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 $M \models \Phi$?”

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:
    - $\text{solve}(\Phi) = SAT$, then $\exists M. M \models \Phi$
    - $\text{solve}(\Phi) = UNSAT$, then $\forall M. M \not\models \Phi$
    - $\text{solve}(\Phi)$ 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 → int → int
declare = : int → int → form
declare and : form → form → form
```

Judgement

```
declare ⊢ : form → type
```

Rules

```
declare AndE₁ : (φ : form) → (ψ : form) →
(P : (⊢ (and φ ψ))) → (⊢ φ)
```

Proof

```
λF : form. λG : form. λP : (⊢ (and F G)).
(AndE₁ _ _ P)
```
Power of LFSC

LF’s Higher-Order Abstract Syntax

```
declare forall  : (int → form) → form
declare inst    : (F : int → form) → (⊢ (forall F)) →
                   (y : int) → (⊢ (F y))
declare Impl-I  : (F : form) → (G : form) → ((⊢ F) → (⊢ G)) →
                   (⊢ (imp F G))
```

Side Condition Language Extension

\[
\begin{align*}
\frac{\vdash C \lor v \quad \vdash D \lor \neg v}{\vdash C \lor D} & \quad \text{VS} \\
\frac{\vdash C}{\vdash E} & \quad \text{resolve}(C, D, v) = E
\end{align*}
\]

- 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
- $\text{idl}_< : (x : \text{int}) \rightarrow (y : \text{int}) \rightarrow (\not\exists \ x < y) \rightarrow (\not\exists \ x - y \leq -1)$
- $\text{idl}_\text{contra} : \cdots \rightarrow (\not\exists \ x - x \leq c) \rightarrow \{ \ c < 0 \ \} \rightarrow (\not\exists \ false)$

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

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Completed</th>
<th>Timeouts</th>
<th>Failures</th>
<th>Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>clsat (solve only)</td>
<td>542</td>
<td>30</td>
<td>-</td>
<td>29,507.7</td>
</tr>
<tr>
<td>clsat+LFSC</td>
<td>539</td>
<td>32</td>
<td>1</td>
<td>38,833.6</td>
</tr>
</tbody>
</table>

**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: $\exists M. M \models \Phi$
- 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 engineering
- 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: “∀M. M ⊭ Φ”
- Proof Theoretically: “Φ ⊢ ⊥”
- 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 \( lit := \text{word} \)
Define \( clause := \langle \text{list lit} \rangle \)
Define \( formula := \langle \text{list clause} \rangle \)

Inductive \( pf : \text{Fun}(F : \text{formula})(C : \text{clause}).\text{type} := \)
\( \quad pf_{\text{asm}} : \text{Fun}(F : \text{formula})(C : \text{clause}) \)
\( \quad \quad \quad (u : \{ \text{(member C F eqclause)} = \text{tt} \}). \langle pf F C \rangle \)
\( \quad | \quad pf_{\text{res}} : \text{Fun}(F : \text{formula})(C1 C2 Cr : \text{clause})(l : \text{lit}) \)
\( \quad \quad \quad (d1 : \langle pf F C1 \rangle) \)
\( \quad \quad \quad (d2 : \langle pf F C2 \rangle) \)
\( \quad \quad \quad (u : \{ \text{(is_resolvent Cr C1 C2 l)} = \text{tt} \}). \langle pf F Cr \rangle \)

- \( \langle pf F C \rangle \) type represents the judgement \( F \vdash_{res} C \)
- A value of \( \langle pf F C \rangle \) 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 $\texttt{solve}$ 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> := ...

- $(\text{nil lit})$ is the empty list of literals (the empty clause)
- $\text{spec}$ (specificational) arguments are only for type checking
- So, proofs are not generated at run-time
- $\text{Guru}$ makes sure that $\text{spec}$ arguments are terminating and only dependent on the invariants (always computable)
- $\text{solve}$ 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)

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

<table>
<thead>
<tr>
<th>var</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Not/Pos/Neg</td>
