Formally Certified Satisfiability Solving

Duckki Oe

Computer Science, The University of Iowa, USA

April 23, 2012
Seoul National University

Satisfiability Solving

Satisfiability: “Is there a M such that $M \vDash \Phi$?”

Satisfiability (SAT)

- Input is a propositional formula

Satisfiability Modulo Theories (SMT)

- First-order formulas with equality
- Also w/ theories: integer/real arithmetic, array, bit-vector, ...
- Main target: decidable fragments (usually quantifier-free)
- Various SMT logics: combinations of those theories

SAT/SMT solvers are

- High-performance automated theorem provers
- Used in formal verification and artificial intelligence

SAT/SMT Solver Verification

Motivation

- Solvers are highly optimized (and large, $>$ 50k lines for SMT)
- To increase the trust level of all systems that use them

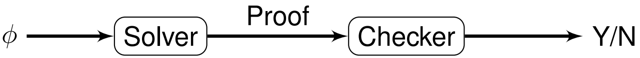
Approaches to Verified SAT/SMT Solving

- Verify the certificate from solvers:
  - Solvers are not trusted, a trusted checker is needed
  - SAT instance: a model that is found by the solver (easy for SAT)
  - UNSAT instance: a refutational proof
- Verify the code:
  - Prove theorems below using formal methods:
  - $\texttt{solve}(\Phi) = \mathit{SAT}$, then $\exists M. M \models \Phi$
  - $\texttt{solve}(\Phi) = \mathit{UNSAT}$, then $\forall M. M \not\models \Phi$
  - $\texttt{solve}(\Phi)$ terminates

Outline

- SAT/SMT Proof Checking
  - LFSC
  - Encoding SMT
  - Results
- Verifying SAT Solver Code
  - GURU
  - Specification
  - Implementation
  - Results

First Approach: SAT/SMT Proof Checking



Challenges

- High-performance checking: Proofs of 100s MB and even GBs
- Flexibility for extending (SMT): new theories are added all the time

The Proposal

Using a meta-language called LFSC

- Based on Edinburgh Logical Framework (LF)
- LF is a meta-logic based on type theory
- Formulas can be encode as *types* (and proofs as *terms*)
- Team @ Iowa developed an efficient LF type checker
- Extended with a side condition language for computations

Papers:

- **Towards an SMT Proof Format.**
  Aaron Stump and Duckki Oe. *SMT '08*
- **Fast and Flexible Proof Checking for SMT.**
  Duckki Oe, Andrew Reynolds, and Aaron Stump. *SMT '09*
- **Combining a Logical Framework with an RUP Checker for SMT Proofs.**
  Duckki Oe, and Aaron Stump. *SMT '11*

Proof Encoding Examples in LFSC

Syntax

declare $+$: int $\rightarrow$ int $\rightarrow$ int
declare $=$: int $\rightarrow$ int $\rightarrow$ form
declare and : form $\rightarrow$ form $\rightarrow$ form

Judgement

declare $\vdash$: form $\rightarrow$ **type**

Rules

declare AndE$_1$: $(\phi : \text{form}) \rightarrow (\psi : \text{form}) \rightarrow$
$(P : (\vdash (\text{and}\ \phi\ \psi))) \rightarrow (\vdash \phi)$

Proof

$\lambda F : \text{form}.\ \lambda G : \text{form}.\ \lambda P : (\vdash (\text{and}\ F\ G)).$
$(\text{AndE}_1\ \_\ \_\ P)$

Power of LFSC

LF's Higher-Order Abstract Syntax

$$\begin{array}{lcl}
\textbf{declare}\ \text{forall} & : & (\text{int} \to \text{form}) \to \text{form} \\
\textbf{declare}\ \text{inst} & : & (F : \text{int} \to \text{form}) \to (\vdash (\text{forall}\ F)) \to \\
 & & \quad (y : \text{int}) \to (\vdash (F\ y)) \\
\textbf{declare}\ \text{Impl-I} & : & (F : \text{form}) \to (G : \text{form}) \to ((\vdash F) \to (\vdash G)) \to \\
 & & \quad (\vdash (\text{imp}\ F\ G))
\end{array}$$

