Development of a Verified, Efficient Checker for SAT Proofs

Matt Kaufmann
(With contributions from Marijn Heule, Warren Hunt, and Nathan Wetzler)

The University of Texas at Austin

ACL2 Workshop 2017

May 22, 2017

OVERVIEW

Boolean Satisfiability (SAT) solvers are proliferating and useful.

OVERVIEW

Boolean Satisfiability (SAT) solvers are proliferating and useful.

PROBLEM: How can we trust their claims of unsatisfiability?

Overview

Boolean Satisfiability (SAT) solvers are proliferating and useful.

PROBLEM: How can we trust their claims of unsatisfiability?

SOLUTION:

OVERVIEW

Boolean Satisfiability (SAT) solvers are proliferating and useful.

PROBLEM: How can we trust their claims of unsatisfiability?

SOLUTION:

- SAT Solver emits a proof, p_0

OVERVIEW

Boolean Satisfiability (SAT) solvers are proliferating and useful.

PROBLEM: How can we trust their claims of unsatisfiability?

SOLUTION:

- SAT Solver emits a proof, p_0
- DRAT-trim (from Marijn Heule) processes p_0, creating smaller proof p_1 that includes hints

OVERVIEW

Boolean Satisfiability (SAT) solvers are proliferating and useful.

PROBLEM: How can we trust their claims of unsatisfiability?

SOLUTION:

- SAT Solver emits a proof, p_0
- DRAT-trim (from Marijn Heule) processes p_0, creating smaller proof p_1 that includes hints
- Verified ACL2 program checks p_1

OVERVIEW

Boolean Satisfiability (SAT) solvers are proliferating and useful.

PROBLEM: How can we trust their claims of unsatisfiability?

SOLUTION:

- SAT Solver emits a proof, p_0
- DRAT-trim (from Marijn Heule) processes p_0, creating smaller proof p_1 that includes hints
- Verified ACL2 program checks p_1

This talk is high-level, avoiding details such as "RAT" and "DRAT".

Outline

Outline

The Problem

Boolean Satisfiability (SAT) solvers are proliferating and useful.

THE PROBLEM

Boolean Satisfiability (SAT) solvers are proliferating and useful.

- They verify unsatisfiability of a Boolean *formula*, represented as a list of *clauses* (each a disjunction of *literals*).

THE PROBLEM

Boolean Satisfiability (SAT) solvers are proliferating and useful.

- They verify unsatisfiability of a Boolean *formula*, represented as a list of *clauses* (each a disjunction of *literals*).
- Example of unsatisfiable formula:

```
(
 (1 2 -3) ; 1 OR 2 OR (not 3)
 (-1)     ; (not 1)
 (-2 -3)  ; (not 2) OR (not 3)
 (3)      ; 3
)
```

THE PROBLEM

Boolean Satisfiability (SAT) solvers are proliferating and useful.

- They verify unsatisfiability of a Boolean *formula*, represented as a list of *clauses* (each a disjunction of *literals*).
- Example of unsatisfiable formula:

```
(
 (1 2 -3) ; 1 OR 2 OR (not 3)
 (-1)     ; (not 1)
 (-2 -3)  ; (not 2) OR (not 3)
 (3)      ; 3
)
```

But how can we *trust* SAT solvers?

OUTLINE

Towards a Solution (1)

Modern SAT solvers [2] emit *proofs*!

TOWARDS A SOLUTION (1)

Modern SAT solvers [2] emit *proofs*!

- Proof step: **Add a clause** to the formula (conjunction of clauses) that **preserves satisfiability**.

Towards a Solution (1)

Modern SAT solvers [2] emit *proofs*!

- Proof step: **Add a clause** to the formula (conjunction of clauses) that **preserves satisfiability**.
  - Eventually add the empty clause.
  - So final formula is unsatisfiable.
  - So input formula must be unsatisfiable!

Towards a Solution (1)

Modern SAT solvers [2] emit *proofs*!

- Proof step: **Add a clause** to the formula (conjunction of clauses) that **preserves satisfiability**.
  - Eventually add the empty clause.
  - So final formula is unsatisfiable.
  - So input formula must be unsatisfiable!
- Also legal: proof steps that **delete a clause** from the formula.
  - Clearly preserves satisfiability.

TOWARDS A SOLUTION (2)

But how do we know that these "proofs" are valid?

Towards a Solution (2)

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

We check them with software programs called *checkers*!

Towards a Solution (2)

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?

Towards a Solution (2)

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?

- Key property: clause addition preserves satisfiability

Towards a Solution (2)

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?

