Development of a Verified, Efficient Checker for SAT Proofs

Matt Kaufmann
(In collaboration with Marijn Heule and Warren Hunt, Jr.)

ACL2 Seminar
The University of Texas at Austin

February 3, 2017
(revised 2/4/2017)
I’ll present a case study, consisting of a sequence of verified checkers that validate SAT proofs. These culminate in an efficient checker that can be used in SAT competitions and in industry. No background in SAT is assumed.
OUTLINE

INTRODUCTION
The Problem
Towards a Solution
Variables, Literals, Clauses, Formulas
Semantics: Assignments and Truth
Proofs
Formalizing Soundness
Efficient Proof-checking

A SEQUENCE OF CHECKERS
[drat]
The LRAT Proof Format
[lrat-1]
[lrat-2]
[lrat-3]
[lrat-4]

CONCLUSION
REFERENCES
# Outline

## Introduction

- The Problem
- Towards a Solution
- Variables, Literals, Clauses, Formulas
- Semantics: Assignments and Truth
- Proofs
- Formalizing Soundness
- Efficient Proof-checking

## A Sequence of Checkers

- [drat]
- The LRAT Proof Format
- [lrat-1]
- [lrat-2]
- [lrat-3]
- [lrat-4]

## Conclusion

## References
Questions welcome during the talk, feedback afterwards.

Feel free to slow me down (will move quickly through early stuff that is probably familiar to all).

Brief summary of talk:

- Nathan Wetzler wrote and verified an ACL2 program that validates SAT proofs.
- This talk discusses development of an efficient such verified checker.

Underlining denotes links to the [ACL2+books online manual](#).
THE PROBLEM

Boolean Satisfiability (SAT) solvers are proliferating and useful. But how can we trust them?

Modern ones [3] emit proofs!

But how do we know that these “proofs” are valid?

We check them with software programs called checkers!

But how do we know that a checker is sound? Inspection?

- Checkers are typically simpler than solvers...
- ... but not that simple, and inspection is error-prone.
Towards a Solution

Nathan Wetzler, under the direction of Marijn Heule and Warren Hunt, developed an ACL2-based solution [6, 5, 4].

He wrote a SAT proof-checker in ACL2, then formalized and proved its correctness (soundness):

**Suppose** the checker takes inputs $\Pi$ (an alleged proof) and $F$ (a formula), and checks that $\Pi$ legally derives a contradiction from $F$.

**Then** $F$ is always false.

Background:
Variables, Literals, Clauses, Formulas

A variable is a propositional atom.

- Traditionally, a Boolean formula might be $P_1 \land \neg P_2$, where $P_1$ and $P_2$ are symbols known as variables.
- For us, variables are positive integers.

A literal is a variable or its negative (negation), e.g., 3 or -3.

Complementary literals are negations of each other.

A clause is a set of literals, implicitly disjoined, containing no complementary literals.

- In ACL2: duplicate-free lists of non-zero integers without complementary literals. Example: $(3 \ 7 \ -2 \ 4)$.

A formula is a set (or list) of clauses, implicitly conjoined. (This is commonly called conjunctive normal form.)
**SEMANTICS: ASSIGNMENTS AND TRUTH**

An *assignment* is a finite function mapping variables to Booleans.

- In ACL2: same representation as for clauses, e.g.,
  
  \[(3\ 7\ -2\ 4)\]  
  \[but\ not\ \(3\ 7\ -3)\].

*Truth value under an assignment*: recursively defined for literals, then clauses, then formulas, to be \(T\), \(NIL\), or 0 (unknown).

**Example:** Is \(F\) true under assignment \(a\)?

\(F\): \(\left(\begin{array}{c} 
1
7
-2
\end{array}\right)\) \(\left(\begin{array}{c}
-3
6
5
\end{array}\right)\) \(\left(\begin{array}{c}
9
2
3
\end{array}\right)\)

\(a\): \((7\ -3)\)

**Answer:** No — the truth value is 0 because of the third clause.

A formula is *satisfiable* if it is true under some assignment; otherwise, it is *unsatisfiable*.
A proof (or clausal proof, or refutation) for a formula $F$ is a sequence $\Pi = \langle p_1, p_2, \ldots, p_k \rangle$ such that:

- Each $p_i$ is $\langle b_i, c_i \rangle$, where $b_i$ is a Boolean and $c_i$ is a clause.
  - Deletion step: $b_i$ is true
  - Addition step: $b_i$ is false

- $b_k$ is false and $c_k$ is the empty clause.

