A Decision Procedure for Program Analysis and Bug Finding

Vijay Ganesh
Affiliation: CSAIL, MIT
Supported by Lincoln Labs
February 7th, 2008
Motivating Example

```c
Foo(int x){
    int A[2];
    int t;
    A[0] = 0;
    A[1] = 1;
    if(0 <= x <= 1) {
        t = 2/(A[x] + x);
    }
}
```

In Theory, Symbolic Execution + DP + Verification Conditions - Unbounded Loops, Gives Verification
Decision Procedures

- Examples: Boolean SAT, Real Arithmetic, Bit-vectors
- Reduction easy for many problems
- Approach better than coming up with special purpose algorithms:
  - More efficient and saves work
- AI, program analysis, bug finding, verification,…
1. **Design and Architecture of STP** (CAV ‘07, CCS ‘06)

2. **Abstraction-Refinement** based heuristics for Deciding Arrays

3. **Solver Algorithm** for deciding Linear Bit-vector Arithmetic $O(n^3)$

4. **Experimental Results**
Projects using STP

- **Bug Finders**
  - EXE by Dawson Engler, Cristian Cadar and others (Stanford)
  - MINESWEEPER by Dawn Song and her group (CMU)
  - CATCHCONV by David Molnar and David Wagner (Berkeley)
  - Backward Path Sensitive Analysis by Tim Leek (MIT Lincoln)

- **Security Tools**
  - REPLAYER: Security analysis thru protocol replay (CMU)
  - Smart Fuzzer by Roberto Paleari (University of Milan, Italy)

- **Program Analysis**
  - by Rupak Majumdar (UCLA)

- **Hardware verification**
  - Cache coherence protocols by Dill group (Stanford)
  - By a chip company

- **Software verification of crypto algorithms by Dill group (Stanford)**
Projects using STP: Smart Fuzzing thru’ Path Selection

- **Smart Fuzzer by Roberto Paleari (University of Milan, Italy)**
  - Do dynamic analysis to determine dependency between input and control transfer (if conditional)
  - Collect path conditions
  - Feed to STP to find values that drive a path
  - Feed to STP to find values that drive the ‘other’ path
Projects using STP: Formal Verification of Crypto Algorithms

- Eric Smith and David Dill

Technique

- Annotate code with Invariants
- Symbolically execute the Java implementation of the Crypto Algo
- Plug the symbolically executed terms into the invariants
- Feed invariants into ACL2 + STP
- ACL2 handles any induction + integer related stuff, and STP handles (in)equalities over bit-vector terms
Projects using STP:
Cross Checking, Model Checking, Equivalence Checking(?)

- Cross Checking: EXE : Dawson Engler, Cristian Cadar, ...
  - Different implementations of grep... Do they match?
  - Symbolic-simulate Grep1
  - Symbolic-simulate Grep2
  - Equate the two and feed to STP

- Model Checking Cache Coherence Protocols: Chang and Dill
  - Does model satisfy property P?
  - Convert to decision problem and feed to STP
  - If you are using BDDs, try SAT or STP

```
Compiler Optimization/
Verilog Synthesis
```

```f() = g()
```

```
STP
Valid/Invalid
```
Projects using STP:
Work by Dawn Song and her group

- Automatic discovery of deviations in binary implementations: error detection and fingerprint generation

- Protocol Replay: Try to reproduce a dialog between an initiator and a network host
  - Auto Generation of modules for honeypots so that they can correctly respond to connection attempts by worms

- Automatic patch based exploit generation: Using STP to reveal exploit information from a windows patch
Quantifier-free Theory of Bit-vectors and Arrays

\[(x + \text{mem}[i] + 0b10 = 0) \text{ OR } (\text{q}[3:1]*0b01 < 0b00)\]

- Expressions in STP correspond to
  - C/Java... programming language expressions
  - Microprocessor instruction set
  - Arrays represent program memory or array data structure in C/Java...

- Except
  - Our bit-vectors are of any fixed length
  - No floating point
  - No loops

- SAT problem for this theory is NP-complete
Quantifier-free Theory of Bit-vectors and Arrays

\[(x + \text{mem}[i] + 0b10 = 0) \text{ OR } (q[3:1]*0b01 < 0b00)\]

- **Bit-vector Terms**
  - Constants: 0b0011
  - Variables
  - +, -, *, (signed) div, (signed) mod
  - Concatenation, Extraction
  - Left/Right Shift, Sign-extend, bitwise-Booleans
Quantifier-free Theory of Bit-vectors and Arrays

\[(x + \text{mem}[i] + 0b10 = 0) \text{ OR } (q[3:1]*0b01 < 0b00)\]

- **Array Terms**
  - Read (Array, index)
  - Write (Array, index, val)
  - Example: \(R(W(A, i, 0b00), i) = 0b00\)