Side Condition Language Extension

$$\frac{\vdash C \vee v \quad \vdash D \vee \neg v}{\vdash C \vee D} \qquad \text{VS} \qquad \frac{\vdash C \quad \vdash D}{\vdash E}\ \text{resolve}(C, D, v) = E$$

- Simple (LISP-like) functional programming language
- Side conditions are like trusted tactics
- Built-in integer/rational operations (GNU MP library)

Encoding a SMT Logic - QF_IDL

QF_IDL - Quantifier Free Integer Difference Logic

Basic SMT reasoning

- SMT Solvers start with CNF conversion
- Elimination rules for logical connectives and *let*-bindings
- 32 CNF conversion rules, resolution rule

IDL theory reasoning

- Atomic formulas of the form: $x - y \leq c$ (w/ variation)
- 15 normalization rules, contradiction, transitivity
- $\text{idl}_{<} : (x : int) \to (y : int) \to (\vdash x < y) \to (\vdash x - y \leq -1)$
- $\text{idl\_contra} : \cdots \to (\vdash x - x \leq c) \to \{\ c < 0\ \} \to (\vdash \text{false})$

897 lines in LFSC (54 lines of side condition code)

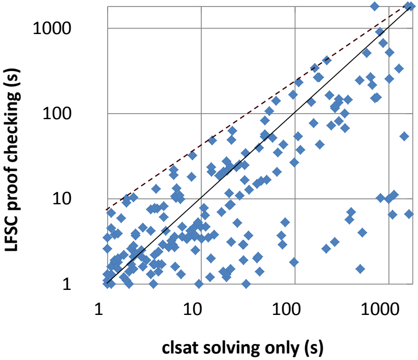
`clsat` - Proof Generating SAT/SMT Solver

- Implemented standard SAT/SMT features (written in C++)
- SMT-COMP 2008 Participant
- Proof generation overhead: $\leq 10\%$
- Proofs are constructed in memory and pruned before output

- Note: proofs can be dumped without storing in memory - larger proofs and larger overhead

Performance Results - QF_IDL

Configuration	Completed	Timeouts	Failures	Time(s)
clsat (solve only)	542	30	-	29,507.7
clsat+LFSC	539	32	1	38,833.6



UNSAT Benchmarks from SMT-LIB*
Timeout: 1,800 sec

Overall overhead: 31.6%
Worst overheads:
close to solving times

*Except for 50 benchmarks with unsupported language features

Summary: Proof Checking in LFSC

Flexibility

- Supports SMT logics and more
- Adding new rules is easy and modular

Performance

- Optimizations implemented once in LFSC
- Much faster checking than interactive theorem provers

Trustworthiness

- Trusted base: encoded logic + generic LFSC checker
- Encoded logics are intuitive
- More secure than ad-hoc proof checkers

Outline

- SAT/SMT Proof Checking
  - LFSC
  - Encoding SMT
  - Results
- Verifying SAT Solver Code
  - GURU
  - Specification
  - Implementation
  - Results

Second Approach: Verifying the Code (Related Works)

S. Lescuyer (2008) [Coq]

Classical DPLL: primitively recursive implementation

N. Shankar (2011) [PVS]

Modern DPLL: conflict analysis, clause learning, backjumping

F. Marić (2009) [Isabelle]

Modern DPLL: conflict analysis, clause learning, backjumping
Also implemented the two-literal watch lists

Summary

- Used model theoretic specification: $\exists M. M \vDash \Phi$
- Proved sound and complete
- Inefficient implementation at low-level

`versat`: a Verified SAT Solver

versat: A Verified Modern SAT Solver.
Duckki Oe, Aaron Stump, Corey Oliver, and Kevin Clancy. *VMCAI '12*

Goal: A verified real-world SAT Solver

- implemented modern-style DPLL with software engineerings
- low-level optimized using efficient data structure
- Verified all the way down to machine words and bits

