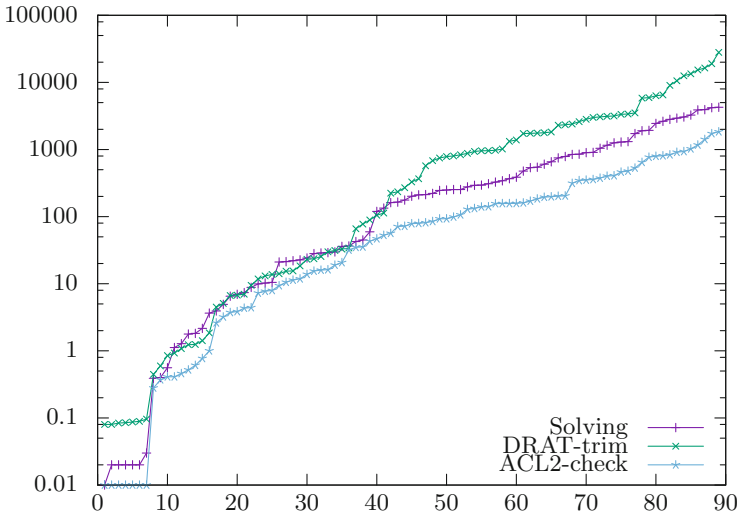

Fig. 1. Cactus plot of the solving times (including DRAT proof logging) of benchmarks for the SAT 2016 competition application benchmarks using CryptoMinisat, the validation times (including CLRAT proof logging) of DRAT-trim, and checking CLRAT proofs using ACL2-check.

`satisfiable-add-proof-clause` in Sect. 4.2): it preserves satisfiability to add a validated clause from an alleged proof. Rather, we discuss the steps taken in order to yield proofs for increasingly efficient code.

All of the soundness theorems for [rat] up through [lrat-3] have essentially the form displayed below: given a formula (list of clauses) and a valid refutation of it, then that formula is unsatisfiable. We will see a small variant for [lrat-4] and a major improvement for [lrat-5].

Soundness.

```
(implies (and (formula-p formula)
              (refutation-p proof formula))
         (not (satisfiable formula))))
```

4.1 Deletion ([drat])

Our first checker is a replacement for the initial checker [26]. A comparison of the two books shows that the original structure was preserved, the key difference being in the notion of a proof step: instead of a clause, it is a pair consisting of a flag and a clause, where the flag indicates whether the clause is to be added to the formula or deleted from it. Conceptually, deletion is trivially correct: if a formula is satisfiable, then it is still satisfiable after deleting one of its clauses. Our soundness proof effort took advantage of the automation provided by ACL2, in particular conditional rewriting: most lemmas were still proved automatically when we modified the checker, and the rest were straightforward to fix.

4.2 Linear RAT ([lrat-1], [lrat-2], [lrat-3])

In this section we discuss some lessons that can be learned from the proofs of soundness for the first three LRAT checkers [lrat-1], [lrat-2], and [lrat-3] described in Sect. 3. Recall that these checkers departed from [drat] by using fast-alists in the representation of formulas.

In order to deal with the new formula data structure and the new proof hints provided by the LRAT format, we chose to develop the soundness proof from scratch, since the main developer for these new checkers was not very familiar with the [rat] development. An early step was to write out a hand proof, so as to avoid getting lost in a proof of this complexity. We started with a top-down approach, supported by ACL2 utility `skip-proofs` [14]: first prove the main result from the key lemmas (whose proofs are skipped), then similarly prove each key lemma from its (proofs skipped) key sublemmas, and so on.[9]