- Key property: clause addition preserves satisfiability
- Checkers (e.g., DRAT-trim) are typically simpler than solvers...

Towards a Solution (2)

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?

- Key property: clause addition preserves satisfiability
- Checkers (e.g., DRAT-trim) are typically simpler than solvers...
  - ... but not *that* simple, and *inspection is error-prone*.

TOWARDS A SOLUTION (3)

Wetzler proved soundness of an ACL2-based solution [6, 5, 4].

TOWARDS A SOLUTION (3)

Wetzler proved soundness of an ACL2-based solution [6, 5, 4].
I'll explain our "**[lrat-4]**" and "**[lrat-5]**" versions of soundness:

Towards a Solution (3)

Wetzler proved soundness of an ACL2-based solution [6, 5, 4].
I'll explain our "**[lrat-4]**" and "**[lrat-5]**" versions of soundness:

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

TOWARDS A SOLUTION (3)

Wetzler proved soundness of an ACL2-based solution [6, 5, 4].
I'll explain our "**[lrat-4]**" and "**[lrat-5]**" versions of soundness:

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

(let ((formula
       (mv-nth 1 (proved-formula cnf-file clrat-file
                                 chunk-size debug
                                 nil ; incomplete-okp
                                 ctx state))))
  (implies formula
           (not (satisfiable formula))))
```

TOWARDS A SOLUTION (3)

Wetzler proved soundness of an ACL2-based solution [6, 5, 4].
I'll explain our "**[lrat-4]**" and "**[lrat-5]**" versions of soundness:

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

```
(let ((formula
       (mv-nth 1 (proved-formula cnf-file clrat-file
                                 chunk-size debug
                                 nil ; incomplete-okp
                                 ctx state))))
  (implies formula
           (not (satisfiable formula))))
```

; Print proved formula, to `diff` *against input formula:*

```
(defmacro print-formula (formula &optional filename)
  ...)
```

Towards a Solution (4)

Problem: Efficiency.

TOWARDS A SOLUTION (4)

Problem: Efficiency.

On one example:

- DRAT-trim: 1.5 seconds
- Verified checker [5]: ~ 1 week

TOWARDS A SOLUTION (4)

Problem: Efficiency.

On one example:

- DRAT-trim: 1.5 seconds
- Verified checker [5]: ~ 1 week

NOTE:

- Wetzler's ITP 2013 checker [5] was intended to be a proof of concept, not an efficient tool.
- He did some preliminary work towards increasing efficiency (no timings reported).

Outline

A SEQUENCE OF CHECKERS (1)

A SEQUENCE OF CHECKERS (1)

1. **[rat]** Nathan's ITP 2013 RAT checker [5]: no deletion

A Sequence of Checkers (1)

1. **[rat]** Nathan's ITP 2013 RAT checker [5]: no deletion
2. **[drat]** Support deletion (thus implementing DRAT)

A SEQUENCE OF CHECKERS (1)

1. **[rat]** Nathan's ITP 2013 RAT checker [5]: no deletion
2. **[drat]** Support 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

A SEQUENCE OF CHECKERS (1)

1. **[rat]** Nathan's ITP 2013 RAT checker [5]: no deletion
2. **[drat]** Support 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 formulas small

A SEQUENCE OF CHECKERS (1)

1. **[rat]** Nathan's ITP 2013 RAT checker [5]: no deletion
2. **[drat]** Support 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 formulas small
5. **[lrat-3]** Minor tweak to formula data-structure

A Sequence of Checkers (1)

1. **[rat]** Nathan's ITP 2013 RAT checker [5]: no deletion
2. **[drat]** Support 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 formulas small
5. **[lrat-3]** Minor tweak to formula data-structure
6. **[lrat-4]** Use **stobjs** for assignments

A SEQUENCE OF CHECKERS (1)

1. **[rat]** Nathan's ITP 2013 RAT checker [5]: no deletion
2. **[drat]** Support 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 formulas small
5. **[lrat-3]** Minor tweak to formula data-structure
6. **[lrat-4]** Use **stobjs** for assignments
7. **[lrat-5]** Support incremental file reading using improved `read-file-into-string`; verify improved soundness theorem

A SEQUENCE OF CHECKERS (2)

This table shows times (in seconds) for some checker runs (including parsing), on examples provided by Marijn Heule. Test "R_4_4_18" is the one that took a week with Wetzler's ITP 2013 checker.

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

A Sequence of Checkers (2)

This table shows times (in seconds) for some checker runs (including parsing), on examples provided by Marijn Heule. Test "R_4_4_18" is the one that took a week with Wetzler's ITP 2013 checker.

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

NOTE: For the last (Schur) example: 4.3 minutes for checker adds little to the DRAT-trim time of 20 minutes.

A Sequence of Checkers (3)

This project illustrates the interplay between ACL2 as a programming language and as a theorem prover:

A SEQUENCE OF CHECKERS (3)

This project illustrates the interplay between ACL2 as a programming language and as a theorem prover:

- Optimize the program for efficiency.

A SEQUENCE OF CHECKERS (3)

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.

A SEQUENCE OF CHECKERS (3)

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.

Profiling was very useful.

A SEQUENCE OF CHECKERS (3)

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.

Profiling was very useful.

Plan: Our **[lrat-5]** checker will be used in the 2017 SAT competition.

A SEQUENCE OF CHECKERS (3)

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.

Profiling was very useful.

Plan: Our **[lrat-5]** checker will be used in the 2017 SAT competition.

Time comparison on a set of examples (courtesy of Marijn Heule and J Moore):