- **Conditional in programming/multiplexors in hardware**
  - \(\text{ite}(c, t1, t2) = \text{if } (c) \text{ then } t1 \text{ else } t2 \text{ endif}\)

- **Predicates:** =, <=, <=s
Features of STP

- Can handle very large formulas efficiently
  - Large number of array reads ($10^5$)
  - Deeply nested array writes ($10^4$ deep)
  - Very large number of linear equations ($10^6$)
  - Very large number of variables ($10^6$)

- Enabled several software and hardware applications

- Won the SMTCOMP 2006 competition in bit-vector category
STP Architecture

Input Formula

Substitutions

Simplifications

Linear Solving

Array Abstraction

BitBlast

CNF Conversion

Boolean SAT

Refinement Loop

Result
Alternative Architectures

Input Formula

SAT

Simplifier

Result

New Derived Constraints

$\text{DP}_1$ $\text{DP}_2$ $\ldots$ $\text{DP}_n$

Combination:
- NO79, Sho84, RS02
- CVC3 (BB04)
- CVC (SBD02)
- z3 (DeMB07)
- Yices (DeMB05)

Others:
- STP (GD06, GD07)
- UCLID (BS05)
- BAT (M06)
- Cogent (BK05)
Alternative Architectures

Input Formula

SAT

Simplifier

Result

DP₁
DP₂
…
DPₙ

New Derived Constraints

Refinement Loop

Input Formula

Substitutions

Simplifications

Linear Solving

Array Abstraction

BitBlast

CNF Conversion

Boolean SAT
1. Design and Architecture of STP

2. Abstraction-Refinement based heuristics for Deciding Arrays

3. Solver Algorithm for deciding Linear Bit-vector Arithmetic $O(n^3)$

4. Experimental Results
Standard Handling of Array reads

- Problem: $O(n^2)$ axioms added, $n$ is number of read indices
  - Lethal, if $n$ is large: $n = 10000$, # of axioms: ~ 100 million
  - Blowup seems hard to avoid (e.g. UCLID)
- This is “aliasing” from another perspective
- Key Observation: Most indices don’t alias

Replace array reads with fresh variables and axioms

\[
\begin{align*}
    v_0 &= t_0 \\
    v_1 &= t_1 \\
    & \quad \vdots \\
    v_n &= t_n \\
    (i_1 = i_0) & \Rightarrow v_1 = v_0 \\
    (i_2 = i_0) & \Rightarrow v_2 = v_0 \\
    (i_2 = i_1) & \Rightarrow v_2 = v_1 \\
    & \quad \vdots
\end{align*}
\]
Abstraction-Refinement for Array Reads

Input → Array Transform → To SAT Solver without Axioms

Check Input on Assignment

Assignment is Correct

Add False Axioms to SAT Solver

Refinement Loop

Incorrect

SAT → UNSAT

SAT

UNSAT
Abstraction-Refinement for Array Reads

Input:
- \( \text{Read}(A, i) = 0 \)
- \( \text{Read}(A, k) = 1 \)
- \( i = k \)

Abstraction:
- \( v_i = 0 \)
- \( v_k = 1 \)
- \( i = k \)

SAT Solver:
- \( i = 0, k = 0 \)
- \( v_i = 0 \)
- \( v_k = 1 \)

SAT Assignment:
- UNSAT

Refinement Step:
Add Axiom
\( (i = k) \Rightarrow v_i = v_k \)

Check Input on Assignment:
- \( \text{Read}(A, 0) = 0 \)
- \( \text{Read}(A, 0) = 1 \)

False
Experience with Read Abstraction-Refinement

- Heuristic is Robust
  - In Real SAT assignment very few indices aliased
  - Few axioms need to be added during refinement
  - ~10X speed-up
  - Important for software analysis

<table>
<thead>
<tr>
<th></th>
<th>Only Read Refinement (sec)</th>
<th>No Read Refinement (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time for all tests</td>
<td>624</td>
<td>3378</td>
</tr>
<tr>
<td># of timeouts (60 sec)</td>
<td>1</td>
<td>36</td>
</tr>
</tbody>
</table>

# of Tests: 8495

3.2 GHz Pentium, 512Kb Cache, 32 bit machine
Examples courtesy Dawson Engler
**Standard Handling of Array Writes**

![Mathematical expression]

### Key Observation
- Not all read indices read from write term

#### Sharing of sub-expression in DAG
- Array Writes are deeply nested, shared over many reads
- Problem: Standard translation breaks sharing & blowup
  - \( O(n^*m) \) blowup, \( n = \) # of levels of writes, \( m = \) # of reads
  - \( n = 10,000, m = 1000 \) : blow-up ~ 10 million new nodes