</tr>
</tbody>
</table>

\[ C \lor \bar{l} \]
\[ D \lor l \]
\[ l \]
\[ C \lor D \]
- $l_1 \ldots l_n$ are assigned after the last decision literal
- $C'$ will have only one literal assigned after the last decision
An Example Conflict:

<table>
<thead>
<tr>
<th>Clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bar{3} \bar{4}$</td>
</tr>
<tr>
<td>$\bar{1} 4 5$</td>
</tr>
<tr>
<td>$2 \bar{3} 4 \bar{5}$</td>
</tr>
</tbody>
</table>

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

Analysis:

<table>
<thead>
<tr>
<th>$C$</th>
<th>$D$</th>
<th>$I$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2 \bar{3} 4 \bar{5}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Conflict Analysis with Fast Resolution (3/7)

An Example Conflict:

<table>
<thead>
<tr>
<th>Clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \bar{3} \bar{4} )</td>
</tr>
<tr>
<td>( \bar{1} 4 5 )</td>
</tr>
<tr>
<td>( 2 \bar{3} 4 \bar{5} )</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

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

\[ \Rightarrow \text{conflicting with } 2 \bar{3} 4 \bar{5} \]

Analysis:

<table>
<thead>
<tr>
<th>C</th>
<th>D</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 2 \bar{3} 4 \bar{5} )</td>
<td>( \bar{1} 4 5 )</td>
<td>5</td>
</tr>
<tr>
<td>( 2 \bar{3} 4 \bar{1} )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Conflict Analysis with Fast Resolution (3/7)

An Example Conflict:

<table>
<thead>
<tr>
<th>Clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bar{3} \bar{4}$</td>
</tr>
<tr>
<td>$\bar{1} \bar{4} \bar{5}$</td>
</tr>
<tr>
<td>$2 \bar{3} \bar{4} \bar{5}$</td>
</tr>
</tbody>
</table>

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

Analysis:

<table>
<thead>
<tr>
<th>$C$</th>
<th>$D$</th>
<th>$I$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2 \bar{3} \bar{4} \bar{5}$</td>
<td>$\bar{1} \bar{4} \bar{5}$</td>
<td>5</td>
</tr>
<tr>
<td>$2 \bar{3} \bar{4} \bar{1}$</td>
<td>$\bar{3} \bar{4}$</td>
<td>$\bar{4}$</td>
</tr>
<tr>
<td>$2 \bar{3} \bar{1}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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:

<table>
<thead>
<tr>
<th>Clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \overline{3} \overline{4} )</td>
</tr>
<tr>
<td>( \overline{1} 4 5 )</td>
</tr>
<tr>
<td>( 2 \overline{3} 4 \overline{5} )</td>
</tr>
</tbody>
</table>

Assignment Sequence: \( 1^d, \overline{2}, 3^d, \overline{4}, 5 \)

\( \implies \) conflicting with \( 2 \overline{3} 4 \overline{5} \)

Analysis (Old & Improved):

<table>
<thead>
<tr>
<th>( C )</th>
<th>( D )</th>
<th>( I )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 2 \overline{3} 4 \overline{5} )</td>
<td>( \overline{1} 4 5 )</td>
<td>( 5 )</td>
</tr>
<tr>
<td>( 2 \overline{3} 4 \overline{1} )</td>
<td>( \overline{3} 4 )</td>
<td>( 4 )</td>
</tr>
<tr>
<td>( 2 \overline{3} \overline{1} )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( C_1 )</th>
<th>( C_2 )</th>
<th>( D )</th>
<th>( I )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 2 )</td>
<td>( \overline{3} 4 \overline{5} )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Conflict Analysis with Fast Resolution (4/7)

An Example Conflict:

<table>
<thead>
<tr>
<th>Clauses</th>
<th>Assignment Sequence: $1^d, \overline{2}, 3^d, \overline{4}, 5$ \implies conflicting with $2 \overline{3} 4 \overline{5}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\overline{3} \overline{4}$</td>
<td></td>
</tr>
<tr>
<td>$\overline{1} 4 5$</td>
<td></td>
</tr>
<tr>
<td>$2 3 4 \overline{5}$</td>
<td></td>
</tr>
</tbody>
</table>

#### Analysis (Old & Improved):

<table>
<thead>
<tr>
<th>$C$</th>
<th>$D$</th>
<th>$I$</th>
<th>$C_1$</th>