```
DRAT-trim          210223 seconds
[lrat-5] checker    20811 seconds
```

Outline

Related Work

- [1] The *Linear RAT* (LRAT) proof format and its use in our ACL2 checker, as well as a corresponding Coq-based checker (which takes 10 minutes on one example compared to our 9 seconds)

Related Work

- [1] The *Linear RAT* (LRAT) proof format and its use in our ACL2 checker, as well as a corresponding Coq-based checker (which takes 10 minutes on one example compared to our 9 seconds)
- [3] An Isabelle development using a refinement framework that (independently of our work) produces an efficient verified checker

OUTLINE

Conclusion

There is now an efficient formally verified SAT checker!

CONCLUSION

There is now an efficient formally verified SAT checker!

- On a large example, its time of 4.3 minutes (including parsing) adds relatively little to the DRAT-trim time of 20 minutes.

CONCLUSION

There is now an efficient formally verified SAT checker!

- On a large example, its time of 4.3 minutes (including parsing) adds relatively little to the DRAT-trim time of 20 minutes.

These checkers are available in the community books under `books/projects/sat/lrat/`:

[rat] `projects/sat/proof-checker-itp13/`

[drat] `projects/sat/lrat/early/drat/`

[lrat-1] `projects/sat/lrat/early/rev1/`

[lrat-2] `projects/sat/lrat/early/rev2/`

[lrat-3] `projects/sat/lrat/list-based/`

[lrat-4] `projects/sat/lrat/stobj-based/`

[lrat-5] `projects/sat/lrat/incremental/`

OUTLINE

REFERENCES

A much more detailed (but somewhat outdated – no mention of [lrat-5]) version of this talk is available on the ACL2 seminar website.

REFERENCES

A much more detailed (but somewhat outdated – no mention of [lrat-5]) version of this talk is available on the ACL2 seminar website.

A preprint of a paper on this work (with Heule, Hunt, and Wetzler) is at:
`http://www.cs.utexas.edu/users/kaufmann/papers/lrat-preprint/index.html`.

REFERENCES

A much more detailed (but somewhat outdated – no mention of [lrat-5]) version of this talk is available on the ACL2 seminar website.

A preprint of a paper on this work (with Heule, Hunt, and Wetzler) is at:
`http://www.cs.utexas.edu/users/kaufmann/papers/lrat-preprint/index.html`.

The final slide has references for citations in this talk.

REFERENCES

A much more detailed (but somewhat outdated – no mention of [lrat-5]) version of this talk is available on the ACL2 seminar website.

A preprint of a paper on this work (with Heule, Hunt, and Wetzler) is at:
`http://www.cs.utexas.edu/users/kaufmann/papers/lrat-preprint/index.html`.

The final slide has references for citations in this talk.

Thank you for your attention!

[1] Luís Cruz-Filipe, Marijn Heule, Warren Hunt, Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In *CADE 2017*. To appear.

[2] Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Verifying refutations with extended resolution. In Maria Paola Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *LNCS*, pages 345–359. Springer, 2013.

[3] Peter Lammich. Efficient verified (UN)SAT certificate checking. In *CADE 2017*. To appear, 2017.

[4] Nathan Wetzler. Supplemental material for a paper appearing in interactive theorem proving 2013 [RAT proof-checker]. https://github.com/acl2/acl2/tree/master/books/projects/sat/proof-checker-itp13/, Accessed: December 2016.

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

[6] Nathan David Wetzler. *Efficient, Mechanically-Verified Validation of Satisability Solvers*. PhD thesis, University of Texas at Austin, 2015.