Certifying SAT Proofs

Marijn J.H. Heule, Warren A. Hunt, Jr., and Matt Kaufmann

THE UNIVERSITY OF TEXAS AT AUSTIN

High Confidence Software and Systems Conference, May 8, 2017

Introduction

Satisfiability (SAT) Solving Has Many Applications

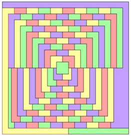
formal verification

security

bioinformatics

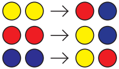
train safety

planning and scheduling



automated theorem proving

exploit generation

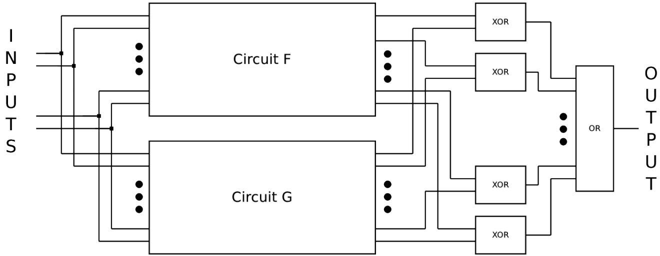


term rewriting termination

encode → **SAT solver** → **decode**

Combinatorial Equivalence Checking

Chip makers use SAT to check the correctness of their designs. Equivalence checking involves comparing a specification with an implementation or an optimized with a non-optimized circuit.



Motivation for Validating Proofs of Unsatisfiability

SAT solvers may have errors and only return yes/no.

- Documented bugs in SAT, SMT, and QSAT solvers;
  [Brummayer and Biere, 2009; Brummayer et al., 2010]
- Competition winners have contradictory results
  (HWMCC winners from 2011 and 2012)
- Implementation errors often imply conceptual errors;
- Proofs now mandatory for the annual SAT Competitions;
- Mathematical results require a stronger justification than a simple yes/no by a solver. UNSAT must be verifiable.

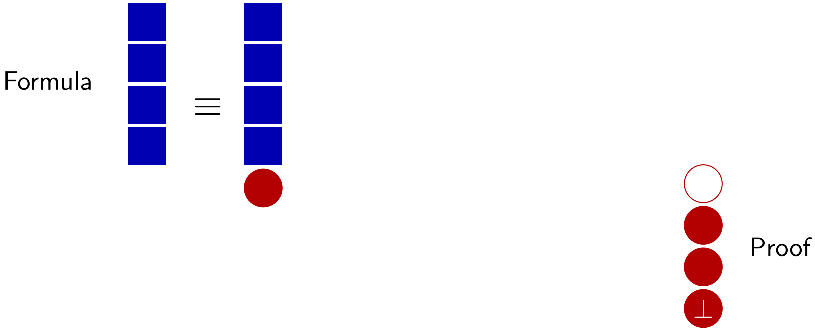
Proofs of Unsatisfiability

Proofs of Unsatisfiability

A clause C is satisfiability-preserving with respect to a formula F if F and $F \wedge C$ are both satisfiable or both unsatisfiable ($\equiv$). This property must be checkable in polynomial time.

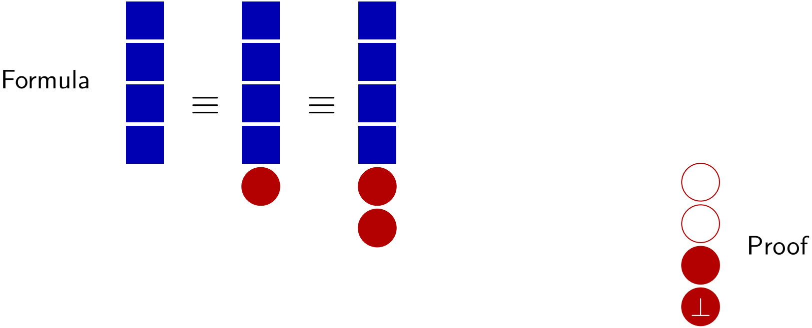
Proofs of Unsatisfiability

A clause C is satisfiability-preserving with respect to a formula F if F and $F \wedge C$ are both satisfiable or both unsatisfiable ($\equiv$). This property must be checkable in polynomial time.