- All addition steps preserve satisfiability (see next slide).
For $\Pi = \langle p_1, p_2, \ldots, p_k \rangle$ as above, recursively define formulas $\langle F_0, F_1, \ldots, F_k \rangle$ by executing the $p_i$:

- $F_0 = F$.
- For $i > 0$ and $b_i$ true, delete $c_i$ from $F_{i-1}$ to get $F_i$.
- For $i > 0$ and $b_i$ false, add $c_i$ to $F_{i-1}$ to get $F_i$.

Then $\Pi$ preserves satisfiability when for each addition step $p_i$, if $F_{i-1}$ is satisfiable then $F_i$ is satisfiable.
NOTE: The definition above of clausal proof is very general. A checker may impose more specific syntactic requirements that guarantee the property.

The next slide shows Nathan’s formalization, for ITP 2013, based on the RAT (Resolution Asymmetric Tautology) check. Details on RAT are not the subject of today’s talk.

All checkers discussed today use a formalization like the one on the next slide, based on RAT.
FORMALIZING SOUNDNESS

Below, \texttt{proofp} is a recognizer for proofs, and \texttt{solutionp} checks that a formula is true under a given assignment,

\begin{verbatim}
(defun refutationp (proof formula)
  (declare (xargs :guard (formulap formula)))
  (and (proofp proof formula)
       (member *empty-clause* proof)))

(defun-sk exists-solution (formula)
  (exists assignment
   (solutionp assignment formula)))

(defthm main-theorem
  (implies (and (formulap formula)
                 (refutationp clause-list formula))
           (not (exists-solution formula))))
\end{verbatim}
FORMALIZING SOUNDNESS (2)

The following is easily proved by induction.

**Lemma.** Suppose that $\Pi = \langle p_1, p_2, \ldots, p_k \rangle$ is a proof and $F_0$ is satisfiable. Then each $F_i$ is satisfiable.

Soundness argument:

1. Deletion steps clearly preserve satisfiability.
2. **Addition steps preserve satisfiability.** [Must be proved!]
3. By the lemma, if $F_0$ is satisfiable then $F_k$ is satisfiable.
4. Since $p_k$ adds the empty clause, $F_k$ is unsatisfiable.
5. It follows immediately that $F_0$ is unsatisfiable.
EFFICIENT PROOF-CHECKING

HOWEVER: Nathan’s ITP 2013 checker, discussed above, was intended to be a proof of concept, not an efficient tool. (I have since learned of a later checker in Nathan’s dissertation that, like our [lrat-4] discussed below, used stobjs for assignments. It did not use deletion or fast-alists, or claim efficiency. This talk does not mention that checker further.)

On one example:

- Marijn’s checker: 1.5 seconds
- Nathan’s (ITP 2013) checker: 1 week

Marijn’s request: a formally verified checker for SAT competitions
This talk tells the (true) story of the development of such a checker.

- Its efficiency benefits in part from some techniques not yet invented at the time of Nathan’s work.
The flow for **efficient**, verified SAT proof-checking:

1. SAT solver verifies unsatisfiability of formula $F$; generates alleged proof, $\Pi_0$.

2. *DRAT-trim* [2] takes inputs $\Pi_0$ and $F$; outputs alleged proof $\Pi_1$ for checker, **in a format amenable to efficient checking**.

3. Verified ACL2 checker validates that $\Pi_1$ is a proof for $F$. 
OUTLINE

INTRODUCTION
  The Problem
  Towards a Solution
  Variables, Literals, Clauses, Formulas
  Semantics: Assignments and Truth
  Proofs
  Formalizing Soundness
  Efficient Proof-checking

A SEQUENCE OF CHECKERS
  [drat]
  The LRAT Proof Format
  [lrat-1]
  [lrat-2]
  [lrat-3]
  [lrat-4]

CONCLUSION

REFERENCES
A Sequence of Checkers

This table shows times (in seconds) for some checker runs, on examples provided by Marijn.

