Generating Small Countermodels using SMT

Andrew Reynolds
MVD
September 21, 2012
Acknowledgements

• Intel Corporation
  – Amit Goel, Sava Krstic

• University of Iowa
  – Cesare Tinelli, Francois Bobot

• New York University
  – Clark Barrett, Morgan Deters, Dejan Jovanovic
Overview

• Satisfiability Modulo Theories (SMT)
• SMT-Based System Verification
  – Deductive Verification Framework (DVF)
• Challenge of quantifiers in SMT
  – Why do we care about quantifiers?
  – Why are quantifiers difficult?
• Finite Model Finding
• Experimental Results
Satisfiability Modulo Theories (SMT)

• SMT solvers:
  – Are powerful tools for determining satisfiability of ground formulas
    • Built-in decision procedures for many theories
      – Arithmetic, arrays, bit vectors, datatypes, ...
  – Have improved performance in past 10 years

• Verification applications rely on SMT solvers
  – System verifier DVF used by Intel
SMT-Based System Verification

System + Specifications → DVF

DVF → Verification Condition

Verification Condition → SMT solver

SMT solver → All verification conditions hold

SMT solver → Some verification condition fails
DVF Example

- **Definitions**
  ```
  type resource
  const resource null
  type process
  var array(resource, bool) valid = mk_array[resource](false)
  var array(resource, int) count
  var array(process, resource) ref = mk_array[process](null)
  ...
  module S = Set?type process>
  ```

- **Transition System**
  ```
  transition create (resource r)
  require (r != null, !valid[r]){
    valid[r] := true;
    count[r] := 0;
  }
  ...
  ```

- **Properties**
  ```
  def bool prop = forall (process p) (ref[p] != null => valid[ref[p]])
  def bool refs_non_zero = forall (process p) (ref[p] != null => count[ref[p]] > 0)
  ...
  ```

- **Goals**
  ```
  goal main = invariant prop assuming refs_non_zero
  ...
  goal rnz = formula (... && prop && ... => refs_non_zero)
  ```

- Language corresponds closely to SMT constraints
Goals translated into (possibly multiple) SMT queries
SMT Query

Definitions

S, P, R : type
null : R
valid: Array( R, Bool )
count: Array( R, Int )
ref: Array( P, R )
empty : S
mem : (S, P) -> Bool
add, remove : (S, P) -> S
...

Axioms

∀x : R. count[x] > 0 => valid[x]
∀x : P. ¬ mem( empty, x )
∀x : S, y, z : P. mem( add( x, y ), z ) => ( z = y ∨ mem( x, z ) )
∀x : S, y, z : P. mem( remove( x, y ), z ) => ( z ≠ y ∧ mem( x, z ) )
...

¬ ( ... ∀x. (ref[x] != null => valid[ref[x]]) ...)

Property to verify
SMT: DPLL(T) Architecture

Formula $F$

SAT Solver

Theory Solvers

Satisfying assignment $M$

$F$ is SAT

$F$ is UNSAT

UNSAT

$M$ is T-Consistent

$M$ is T-Inconsistent

Clauses to add to $F$

$F$ is SAT

$F$ is UNSAT
Why Quantifiers?

• Quantifiers exist in verification conditions:

Definitions

- S, P, R : type
- null : R
- valid: Array( R, Bool )
- count: Array( R, Int )
- ref: Array( P, R )
- empty : S
- mem : (S, P) -> Bool
- add : (S, P) -> S

Axioms

- $\forall x : R. \text{count}[x] > 0 \Rightarrow \text{valid}[x]$
- $\forall x : P. \neg \text{mem}(\text{empty}, x)$
- $\forall x : S, y, z : P. \text{mem}(\text{add}(x, y), z) \Rightarrow (z = y \lor \text{mem}(x, z))$
- $\forall x : S, y, z : P. \text{mem}(\text{remove}(x, y), z) \Rightarrow (z \neq y \land \text{mem}(x, z))$
- ...

Property to verify

- $\neg (\ldots \forall x. (\text{ref}[x] \neq \text{null} \Rightarrow \text{valid}[\text{ref}[x]]) \ldots)$
Challenge of Quantifiers in SMT

• In general, determining T-consistency of a set of quantified formulas is **undecidable**

• SMT solvers will typically:
  – Add ground instances of quantified formulas
    • i.e. for $\forall x. F$, add lemmas $F[t_1/x]$, $F[t_2/x]$, ...
      – If ground conflict exists, answer UNSAT
      – Otherwise, may continue indefinitely
  – Sound but incomplete procedure
Handling Verification Conditions

Verification Condition for property P

SMT Solver

UNSAT

Property P is verified

Unknown

Candidate Model

Manual Inspection
Handling Verification Conditions

Verification Condition for property P

SMT Solver

UNSAT

Property P is verified

Unknown

SAT

Candidate Model

Model

Manual Inspection

⇒ Need method for answering SAT
Finite Model Finding

• Method to answer SAT in presence of quantifiers
• Given (G, Q):
  – Set of ground constraints G
  – Set of quantified assertions Q

1. Find a “smallest” model for G
   • Least number of equivalence classes for terms
2. Try every instance of Q in the model
   • Feasible if # eq classes we need to consider is finite
3. If every instance is true in model, answer SAT

• Consider quantifiers over uninterpreted sorts
  – Values, Addresses, Processes, Resources, Sets, ...
Finite Model Finding : Example

\[
\begin{align*}
\text{G} & \quad \text{Q} \\
\{a\} & \quad \{b, c\} \\
a \neq b, b = c, \forall x. f(x) = x
\end{align*}
\]