Focus on performance & productivity

- Statically verified to produce sound UNSAT answers
- SAT certificates are checked at run-time (low overhead)
- No completeness(termination) proof
- Performance is more important than guarantee of termination

The Guru Programming Language

Guru is a functional programming language with:

Dependent type system (for verification)

Resource type system (for efficient code generation)

The Guru Programming Language

Guru is a functional programming language with:

Dependent type system (for verification)

- inductive datatypes (induction on first order variables)
- general recursion (reasoning about partial functions)
- first order logic with equality predicate (formula types)
- provable equality over operational semantics (call-by-value)

Resource type system (for efficient code generation)

The Guru Programming Language

Guru is a functional programming language with:

Dependent type system (for verification)

- inductive datatypes (induction on first order variables)
- general recursion (reasoning about partial functions)
- first order logic with equality predicate (formula types)
- provable equality over operational semantics (call-by-value)

Resource type system (for efficient code generation)

- configurable resource management policies:
- reference counting (default, automatic)
- linear typing (mutable data structures, annotations)
- arrays with constant time access

Specification: Soundness of UNSAT answer

Statement of Unsatisfiability

- Model Theoretically: “$\forall M. M \not\models \Phi$”
- Proof Theoretically: “$\Phi \vdash \bot$”
- They are all equivalent for propositional logic
- Solver returns UNSAT when the empty clause is deduced

Verification Strategy

- Verify the deduction steps follow the proof rules
- Isolate conflict analysis, where deductions are performed

Specification: Inference System (the `pf` type)

The `pf` type encodes "$\vdash_{res}$" (refutation complete)

```
Define lit := word
Define clause := <list lit>
Define formula := <list clause>

Inductive pf : Fun(F : formula)(C:clause).type :=
  pf_asm : Fun(F : formula)(C:clause)
              (u : { (member C F eq_clause) = tt }).    <pf F C>
| pf_res : Fun(F : formula)(C1 C2 Cr : clause)(l:lit)
              (d1 : <pf F C1>)
              (d2 : <pf F C2>)
              (u : { (is_resolvent Cr C1 C2 l) = tt }). <pf F Cr>
```

- `<pf F C>` type represents the judgement $F \vdash_{res} C$
- A value of `<pf F C>` represents a proof of $F \vdash_{res} C$
- Term constructors represents the inference rules
- `is_resolvent` tests if Cr is a resolvent of C1 and C2

Specification: The type of `solve` function

```
Inductive answer : Fun(F:formula).type :=
  sat : Fun(spec F:formula).<answer F>
| unsat : Fun(spec F:formula)(spec p:<pf F (nil lit)>).<answer F>

Define solve : Fun(F:formula).<answer F> := ...
```

- `(nil lit)` is the empty list of literals (the empty clause)
- `spec` (specificational) arguments are only for type checking
- So, proofs are not generated at run-time
- GURU makes sure that `spec` arguments are terminating and only dependent on the invariants (always computable)
- `solve` function should contain implementation and proof (internal verification)

Specification: Summary

- Encodes the propositional logic
- The trusted core of `versat`
- The rest of `versat` is actual implementation and proof
    - to be checked and certified by the GURU compiler
- Size: 259 lines of GURU code (small and straightforward)
- Includes a parser for DIMACS benchmark format
    - a trusted interpretation of string as `formula`
    - 145 lines (out of 259 lines)!

Implemented Features

A core set of modern features

Engineering:

- Two-literal Watch Lists
- Conflict Analysis + Fast Resolution
- Backjumping (Non-chronological Backtracking)

Heuristics:

- Decision Heuristics (Scoring variable activities)
- Clause Learning

Summary:

- 9884 lines of GURU code (20%) and proofs (80%)
- Proved 247 lemmas
- Also, added numerous lemmas in the standard library