To see this top-down style in action, consider the [lrat-1] book `satisfiable-add-proof-clause.lisp`. As displayed below (with comments added, each following a semicolon (;)), that book locally includes two books that each prove a key lemma in order to export those lemmas (not the other contents of the books) from its scope, which are then used to prove the desired theorem.

```
(local ; Do not export the following outside this book.
 (encapsulate () ; Introduce a scope
   ; Load the two indicated books.
   (local (include-book"satisfiable-add-proof-clause-rup"))
   (local (include-book "satisfiable-add-proof-clause-drat"))
   ; Export two key lemmas outside the encapsulate scope.
   (defthm satisfiable-add-proof-clause-rup ...)
   (defthm satisfiable-add-proof-clause-drat ...)))
; Prove the main theorem of this book.
(defthm satisfiable-add-proof-clause
  ; Theorem statement is omitted in this display.
  :hints (("Goal"
           ; Prove that the two key lemmas imply this theorem.
           :use (satisfiable-add-proof-clause-rup
                 satisfiable-add-proof-clause-drat)
           ; Disabling most rules improves reliability and speed.
           :in-theory (union-theories '(verify-clause)
                                      (theory 'minimal-theory)))))
```

The [lrat-1] book `sat-drat-claim-2-3.lisp` also follows our hand proof.

The correctness proof for [lrat-1] was tedious, but presented no surprises. One key proof technique, found in the [lrat-1] book `soundness.lisp`, is to define a

[9] The hand proof may be found in a comment near the top of the book `satisfiable-add-proof-clause.lisp` (see for example community books directory `projects/sat/list-based/`). That informal proof is annotated with names of lemmas from the actual proof script.

function `extend-with-proof` that recurs much like the checker, except instead of returning a Boolean, it returns the formula produced by applying the proof steps in sequence, starting with the original formula, with each step deleting or adding a clause. The following lemma is then key; with enough lemmas in place, it is proved automatically by induction using the recursion scheme for that function.

```
(defthm proof-contradiction-p-implies-false
  (implies (and (formula-p formula)
                (proofp proof)
                (proof-contradiction-p proof))
           (equal (evaluate-formula (extend-with-proof formula proof)
                                    assignment)
                  nil)))
```

Of course, the phrase "enough lemmas in place" above hides all the real work in the proof, for example in proving that the RAT check suffices for concluding that the addition of a clause preserves satisfiability.

With the proof of [lrat-1] complete, the next step was to improve efficiency by shrinking the formula from time to time, as explained in the description of [lrat-2] in Sect. 3. The [lrat-2] code was thus structurally similar but incorporated this shrinking. By keeping the top-level shrinking function disabled, it was reasonably straightforward to update the proof. Our process was to see where the former proof failed: when an ACL2 proof fails, it prints *key checkpoints*, which are formulas that can no longer be simplified. They often provide good clues for lemmas to formulate and prove.

The migration from [lrat-2] to [lrat-3] was very easy, including modifying the soundness proof. The key change was to avoid storing a maximum index field in the formula, so that the formula became exactly its fast-alist. This change had little effect on efficiency, though it did avoid some memory allocation (from building `cons` pairs). Rather, the point was to simplify the proof development, in preparation for our final step.

4.3 Using Stobjs ([lrat-4])

The introduction of stobjs for assignments presented the possibility of modifying the existing soundness proof. However, that seemed potentially difficult, given the disparity in the two representations of assignments: in the list version, assignments are extended using `cons` and retracted by going out of the scope of a `LET` binding; by contrast, the stobj version modifies assignments by updating array entries and stack pointers.

So instead of modifying the proof of the [lrat-3] soundness theorem, we decided to *apply* that theorem by relating the [lrat-3] list-based checker and the [lrat-4] stobj-based checker. A summary of that approach is presented below, followed by some deeper exploration. See the [lrat-4] (stobj-based) book `equiv.lisp` for the ACL2 theorems that relate the two checkers.

Applying [lrat-3] Correctness Using a Correspondence Theorem. The [lrat-3] and [lrat-4] checkers are connected using the correspondence

theorem below, `refutation-p-equiv`[10]. It is formulated using a function `refutation-p$`, defined for the stobj-based checker in analogy to the list-based recognizer function `refutation-p` for valid refutations, but using a so-called *local stobj*.

```
(defthm refutation-p-equiv
  (implies (and (formula-p formula)
                (refutation-p$ proof formula))
           (refutation-p proof formula)))
```

That correspondence theorem trivially combines with the list-based checker's soundness theorem (stated near the beginning of Sect. 4) to yield soundness for the stobj-based checker.

```
(defthm main-theorem-stobj-based
  (implies (and (formula-p formula)
                (refutation-p$ proof formula))
           (not (satisfiable formula))))
```

Guard Verification and a Stobj Invariant. Our first step was to verify guards for the stobj-based checker definitions, to support high-performance execution. This step was undertaken before starting the proof of `refutation-p-equiv` or its supporting lemmas, so that useful insights and lemmas developed during guard verification could be reused when developing the correspondence proofs. In particular, it was clear that guard verification would require developing an invariant on the stobj—e.g., to guarantee that extending an assignment never writes to the stack at an out-of-bounds index—and that proving invariance could be useful when proving the correspondence theorems.