1. Smallest model for G, size 2 : \{a\}, \{b, c\}
2. Substitute Q with [a/x], [b/x]:
   - f(a) = a, f(b) = b added to G
3. Afterwards: \{a, f(a)\}, \{b, c, f(b)\}
   - All instances are true
     \[\Rightarrow\] answer SAT
Finding Small Models

• “Smallest” model for sort S means:
  – Fewest # equivalence classes of sort S

• To find small models:
  – Try to find models of size 1, 2, 3, … etc.
    • Impose cardinality constraints

• Requires solver for equality with cardinality constraints
Solver for Eq + Cardinality Constraints

• Maintain disequality graph
  – Nodes are equivalence classes
  – Edges are disequalities
• For cardinality $k$, interested whether graph is $k$-colorable

• Partition disequality graph of the solver into regions where the edge density is high, so that we:
  – Discover cliques local to regions
  – Suggest relevant terms to identify
Why Small Models?

• Easier to test against quantifiers
  – Given quantified formula $\forall x_1...x_n. F$
    • Naively, we require $k^n$ instantiations,
      – where $k$ is the cardinality of $\text{sort}(x_1...x_n)$
  – Feasible if either:
    • Both $n$ and $k$ are small
    • We can recognize/eliminate redundant instantiations
      – *Model-Based Quantifier Instantiation* [Ge/deMoura 09]
        – i.e. do not consider instances that are already true
Anatomy of Finite Model Finding

SAT Solver

Theory Solvers

Verification Condition for property P

Satisfying assignment M
(with quantifiers)

M is T-Consistent

M is T-Inconsistent

Theory conflicts

UNSAT
Anatomy of Finite Model Finding

SAT Solver

Satisfying assignment $M$ (with quantifiers)

Theory Solvers

$M$ is $T$-Consistent

Eq + Cardinality Solver

$M$ is not minimal

Cardinality conflicts, splits

$M$ is minimal

UNSAT

Verification Condition for property $P$
Anatomy of Finite Model Finding

Verification Condition for property P

SAT Solver

Satisfying assignment M (with quantifiers)

Theory Solvers

M is T-Consistent

Eq + Cardinality Solver

M is minimal

Exhaustive Quant. Instantiation

No new instantiations

Filter Based on Model

SAT

Relevant instantiations

UNSAT
FMF + Heuristic Instantiation

• Idea:
  – First see if instantiations based on heuristics exist
  – If not, resort to exhaustive instantiation

• May lead to:
  – Answering UNSAT more often
    • Discover easy conflicts, if they exist
  – Arriving at model faster
    • Instantiations rule out spurious models
FMF + Heuristic Instantiation

Verification Condition for property P

SAT Solver

Satisfying assignment M (with quantifiers)

Theory Solvers

Eq + Cardinality Solver

M is T-Consistent

M is minimal

Heuristic Quant. Instantiation

No new instantiations

Exhaustive Quant. Instantiation

No new instantiations

Filter Based on Model

SAT

UNSAT

Instantiations
Experimental Results

• Implemented in SMT Solver CVC4

• DVF Benchmarks
  – Taken from real examples of interest to Intel
  – Both SAT/UNSAT benchmarks
    • SAT benchmarks generated by removing necessary pf assumptions
  – Many theories: UF, arithmetic, arrays, datatypes

• TPTP Benchmarks
  – Taken from ATP community
  – Heavily quantified
  – Unsorted logic
## Results: DVF

<table>
<thead>
<tr>
<th></th>
<th>german</th>
<th>refcount</th>
<th>agree</th>
<th>apg</th>
<th>bmk</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>cvc4</strong></td>
<td>145</td>
<td>40</td>
<td>600</td>
<td>304</td>
<td>244</td>
<td>1333</td>
</tr>
<tr>
<td><strong>cvc4+fmf</strong></td>
<td>145</td>
<td>40</td>
<td>604</td>
<td>294</td>
<td>236</td>
<td>1319</td>
</tr>
<tr>
<td><strong>z3</strong></td>
<td>145</td>
<td>40</td>
<td>604</td>
<td>304</td>
<td>244</td>
<td>1337</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>german</th>
<th>refcount</th>
<th>agree</th>
<th>apg</th>
<th>bmk</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>cvc4</strong></td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td><strong>cvc4+fmf</strong></td>
<td>45</td>
<td>6</td>
<td>62</td>
<td>16</td>
<td>36</td>
<td>165</td>
</tr>
<tr>
<td><strong>z3</strong></td>
<td>45</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>46</td>
</tr>
</tbody>
</table>

- 60 second timeout
Results: TPTP

• 10 second timeout
• 11613 UNSAT benchmarks:
  – z3: 5471 solved
  – cvc4: 4868 solved
  – cvc4+fmf: 2246 solved, but orthogonal
    • 288 solved that cvc4 w/o finite model finding cannot
  – Either cvc4 or cvc4+fmf: 5158 solved
• 1933 SAT benchmarks:
  – z3: 866 solved
  – cvc4+fmf: 920 solved
• Model-Based filtering of instances is essential
Summary

• Finite model finding in CVC4:
  – Finds minimal models for ground constraints
  – Uses exhaustive instantiation to test models
  • Instantiations filtered by model
  – Optionally, uses heuristic instantiation
Conclusions

• Finite Model Finding:
  – Practical approach for SMT + quantifiers
  – Can answer SAT quickly
    • Generate simple counterexamples for DVF
      – Many models in real examples have cardinality 2 or 3
  – Improves coverage in UNSAT cases
    • Increased ability to discharge verification conditions
  – Orthogonal to other approaches
Questions?