<table>
<thead>
<tr>
<th>test</th>
<th>[rat] (Wetzler)</th>
<th>[drat] (deletion)</th>
<th>[lrat-1] (fast-alist)</th>
<th>[lrat-2] (shrink)</th>
<th>[lrat-3] (clean up)</th>
<th>[lrat-4] (stobjs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>uuf-100-3</td>
<td>20.64</td>
<td>8.59</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.00</td>
</tr>
<tr>
<td>tph6[-dd]</td>
<td>-</td>
<td>-</td>
<td>6.18</td>
<td>0.56</td>
<td>0.54</td>
<td>0.46</td>
</tr>
<tr>
<td>R_4_4_18</td>
<td>∼1 week</td>
<td>-</td>
<td>217.91</td>
<td>9.62</td>
<td>3.21</td>
<td>2.56</td>
</tr>
<tr>
<td>transform</td>
<td>-</td>
<td>-</td>
<td>47.80</td>
<td>9.59</td>
<td>8.82</td>
<td>8.77</td>
</tr>
<tr>
<td>schur</td>
<td>-</td>
<td>-</td>
<td>4674.18</td>
<td>1872.07</td>
<td>1884.23</td>
<td>246.94</td>
</tr>
</tbody>
</table>

Times do not include parsing. Warren Hunt has sped up our original parser, and there are plans to speed it up further by using a binary proof format (not discussed further here).
A Sequence of Checkers (2)

How this work progressed (will elaborate on the next slides).

1. [rat] Nathan’s ITP 2013 RAT checker: no deletion
2. [drat] Added deletion (thus implementing DRAT)
3. [lrat-1] Avoid search and delete clauses efficiently, using fast-alists (applicative hash tables) and a linear proof format, and with soundness proved from scratch
4. [lrat-2] Shrink fast-alists to keep the formulas $F_i$ small
5. [lrat-3] Minor tweak to formula data-structure
6. [lrat-4] Added stobjs for assignments
Acknowledgments:

- Marijn helped a lot with getting us up to speed on SAT proof-checking based on RAT, and by supplying examples.
- Warren worked with me in the initial stages.
- Of course, many ACL2 features were crucial, including **proof procedures** (many inherited from earlier Boyer-Moore provers) and **fast-alist**s (initially Boyer/Hunt, later Davis/Swords).

Profiling (Marijn’s suggestion) helped with discovering bottlenecks:

```
(include-book "centaur/memoize/old/profile"
   :dir :system)
(profile-all) ; or just profile specific functions
<evaluate forms>
(memsum)
```
This project illustrates the interplay between ACL2 as a programming language and as a theorem prover:

- Optimize the program for efficiency.
- Deal with proving correctness for the optimizations.
Incorporating deletion was straightforward.

- In [rat], a proof is a list of clauses to be added (no deletion).
- A [drat] proof is a list of pairs \( \langle b, c \rangle \) — in ACL2, \((b \ . \ . \ c)\), where \(b\) is a Boolean deletion flag and \(c\) is a clause.
- Warren and I easily modified Nathan’s ITP 2013 proof.

Deletion should help with speed by keeping the formulas \(F_i\) small.

But the [drat] checker is still slow. Why?
[drat]: Why It’s Slow

- *Unit propagation* (UP) results in *many* linear searches through $F_i$.
- Deletion does a linear search and *much* consing.


THE LRAT PROOF FORMAT

Marijn, with others, has developed a Linear RAT (LRAT) proof format.[1]

▶ “Others” includes 2 Coq users who have also developed a verified SAT proof-checker.
▶ Theirs takes 10 minutes on one example compared to our 9 seconds.

Example LRAT proof step $p_i$:

820  -59  -17  -58  0  807  246  423  40  -87  308  117  819  809  404  310  -163  -313  0

The next slide breaks this line apart.
The clause to be added has index 820:

820

It is the set of literals, \{−59 \ −17 \ −58\}:

\[−59 \ −17 \ −58\]

Separator:

0

Apply unit propagation (UP) to these four clauses, in order:

807 246 423 40

For the RAT check on clause 87, restrict UP to the clauses 308, 117, ..., and 310, in order.

For the RAT check on clause 163, no UP is performed.

For the RAT check on clause 313, no UP is performed.

\[-87 \ 308 \ 117 \ 819 \ 809 \ 404 \ 310 \ −163 \ −313\]

End of proof step:

0
Hints direct exactly where unit propagation is done – no search! This addresses the first of the two “Why It’s Slow” problems. Again:

- *Unit propagation* (UP) results in many linear searches through $F_i$.

- Deletion does a linear search and much consing.

Clause indices help solve the second problem.

The next checker implements these efficiencies.
Proof steps represent the LRAT format.