The stobj invariant, `a$p`, is defined in terms of several recursively-defined properties. Informally, it says that the stack and array of `a$` correspond nicely: the stack has no duplicates, and the variables below the top of the stack are exactly the variables with an assigned value of true (`T`) or false (`NIL`) in the array, as opposed to being undefined (value 0). It was rather challenging to complete all of the guard verification, but then perhaps more straightforward to prove the correspondence theorems, culminating in the theorem `refutation-p-equiv` shown above.

A Challenge in Proving Correspondence. A glitch arose while attempting the correspondence proofs. Consider the following correspondence theorem.

```
(defthm negate-clause-equiv-1
  (implies (and (a$p a$)
                (= (a$ptr a$) 0)
```

[10] The subsidiary correspondence theorems all state equivalences, so the suffix "`-equiv`" was used in the names of correspondence theorems, even though the top-level theorem, `refutation-p-equiv`, is actually an implication.

```
      (clausep$ clause a$))
 (equal (list-assignment (mv-nth 1 (negate-clause clause a$)))
        (negate-clause-or-assignment clause)))
```

The call of `negate-clause` on the left-hand side of the equality pushes each literal of the clause onto the stack, and then the function `list-assignment` extracts a list-based assignment from the resulting stack. However, the function `negate-clause-or-assignment` (defined for [lrat-3]) simply mapped negation over the clause, for example transforming the clause `(3 -4 5)` to `(-3 4 -5)`—whereas the left-hand side produces `(-5 4 -3)`—reversed! Fortunately, this was the only case in which the list-based and corresponding stobj-based function didn't match up.

By the time this issue surfaced, soundness had been established for the [lrat-3] (list-based) checker, guards had been verified for the [lrat-4] (stobj-based) checker, and some of the equivalence proofs had been completed. So we followed the steps below to modify the [lrat-3] checker to support the remaining equivalence proofs and avoid excessive re-work; after these steps, we completed the remaining correspondence proofs without undue difficulty.

1. We modified [lrat-3] by disabling `negate-clause-or-assignment` and attempting the proofs, expecting them to fail since that definition was no longer available.
2. We fixed the failed proofs—there were only a few—by providing them with hints to re-enable `negate-clause-or-assignment`.
3. We redefined `negate-clause-or-assignment` as a call to a tail-recursive function that reversed the order. Because of the steps above, the only proofs that failed were those explicitly enabling `negate-clause-or-assignment`.
4. With relatively modest effort we fixed all failed proofs.

4.4 The [lrat-5] Proof

Our [lrat-4] and [lrat-5] code were essentially the same except for the highest-level functions. It was thus straightforward to work through the proof in a top-down style, reusing previous lemmas once we worked our way down to reasoning about functions that had not changed.

We improved the soundness theorem. Previous versions simply stated that every formula with a refutation is unsatisfiable. To see why that statement is insufficient, imagine an "evil" parser that always returns the trivial formula, containing only the empty clause. Then when the checker validates a proof, such a soundness theorem will only tell us that the empty clause is unsatisfiable! In principle a solution is to verify the parser, but that seems to us a difficult undertaking.

Instead we define a function, `proved-formula`, which takes two input files and various other parameters (such as how much of the proof file to read at each iteration). When a proof is successfully checked, this function returns the formula proved—essentially, what was read from the formula input file. This is the function that we actually run to check proofs. The following theorem states that if `proved-formula` is applied to a given formula file, `cnf-file`, and proof

file, `clrat-file`, and it returns a formula F (rather than `nil`, which represents failure), then F is unsatisfiable.[11]

```
(defthm soundness
  (let ((formula (mv-nth 1 (proved-formula cnf-file clrat-file ...))))
    (implies formula
             (not (satisfiable formula)))))
```

For extra confidence, a very simple program[12], whose correctness can easily be ascertained by inspection, can print to a new file the formula returned by `proved-formula`. We have used this utility to compare the new file to the input formula using the `diff` utility, thereby providing confidence that the unsatisfiable formula returned by `proved-formula` truly represents the contents of the input formula file.