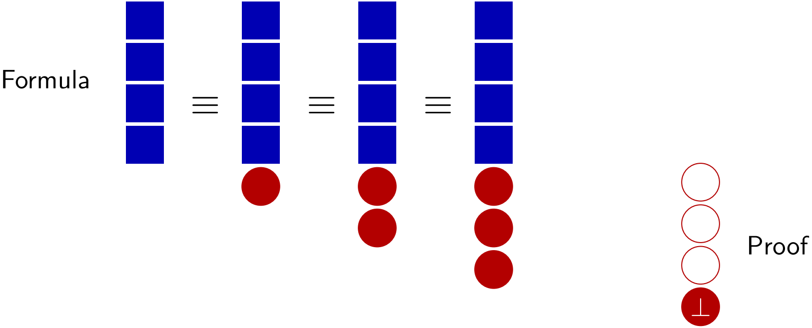
Proofs of Unsatisfiability

A clause C is satisfiability-preserving with respect to a formula F if F and $F \wedge C$ are both satisfiable or both unsatisfiable ($\equiv$). This property must be checkable in polynomial time.



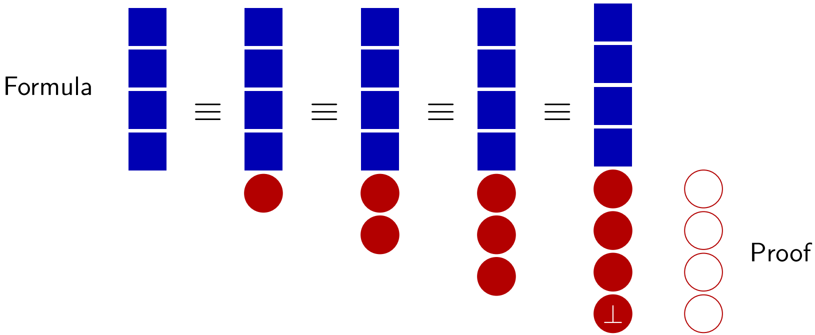
Proofs of Unsatisfiability

A clause C is satisfiability-preserving with respect to a formula F if F and $F \wedge C$ are both satisfiable or both unsatisfiable ($\equiv$). This property must be checkable in polynomial time.



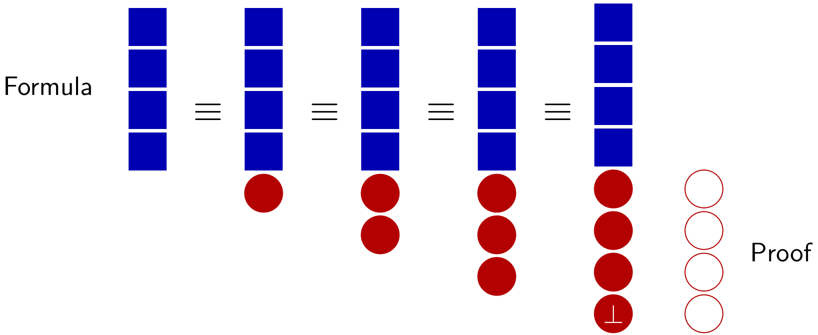
Proofs of Unsatisfiability

A clause C is satisfiability-preserving with respect to a formula F if F and $F \wedge C$ are both satisfiable or both unsatisfiable ($\equiv$). This property must be checkable in polynomial time.