A formula represents a list of pairs \((i \ . \ c)\) where \(i\) is a natural number, the index of clause \(c\).

- This list is a fast-alist: ACL2 uses a hash-table to find \(c\) from \(i\) in essentially constant time.

- A formula is a pair \((\text{max} \ . \ \text{fal})\), where \(\text{fal}\) is its fast-alist and \(\text{max}\) is an upper bound on its indices.
How do fast-alists help with efficiency?

- Unit propagation benefits from fast lookup to obtain a clause from its index; and

- Deletion of clause $i$ simply extends the fast-alist with a pair $(i . \ *\text{deleted-clause}*)$.
  
  - The value of $\*\text{deleted-clause\*}$ is a non-nil atom, hence not a clause.
Soundness Proof Problem:
How to manage the substantial change from [drat] to [lrat-1].

▶ Painful to rework another’s proof
▶ Decision: Sketch hand proof and manage a fresh proof
▶ Used top-down approach (see my 1999 ACL2 Workshop paper)
satisfiable-add-proof-clause.lisp

<hand proof in comment>
(in-package "ACL2")
(include-book "lrat-checker")

(local (encapsulate ()
  (local (include-book "satisfiable-add-proof-clause-rup"))
  (local (include-book "satisfiable-add-proof-clause-drat"))
  (set-enforce-redundancy t)
  (defthm satisfiable-add-proof-clause-rup 
    ...) 
  (defthm satisfiable-add-proof-clause-drat 
    ...) )
)

(defthm satisfiable-add-proof-clause 
  ...
  :hints
  (("Goal" :use (satisfiable-add-proof-clause-rup 
    satisfiable-add-proof-clause-drat) 
  :in-theory (union-theories '(verify-clause) 
    (theory 'minimal-theory))))
Profiling showed 69% of the time inside `hons-get` (looking up clause indices).

The RAT check visits *every* clause in the formula $F_i$.

The [lrat-2] checker improves on [lrat-1] in two ways:

- Shrink the formula’s fast-alist when heuristics say to do so.
- RAT check recurs through the fast-alist instead of recurring down from the max index.
[lrat-2]: SHRINKING

Two counts maintained on the formula:

- \( ndel \): number of pairs \((i \cdot \text{deleted-clause})\)
- \( ncls \): the number of pairs \((i \cdot c)\) representing clauses that have not been deleted

Heuristically shrink the fast-alist at an addition proof step, based on experimentation:

- whenever \( ndel > 10 \times ncls \);
- when RAT check is necessary, shrink first if \( ndel > 1/3 \times ncls \).
To shrink a fast-alist (will discuss only if time):

```lisp
(defun remove-deleted-clauses (fal acc)
  (declare (xargs :guard (alistp fal)))
  (cond ((endp fal) (make-fast-alist acc))
        (t (remove-deleted-clauses
            (cdr fal)
            (if (deleted-clause-p (cdar fal))
                acc
                (cons (car fal) acc))))))

(defun shrink-formula-fal (fal)
  (declare (xargs :guard (formula-fal-p fal)))
  (let ((fal2 (fast-alist-clean fal)))
    (fast-alist-free-on-exit
      fal2
      (remove-deleted-clauses fal2 nil))))
```
[lrat-2]: PROOF

Proved soundness by tweaking the [lrat-1] proof:

- Disabled the top-level “maybe shrink” function
- Re-ran the [lrat-1] proof on [lrat-2]
- Looked at key checkpoints on failure to determine lemmas to prove (about shrinking).
[lrat-3]

Changed formula from $\text{max . fal}$ to simply $\text{fal}$.

- $\text{Max}$ was only used for RAT check recursion, but [lrat-2] recurs through $\text{fal}$.
- This simplification seemed useful before starting the next checker, and it saves consing.
- Soundness proof for [lrat-2] was easy to modify for [lrat-3].
A bottleneck in [lrat-3]: evaluation of a literal $n$ requires a linear-time search for either $n$ or $-n$ in the assignment.

[lrat-4] solution: use single-threaded objects (stobjs) to model assignments.