5 Conclusion

We now have an efficient, verified SAT checker that can rapidly check SAT proof files of many gigabytes. We expect that it will be used in applications of SAT solvers that demand validation, both in SAT competitions and in industry. Performance data on hard problems of the recent SAT competition suggest that the ACL2-based [lrat-5] checker generally adds less than 25% to the time spent by unverified proof-checking alone. The soundness proof for the stobj-based checker was split quite nicely into a sequence of proof efforts. Here are some reflections on those efforts, based on checker names introduced in Sect. 3.1.

1. We easily proved the soundness of [drat] by modifying the proof for [rat].
2. We developed the soundness proof for [lrat-1] essentially from scratch, starting with development of a hand proof. We believe that this helped us to deal with proof fallout from changes to the code from [drat] to [lrat-1].
3. We modified the proof for [lrat-1] in a modular way to produce a proof for [lrat-2], which shrinks the formula's fast-alist heuristically to boost performance significantly. This step (and others) benefited from ACL2's automation, in particular its display of key checkpoints upon proof failure. We believe that the structuring of the [lrat-1] soundness proof to follow a hand proof helped us to be efficient, by adding clarity to what we were trying to do.
4. The change from [lrat-2] to [lrat-3] was quite easy. The simplified notion of formula was expected to be useful for the next step, and we believe it was.
5. The change from [lrat-3] to [lrat-4] introduced stobj-based code. It seemed simplest to avoid trying to modify the soundness proof, instead deriving soundness as a corollary of a correspondence theorem that relates those two checkers. That worked out nicely, though it involved modifying a function in [lrat-3]. That modification was done in a modular way, in a succession of

[11] The `mv-nth` expression extracts the returned formula from a multiply-valued result.
[12] `projects/sat/lrat/incremental/print-formula.lisp` in the community books.

steps for which that function was disabled. Guard verification was challenging, but its supporting theorems and techniques helped with the soundness proof. Specifically, patterned-based congruences [13] developed for guard verification were also used in proving correspondence theorems.

6. The change from [lrat-4] to [lrat-5] caused us to extend the ACL2 system (and logical theory) with a utility for reading a portion of a file into a string. This utility supports efficient input from very large proof files.

Our software development approach used a form of refinement. We first specified and verified a very simple, but inefficient SAT proof checker. We then introduced another more efficient, but more complex, SAT proof-checker, that we then verified. We continued this process until we had a solution that was fast enough and verified to be correct. We believe that this stepwise approach was an effective, efficient way to develop a high-performance formally verified SAT proof-checker. This effort adds evidence one can build formally-verified production-class software.

References

1. Blanchette, J.C., Fleury, M., Weidenbach, C.: A verified SAT solver framework with learn, forget, restart, and incrementality. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS, vol. 9706, pp. 25–44. Springer, Cham (2016). doi:10.1007/978-3-319-40229-1_4
2. Boyer, R.S., Moore J S.: Single-threaded objects in ACL2. In: Krishnamurthi, S., Ramakrishnan, C.R. (eds.) PADL 2002. LNCS, vol. 2257, pp. 9–27. Springer, Heidelberg (2002). doi:10.1007/3-540-45587-6_3
3. Cruz-Filipe, L., Heule, M.J.H., Hunt Jr., W.A., Kaufmann, M., Schneider-Kamp, P.: Efficient certified RAT verification. In: de Moura, L. (ed.) CADE 2017. LNAI, vol. 10395, pp. 220–236. Springer, Cham (2017). doi:10.1007/978-3-319-63046-5_14
4. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM **5**(7), 394–397 (1962)
5. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM (JACM) **7**(3), 201–215 (1960)
6. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004). doi:10.1007/978-3-540-24605-3_37
7. Greve, D.A., Kaufmann, M., Manolios, P., Moore J S., Ray, S., Ruiz-Reina, J.L., Sumners, R., Vroon, D., Wilding, M.: Efficient execution in an automated reasoning environment. J. Funct. Program. **18**(1), 15–46 (2008)
8. Heule, M.J.H., Biere, A.: Clausal proof compression. In: 11th International Workshop on the Implementation of Logics. EPiC Series in Computing, vol. 40, pp. 21–26 (2016)
9. Heule, M.J.H., Hunt Jr., W.A., Wetzler, N.D.: Trimming while checking clausal proofs. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, 20–23 October 2013, pp. 181–188 (2013)
10. Heule, M.J.H., Hunt Jr., W.A., Wetzler, N.D.: Verifying refutations with extended resolution. In: Bonacina, M.P. (ed.) CADE 2013. LNCS (LNAI), vol. 7898, pp. 345–359. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38574-2_24