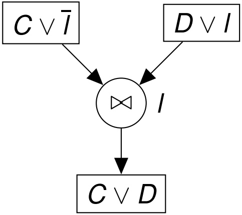
Efficient Representation of Clauses

`aclause` type: array-based clause and invariants

```
Inductive aclause : Fun(nv:word)(F:formula).type :=
  mk_aclause : Fun(spec nv:word)(spec F:formula)
                  (spec n:word)(l:<array lit n>)
                  (u1:{ (array_in_bounds nv l) = tt })
                  (spec c:clause)(spec pf_c:<pf F c>)
                  (u2:{ c = (to_cl l) })
                 .<aclause nv F>
```

- keep specification simple, implementation efficient
- `aclause` stores a clause in the array
- array_in_bounds: all variable numbers are within bounds and the array is null-terminated
- `to_cl` interprets a null-terminated array as a list
- the interpretation of array is valid in F

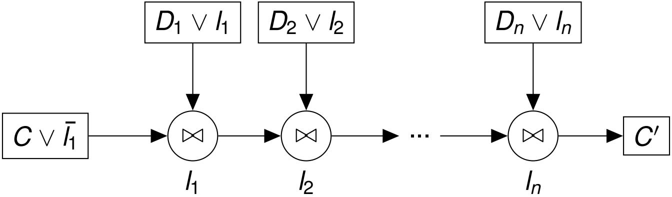
Conflict Analysis with Fast Resolution (1/7)



var	status
$\vdots$	
v	Not/Pos/Neg
$\vdots$	

- Problem: duplicate literals
- Solution: a look-up table

Conflict Analysis with Fast Resolution (2/7)



- $l_1 \dots l_n$ are assigned after the last decision literal
- C' will have only one literal assigned after the last decision

Conflict Analysis with Fast Resolution (3/7)

An Example Conflict:

Clauses
$\bar{3}\ \bar{4}$
$\bar{1}\ 4\ 5$
$2\ \bar{3}\ 4\ \bar{5}$
$\vdots$

Assignment Sequence: $1^d, \bar{2}, 3^d, \bar{4}, 5$

$\Longrightarrow$ conflicting with $2\ \bar{3}\ 4\ \bar{5}$

Analysis:

C	D	I
$2\ \bar{3}\ 4\ \bar{5}$		

Conflict Analysis with Fast Resolution (3/7)

An Example Conflict:

Clauses
$\bar{3}\ \bar{4}$
$\bar{1}\ 4\ 5$
$2\ \bar{3}\ 4\ \bar{5}$
$\vdots$

Assignment Sequence: $1^d, \bar{2}, 3^d, \bar{4}, 5$

$\Longrightarrow$ conflicting with $2\ \bar{3}\ 4\ \bar{5}$

Analysis:

C	D	l
$2\ \bar{3}\ 4\ \bar{5}$	$\bar{1}\ 4\ 5$	5
$2\ \bar{3}\ 4\ \bar{1}$		

Conflict Analysis with Fast Resolution (3/7)

An Example Conflict:

Clauses
$\bar{3}\ \bar{4}$
$\bar{1}\ 4\ 5$
$2\ \bar{3}\ 4\ \bar{5}$
$\vdots$

Assignment Sequence: $1^d, \bar{2}, 3^d, \bar{4}, 5$

$\Longrightarrow$ conflicting with $2\ \bar{3}\ 4\ \bar{5}$

Analysis:

C	D	l
$2\ \bar{3}\ 4\ \bar{5}$	$\bar{1}\ 4\ 5$	5
$2\ \bar{3}\ 4\ \bar{1}$	$\bar{3}\ \bar{4}$	$\bar{4}$
$2\ \bar{3}\ \bar{1}$		

- Time complexity of removal depends on the length of C
- Literals being resolved are assigned after the last decision

Conflict Analysis with Fast Resolution (4/7)

An Example Conflict:

Clauses
$\bar{3}\ \bar{4}$
$\bar{1}\ 4\ 5$
$2\ \bar{3}\ 4\ \bar{5}$
$\vdots$

Assignment Sequence: $1^d, \bar{2}, 3^d, \bar{4}, 5$

$\Longrightarrow$ conflicting with $2\ \bar{3}\ 4\ \bar{5}$

Analysis (Old & Improved):