- Lookup is a constant-time array reference.
- Avoids memory allocation (consing) when pushing new literals onto assignment.
[lrat-4]: ASSIGNMENTS

(defstobj a$
  (a$ptr :type (integer 0 *)) ; stack pointer
    :initially 0)
(a$stk :type (array t (1))) ; stack of a$arr indices
    :resizable t)
(a$arr :type (array t (1))) ; array of 0, t, nil
    :initially 0
    :resizable t)
 :renaming ((a$arrp a$arrp-weak)
          (a$p a$p-weak)))
[lrat-4]: ASSIGNMENTS (2)

Operations on assignments:

- \((\text{push-literal } \text{lit } a\$)\) extends assignment \(a\$\) with literal \(\text{lit}\) (writes to \(a\$\text{stk}\), increments \(a\$\text{ptr}\)).

- \((\text{pop-literals } \text{ptr } a\$)\) updates \(a\$\text{ptr}\) to \(\text{ptr}\).

**KEY OBSERVATION:** These operations generate calls to \(\text{nth}\) and \(\text{update-nth}\), but for [lrat-3], they are implemented with \(\text{cons}\) and \(\text{cdr}\).

Tweaking the [lrat-3] proof seemed difficult! Instead....
[lrat-4]: PROOF


- Then I derived the soundness of [lrat-4] directly from those correspondence theorems and the soundness of [lrat-3].
(defthm main-theorem-list-based
  (implies (and (formula-p formula)
               (refutation-p proof formula))
           (not (satisfiable formula)))
  :hints ...)

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

(defthm main-theorem-stobj-based
  (implies (and (formula-p formula)
               (refutation-p$ proof formula))
           (not (satisfiable formula)))
  :hints ("Goal"
          :in-theory '(refutation-p-equiv
                      :use main-theorem-list-based)))
All of these checkers are guard-verified, for runtime efficiency. For that, needed invariant on the stobj that is preserved when a function returns a modified stobj.

1. Developed that invariant, (a$p a$)

2. Verified guards (perhaps easier than correspondence theorems), which required invariance proofs

3. Proved correspondence theorems
[lrat-4]: PROOF (4)

I’ll very briefly discuss the invariant:

(defun a$p (a$)
  (declare (xargs :stobjs a$))
  (and (a$p-weak a$)
       (<= (a$ptr a$) (a$stk-length a$))
       (equal (a$arr-length a$)
               (1+ (a$stk-length a$)))
       (good-stk-p (a$ptr a$) a$)
       (a$arrp a$)
       (arr-matches-stk (a$arr-length a$) a$)))
[lrat-4]: PROOF (5)

A challenge: The correspondence proofs broke down!


- One [lrat-3] function, `negate-clause-or-assignment`, did not match up with its corresponding [lrat-4] function.

The [lrat-2] function (originally used in [lrat-3]):

```lisp
(defun negate-clause-or-assignment (clause)
  (declare (xargs :guard (clause-or-assignment-p clause)))
  (if (atom clause)
      nil
      (cons (negate (car clause))
            (negate-clause-or-assignment (cdr clause))))
)```
[lrat-4]: PROOF (6)

What to do? Status when problem was discovered:

- Soundness for [lrat-3] was already established
- Guards for [lrat-4] were already verified.
- Some equivalence proofs were complete.

Solution: Modified [lrat-3] by changing the definition of function `negate-clause-or-assignment` and fixing failed proofs.

Then completed correspondence theorems, which yielded soundness for [lrat-4].
OUTLINE

INTRODUCTION
The Problem
Towards a Solution
Variables, Literals, Clauses, Formulas
Semantics: Assignments and Truth
Proofs
Formalizing Soundness
Efficient Proof-checking

A SEQUENCE OF CHECKERS
[drat]
The LRAT Proof Format
[lrat-1]
[lrat-2]
[lrat-3]
[lrat-4]

CONCLUSION

REFERENCES
CONCLUSION

There is now an **efficient formally verified SAT checker**!

- On a large example, its time of 4.1 minutes (without parsing) compares very favorably with DRAT-trim time of 20 minutes (with very fast C parsing).

- Warren is working on a faster parser (it takes about 20 minutes with mine, which is based on `read-object`).

Checkers **[lrat-3]** and **[lrat-4]** are in the community books in these directories, respectively.

`projects/sat/lrat/list-based/`
`projects/sat/lrat/stobj-based/`

Other checkers are available via links from the seminar page.
This work can be found in the community books, with the latest version on github:

books/projects/sat/lrat/

Nathan Wetzler’s ITP 2013 and dissertation checkers can also be found in the community books:

books/projects/sat/proof-checker-itp13/
books/projects/sat/proof-checker-array/

The next slide has references for citations in this talk.