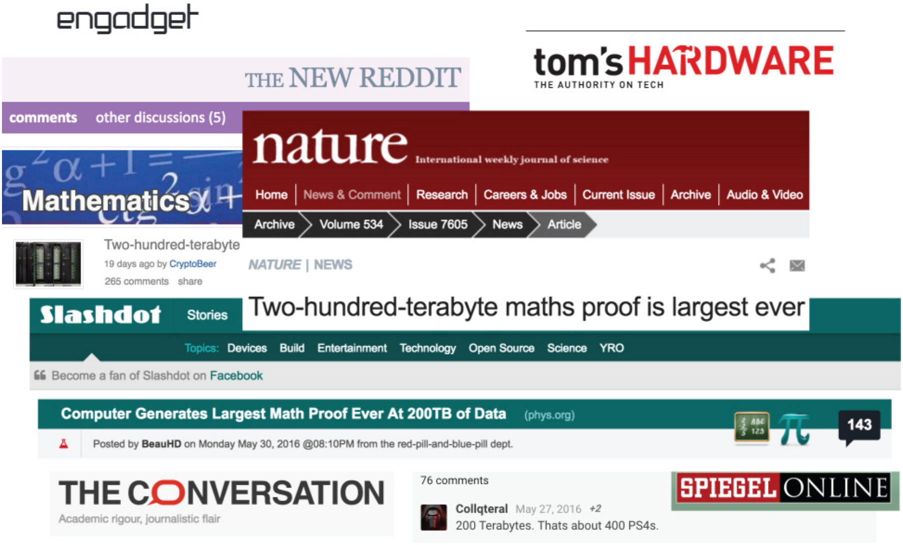
Proofs of Unsatisfiability

A clause C is satisfiability-preserving with respect to a formula F if F and $F \wedge C$ are both satisfiable or both unsatisfiable ($\equiv$). This property must be checkable in polynomial time.

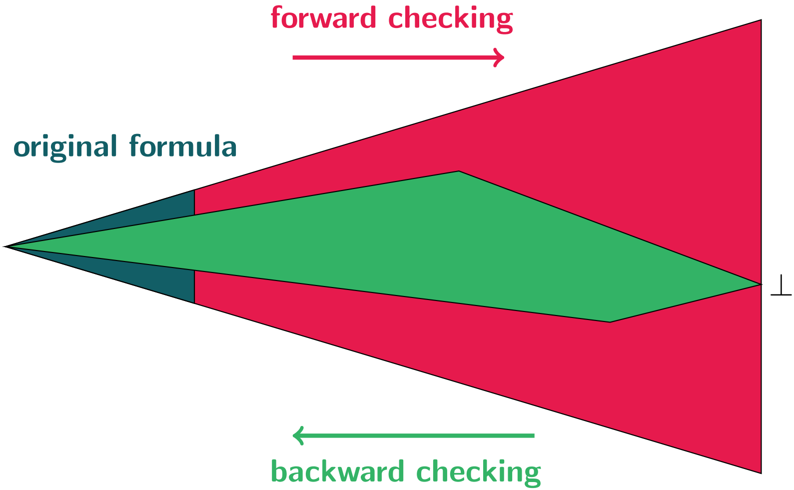


The Pythagorean Triples proof consists of a trillion added clauses (200TB), and it has been validated in 13,000 CPU hours.

Media: The Largest Math Proof Ever



Forward vs Backward Proof Checking



Efficient Proof Checking is Complicated

Forward Checking checks each addition step in a proof.

Backward Checking

- initialize by marking the empty clause;
- mark clauses (to check) using conflict analysis;
- skip unmarked clauses (up to 99% can be skipped);
- requires the entire proof to be in memory.

The key technique used to determine satisfiability-preserving of clauses is unit propagation. A naive algorithm scans over the entire formula for each unit. Sophisticated data-structures can boost performance, but are hard to reason about.

Efficient Certified Tools

ACL2: An Efficient, Interactive Theorem-Proving System

Our group has been working on the development and deployment of mechanized reasoning tools for 40 years.

A major focus of the development of the ACL2 theorem-proving system has been on efficient performance.

Some organizations using ACL2:

ACL2-Based, SAT Proof Checker

We developed a mechanically verified, ACL2-based, proof checker for proofs of unsatisfiability.

Given files containing:

- the initial conjecture, as a set of clauses, and
- an ordered list of proof steps ending with the empty clause,

our mechanically verified, SAT proof checker attempts to confirm the veracity of each proof step.

Parsing is hard, while writing is easy.

- after verification, we emit a conjecture that can be compared to the initial conjecture.
- a common tool, such as `diff`, can do the comparison.

Proof Claims

Basic Soundness.