11. Järvisalo, M., Heule, M.J.H., Biere, A.: Inprocessing rules. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS (LNAI), vol. 7364, pp. 355–370. Springer, Heidelberg (2012). doi:10.1007/978-3-642-31365-3_28
12. Kaufmann, M., Manolios, P., Moore J S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Press, Boston (2000)
13. Kaufmann, M., Moore J S.: Rough diamond: an extension of equivalence-based rewriting. In: Klein, G., Gamboa, R. (eds.) ITP 2014. LNAI, vol. 8558, pp. 537–542. Springer, Cham (2014). doi:10.1007/978-3-319-08970-6_35
14. Kaufmann, M.: Modular proof: the fundamental theorem of calculus. In: Kaufmann, M., Manolios, P., Moore J S. (eds.) Computer-Aided Reasoning: ACL2 Case Studies. Advances in Formal Methods, vol. 4, pp. 75–91. Springer, Boston (2000). doi:10.1007/978-1-4757-3188-0_6
15. Kaufmann, M., Moore J S.: ACL2 home page. http://www.cs.utexas.edu/users/moore/acl2. Accessed 2016
16. Lammich, P.: Efficient verified (UN)SAT certificate checking. In: de Moura, L. (ed.) CADE 2017. LNAI, vol. 10395, pp. 237–254. Springer, Cham (2017). doi:10.1007/978-3-319-63046-5_15
17. 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)
18. Marić, F.: Formalization and implementation of modern SAT solvers. J. Autom. Reason. **43**(1), 81–119 (2009)
19. Marić, F.: Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. Theor. Comput. Sci. **411**(50), 4333–4356 (2010)
20. Marques-Silva, J.P., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, chap. 4, pp. 131–153. IOS Press, Amsterdam (2009)
21. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine (part I). CACM **3**(4), 184–195 (1960)
22. Oe, D., Stump, A., Oliver, C., Clancy, K.: versat: a verified modern SAT solver. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 363–378. Springer, Heidelberg (2012). doi:10.1007/978-3-642-27940-9_24
23. Shankar, N., Vaucher, M.: The mechanical verification of a DPLL-based satisfiability solver. Electron. Notes Theor. Comput. Sci. **269**, 3–17 (2011)
24. Steele Jr., G.L.: Common Lisp the Language, 2nd edn. Digital Press, Burlington (1990)
25. Swords, S.: Private communication, March/April 2017
26. Wetzler, N.D., Heule, M.J.H., Hunt Jr., W.A.: Mechanical verification of SAT refutations with extended resolution. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 229–244. Springer, Heidelberg (2013). doi:10.1007/978-3-642-39634-2_18
27. Wetzler, N.D., Heule, M.J.H., Hunt Jr., W.A.: DRAT-trim: efficient checking and trimming using expressive clausal proofs. In: Sinz, C., Egly, U. (eds.) SAT 2014. LNCS, vol. 8561, pp. 422–429. Springer, Cham (2014). doi:10.1007/978-3-319-09284-3_31
28. Wilding, M.: Design goals for ACL2. Tech. Rep. CLI Technical Report 101, Computational Logic, Inc., August 1994. https://www.cs.utexas.edu/users/moore/publications/km94.pdf