C	D	l
$2\ \bar{3}\ 4\ \bar{5}$	$\bar{1}\ 4\ 5$	5
$2\ \bar{3}\ 4\ \bar{1}$	$\bar{3}\ \bar{4}$	$\bar{4}$
$2\ \bar{3}\ \bar{1}$		

C_1	C_2	D	l
2	$\bar{3}\ 4\ \bar{5}$		

Conflict Analysis with Fast Resolution (4/7)

An Example Conflict:

Clauses
$\bar{3}\ \bar{4}$
$\bar{1}\ 4\ 5$
$2\ \bar{3}\ 4\ \bar{5}$
$\vdots$

Assignment Sequence: $1^d, \bar{2}, 3^d, \bar{4}, 5$

$\Longrightarrow$ conflicting with $2\ \bar{3}\ 4\ \bar{5}$

Analysis (Old & Improved):

C	D	l
$2\ \bar{3}\ 4\ \bar{5}$	$\bar{1}\ 4\ 5$	5
$2\ \bar{3}\ 4\ \bar{1}$	$3\ \bar{4}$	$\bar{4}$
$2\ \bar{3}\ \bar{1}$		

C_1	C_2	D	l
2	$\bar{3}\ 4\ \bar{5}$	$\bar{1}\ 4\ 5$	5
$2\ \bar{1}$	$\bar{3}\ 4$		

Conflict Analysis with Fast Resolution (4/7)

An Example Conflict:

Clauses
$\bar{3}\ \bar{4}$
$\bar{1}\ 4\ 5$
$2\ \bar{3}\ 4\ \bar{5}$
$\vdots$

Assignment Sequence: $1^d, \bar{2}, 3^d, \bar{4}, 5$

$\Longrightarrow$ conflicting with $2\ \bar{3}\ 4\ \bar{5}$

Analysis (Old & Improved):

C	D	l
$2\ \bar{3}\ 4\ \bar{5}$	$\bar{1}\ 4\ 5$	5
$2\ \bar{3}\ 4\ \bar{1}$	$\bar{3}\ \bar{4}$	$\bar{4}$
$2\ \bar{3}\ \bar{1}$		

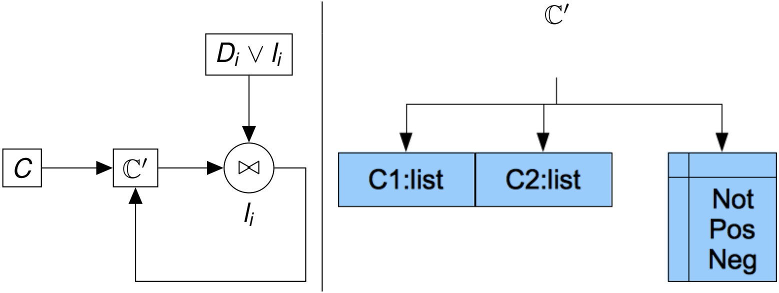
C_1	C_2	D	l
2	$\bar{3}\ 4\ \bar{5}$	$\bar{1}\ 4\ 5$	5
$2\ \bar{1}$	$\bar{3}\ 4$	$\bar{3}\ \bar{4}$	$\bar{4}$
$2\ \bar{1}$	$\bar{3}$		

- Time complexity of removal depends on the length of C_2

Conflict Analysis with Fast Resolution (5/7)

Data Structure:

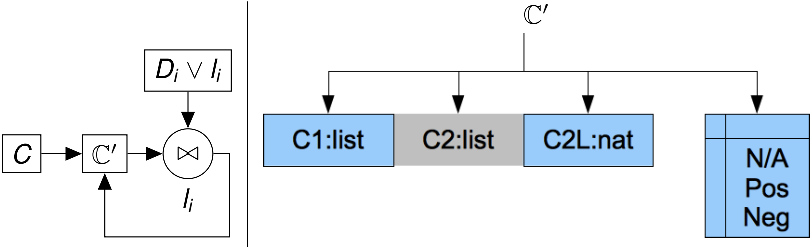
- For duplication removal & faster remove operation



Conflict Analysis with Fast Resolution (6/7)

Data Structure (even better):