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

Soundness Plus Formula Confirmation.

```
(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*

Eliminate Complexity

Certified proof checking challenges:

- backward checking is complex and heavy on memory;
- unit propagation is expensive.

We eliminate both challenges by modifying the proof:

- an efficient unverified tool removes the redundancy, making forward checking as fast as backward checking;
- searching for units is replaced by hints to locate units;
- the modified proofs are not much larger;
- we do not need to trust the unverified tool.

ACL2-Based, SAT Proof Checker Performance

We developed a litany of increasingly efficient solvers:

- use profiling to determine the most costly functions;
- reimplement them more efficiently and prove equivalence.

Table: Proof checking times in seconds 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

Confirming Code

One can attempt verify *real* applications – this is often difficult because a *live* system is often undergoing regular updates.

Another approach is to confirm run of an application by proof.

Both of these approaches have long been known.

Very sophisticated, possibly AI-based programs direct vehicles and weapons, and manage our financial system, and control our medical devices.

In the spirit of proof-carrying-code, we propose that developers emit their rationale for whatever their systems produce, and we develop a science of verifying results.

- Tools whose output can be checked in a fraction of the discovery time are good candidates.
- Developers can then spend less time testing their systems, and likely they can make faster systems.

Bringing It All Together

Tool Chain

Our tool chain to validate unsatisfiability results is as follows:

1. Given a formula F, a SAT solver produces a proof P;
2. A fast uncertified checker optimizes P resulting in Q:
   - Redundant proof steps are removed (up to 99% of the steps)
   - Hints are added to the proof to avoid search
3. A certified checker validates proof Q and emits formula F'.
4. Tools, such as diff, can check equivalence of F and F'.

Tool Chain

Our tool chain to validate unsatisfiability results is as follows:

1. Given a formula F, a SAT solver produces a proof P;
2. A fast uncertified checker optimizes P resulting in Q:
   - Redundant proof steps are removed (up to 99% of the steps)
   - Hints are added to the proof to avoid search
3. A certified checker validates proof Q and emits formula F'.
4. Tools, such as diff, can check equivalence of F and F'.

The efficient certified checker adds little overhead:

- Proof production (solving) is about 35% of the time;
- Proof optimization is about 55% of the time;
- Certified proof validation is about 10% of the time.

Conclusions

Conclusions

Verification of unsatisfiability results can now be achieved with reasonable overhead and high confidence in correctness:

- It is easy to emit proof emission in a SAT solver;
- The complex checking is turned into an oracle;
- A highly trusted checker, proved correct with ACL2, certifies the result.

Conclusions

Verification of unsatisfiability results can now be achieved with reasonable overhead and high confidence in correctness:

- It is easy to emit proof emission in a SAT solver;
- The complex checking is turned into an oracle;
- A highly trusted checker, proved correct with ACL2, certifies the result.

The technology is now ready for real-world applications:

- This tool chain is already used in industry (at Centaur);
- Huge proofs of mathematical theorems can be certified;
- The SAT 2017 Competition plans to use our tools to validate all results.

Certifying SAT Proofs

Marijn J.H. Heule, Warren A. Hunt, Jr., and Matt Kaufmann

THE UNIVERSITY OF TEXAS AT AUSTIN

High Confidence Software and Systems Conference, May 8, 2017

Checking Huge Proofs

Some proofs are too large to check in reasonable time sequentially. We can also check proofs in parallel:

- Solve F under multiple assignments A_i ($F \wedge A_i$ = UNSAT);
- All A_i together must cover the entire space (be a tautology);
- Certify with ACL2 that $F \models \overline{A_i}$ and print F and clause $\overline{A_i}$;
- Certify that all $\overline{A_i}$ together (merge using `cat`) are UNSAT.

Example

Consider assignments $A_1 = (x_1) \wedge (\bar{x}_2)$, $A_2 = (\bar{x}_1)$, $A_3 = (x_2)$.
Solve $F \wedge A_1$, $F \wedge A_2$, and $F \wedge A_3$ in parallel.
Certify that $F \models \overline{A_1}$, $F \models \overline{A_2}$, and $F \models \overline{A_3}$ in parallel.
Certify that $\overline{A_1} \wedge \overline{A_2} \wedge \overline{A_3}$ = UNSAT.