<th>$C_2$</th>
<th>$D$</th>
<th>$I$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2 3 4 \overline{5}$</td>
<td>$\overline{1} 4 5$</td>
<td>5</td>
<td>2 3 4 $\overline{5}$</td>
<td>$\overline{1} 4 5$</td>
<td>5</td>
<td>2 3 4</td>
</tr>
<tr>
<td>$2 \overline{3} 4 \overline{1}$</td>
<td>$3 4$</td>
<td>$\overline{4}$</td>
<td>2 $\overline{1}$</td>
<td>$3 4$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Conflict Analysis with Fast Resolution (4/7)

An Example Conflict:

<table>
<thead>
<tr>
<th>Clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \overline{3} \overline{4} )</td>
</tr>
<tr>
<td>( \overline{1} \overline{4} \overline{5} )</td>
</tr>
<tr>
<td>( 2 \overline{3} \overline{4} \overline{5} )</td>
</tr>
<tr>
<td>( \vdots )</td>
</tr>
</tbody>
</table>

Assignment Sequence: \( 1^d, \overline{2}, 3^d, \overline{4}, 5 \)  
\( \implies \) conflicting with \( 2 \overline{3} \overline{4} \overline{5} \)

Analysis (Old & Improved):

\[
\begin{array}{cccc}
C & D & I & \\
\hline
2 \overline{3} \overline{4} \overline{5} & \overline{1} & 4 & 5 \\
2 \overline{3} \overline{4} & 3 & \overline{4} & 4 \\
2 \overline{3} & \overline{3} & \overline{1} & \\
\end{array}
\quad
\begin{array}{cccc}
C_1 & C_2 & D & I \\
\hline
2 & 3 & 4 & 5 \\
2 & \overline{1} & 3 & 4 \\
2 & \overline{1} & 3 & \\
\end{array}
\]

- Time complexity of removal depends on the length of \( C_2 \)
Data Structure:

- For duplication removal & faster remove operation

\[ D_i \vee l_i \]

\[ C \rightarrow C' \rightarrow \bigcirc \rightarrow l_i \]

\[ C' \rightarrow C1: \text{list} \quad C2: \text{list} \]

\[ \text{Not Pos Neg} \]
Conflict Analysis with Fast Resolution (6/7)

Data Structure (even better):

- For duplication removal & constant time remove operation

- $D_i \lor l_i$

- $C \rightarrow C'$

- $l_i$

- $C' \rightarrow C_1: \text{list}$

- $C_2: \text{list}$

- $C_2L: \text{nat}$

- $C'$

- $C_2$ is not calculated at run-time & $C_2L$ tracks the length

- Removal is a constant time operation!

- At the end, $C_2$ has only one literal ($C_2L = 1$)

- At the end, $C_2$ can be deduced

- N/A Pos Neg
Invariants:

\[
\begin{align*}
D_i \lor l_i & \\
C & \rightarrow C' \\
\rightarrow X \\
l_i &
\end{align*}
\]

\[
\begin{align*}
D_i \lor l_i & \\
C & \rightarrow C' \\
\rightarrow X \\
l_i &
\end{align*}
\]

\[
\begin{align*}
(u1:y\{\text{all_lits_are_assigned}\ T\ \text{(append}\ C1\ C2)\} = \text{tt} )
\end{align*}
\]

\[
\begin{align*}
(u2:y\{\text{cl_has_all_vars}\ \text{(append}\ C1\ C2)\ T\} = \text{tt} )
\end{align*}
\]

\[
\begin{align*}
(u3:y\{\text{cl_set_at_prev_levels}\ d1\ dls\ C1) = \text{tt } )
\end{align*}
\]

\[
\begin{align*}
(u4:y\{\text{C2L} = \text{length}\ C2\} )
\end{align*}
\]

\[
\begin{align*}
(u5:y\{\text{cl_unique}\ C2) = \text{tt } )
\end{align*}
\]
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) }

Duckki Oe
Formally Certified Satisfiability Solving
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

<table>
<thead>
<tr>
<th>Systems</th>
<th>#Solved</th>
<th>#Timeout</th>
<th>#Error/Wrong</th>
</tr>
</thead>
<tbody>
<tr>
<td>versat</td>
<td>19</td>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>picosat-936</td>
<td>46</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>minisat-2.2.0</td>
<td>47</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Systems</th>
<th>#Solved</th>
<th>#Certified</th>
</tr>
</thead>
<tbody>
<tr>
<td>versat</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>picosat + RUP</td>
<td>14</td>
<td>4</td>
</tr>
<tr>
<td>picosat + TraceCheck</td>
<td>14</td>
<td>12</td>
</tr>
</tbody>
</table>

**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++)
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