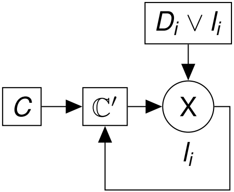
- For duplication removal & constant time remove operation

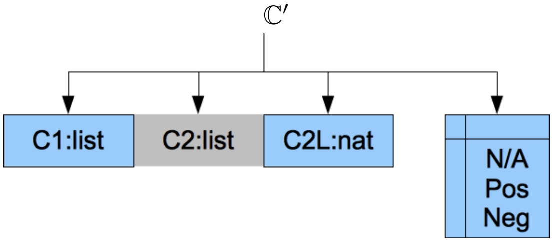


- C2 is not calculated at run-time & C2L tracks the length
- Removal is a constant time operation!
- At the end, C2 has only one literal (C2L = 1)
- At the end, C2 can be deduced

Conflict Analysis with Fast Resolution (7/7)

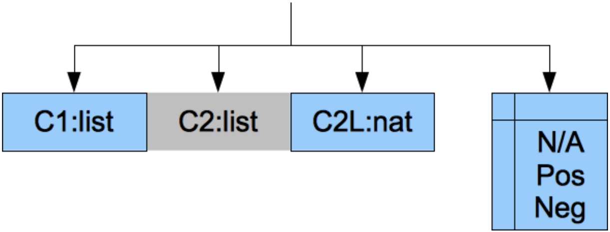
Invariants:





```
(u1:{ (all_lits_are_assigned T (append C1 C2)) = tt })
(u2:{ (cl_has_all_vars (append C1 C2) T) = tt })
(u3:{ (cl_set_at_prev_levels dl dls C1) = tt })
(u4:{ C2L = (length C2) })
(u5:{ (cl_unique C2) = tt })
```

Example Theorem: Clearing the Look-up Table



```
Define cl_has_all_vars_implies_clear_vars_like_new :
  Forall (nv:word)
         (T:<array assignment nv>)
         (C:clause)
         (u:{ (cl_valid nv C) = tt })
         (r:{ (cl_has_all_vars C T) = tt })
    .{ (clear_vars T C) = (array_new nv UN) }
```

Results: `versat` vs. State-of-the-art Solvers

SAT Race 2008 Test Set 1

- 50 benchmarks
- System: Intel Xeon X5650 2.67GHz w/ 12GB of memory
- 900 seconds timeout for solving

Systems	#Solved	#Timeout	#Error/Wrong
versat	19	31	0
picosat-936	46	4	0
minisat-2.2.0	47	3	0

Note: `versat` solved `velev-live-sat-1.0-03` (78MB size, 224,920 variables, 3,596,474 clauses)

Results: `versat` vs. proof checking

The Certified Track benchmarks of SAT Competition 2007

- 16 benchmarks (believed to be UNSAT)
- System: Intel Core 2 Duo 2.40GHz w/ 3GB of memory
- One hour timeout for solving and checking, individually

Systems	#Solved	#Certified
versat	6	6
picosat + RUP	14	4
picosat + TraceCheck	14	12

Trusted Base:

- `versat`: GURU compiler + 259 lines of GURU code
- checker3 (RUP checker): 1,538 lines of C code
- tracecheck (TraceCheck checker): 2,989 lines of C code + `boolforce` library (minisat-2.2.0 is ≈2,500 lines of C++)

Summary: Verifying a SAT Solver

`versat`: a modern SAT solver verified in GURU

- UNSAT-soundness is verified statically
- Can solve and certify realistic formulas
- Comparable with the current proof checking technology
- Source code is available at `http://cs.uiowa.edu/~duoe/`
- Standalone certified C code is also available

Future Work

Reusing `versat` code base

- Add features: Restarting, Preprocessing, CC Minimization
- Implement other tools: verified/efficient RUP proof checker

Real-world software engineering and verification

- Verified code library for software engineering techniques
    - low-level optimizations
    - design patterns
- Software engineering for verification
    - Lemma database: sharing and exchanging proofs
    - Enforcing high-level design onto low-level details and implementations