---

\[ R(W(W(A,i_0,v_0),i_1,v_1),j) = R(W(W(A,i_0,v_0),i_1,v_1),k) \]

\[ \text{If}(i_1=j) \ v_1 \text{ elsif } (i_0=j) \ v_0 \text{ else } R(A,j) = \text{If}(i_1=k) \ v_1 \text{ elsif } (i_0=k) \ v_0 \text{ else } R(A,k) \]
The Problem with Array Writes

\[
R(W(W(A,i_0,v_0),i_1,v_1),j) = R(W(W(A,i_0,v_0),i_1,v_1),k)
\]

\[
\text{If}(i_1=j) v_1 \text{ elsif } (i_0=j) v_0 \text{ else } R(A,j) = \text{If}(i_1=k) v_1 \text{ elsif } (i_0=k) v_0 \text{ else } R(A,k)
\]
Handling of Array Writes in STP

R(W(W(A,i_0,v_0),i_1,v_1),j) = R(W(W(A,i_0,v_0),i_1,v_1),k)

t_1 = t_2

Axioms:
\[ t_1 = \text{ite}(i_1 = j, v_1, \text{ite}(i_0 = j, v_0, R(A,j))) \]
\[ t_2 = \text{ite}(i_1 = k, v_1, \text{ite}(i_0 = k, v_0, R(A,k))) \]

- Avoids \( O(n^2) \) DAG blow-up
- Axioms are added only on a need basis
- Unfortunately, worst-case all axioms added
Abstraction-Refinement for Array Writes

Input → Array Transform → To SAT Solver without Axioms

Check Input on Assignment

SAT → Assignment is Correct

UNSAT → Incorrect

Add False Axioms to SAT Solver

SAT → UNSAT

UNSAT → Assignment is Correct
Abstraction-Refinement for Array Writes

Abstraction:
\[
R(W(A, i, v), j) = 0 \\
R(W(A, i, v), k) = 1 \\
i = j \neq k, v \neq 0
\]

SAT Solver:
\[
t_1 = 0 \\
t_2 = 1 \\
i = j \neq k, v \neq 0
\]

Refinement Step:
\[
t_1 = \text{ite}(i = j, v, R(A, j))
\]

Check Input on Assignment:
\[
R(W(A, 0, v), 0) = 0 \\
v = 1 \\
i = j = 0, k = 1
\]

False

UNSAT
## Experimental Results

### Array Writes

<table>
<thead>
<tr>
<th>Testcase (# of unique nodes)</th>
<th>Result</th>
<th>Write Abstraction (sec)</th>
<th>NO Write Abstraction (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>610dd9dc (15k)</td>
<td>Sat</td>
<td>37</td>
<td>101</td>
</tr>
<tr>
<td>Grep0084 (69K)</td>
<td>Sat</td>
<td>18</td>
<td>506</td>
</tr>
<tr>
<td>Grep0106 (69K)</td>
<td>Sat</td>
<td>227</td>
<td>TO</td>
</tr>
<tr>
<td>Grep0117 (70K)</td>
<td>Sat</td>
<td>258</td>
<td>TO</td>
</tr>
<tr>
<td>Testcase20 (1.2M)</td>
<td>Sat</td>
<td>56</td>
<td>MO</td>
</tr>
</tbody>
</table>

3.2 GHz Pentium, 512 Kb cache, 32 bit machine, MO @ 3.2 GB, TO @ 30 minutes

Examples courtesy Dawn Song (CMU) and David Molnar (Berkeley)
1. **Design and Architecture of STP**

2. **Abstraction-Refinement based heuristics for Deciding Arrays**

3. **Solver Algorithm for deciding Linear Bit-vector Arithmetic O(n^3)**

4. **Experimental Results**
Algorithm for Solving Linear Bit-vector Equations

- Previous Work
  - Mostly Variants of Gaussian Elimination
    - Unsuitable for Online Decision Procedures

- Basic Idea in STP
  - Solve for a variable and substitute it away

- Online Algorithm
  - Enables other algebraic simplifications

- If cannot isolate a whole variable,
  - Then isolate part of bit-vector variable,
  - Solve, and substitute it away
Purpose of Linear Solver

- Helps eliminate lots of redundant variables
- Makes problem much easier for SAT
- Essential for many real-word large examples
Importance of Online Linear Solver

Online Solving enables algebraic Simplifications

Input Formula
- Substitutions
- Simplifications
- Linear Solving
- Array Abstraction
- BitBlast
- CNF Conversion
- Boolean SAT

Refinement Loop

Result
Algorithm for Solving Linear Bit-vector Equations

\[
\begin{align*}
(\text{mod } 8) \\
3x + 4y + 2z &= 0 \\
2x + 2y + 2z &= 0 \\
4y + 2x + 2z &= 0
\end{align*}
\]

Isolate 3x in first equation:
Multiplicative Inverse exists, Solve for x

x = 4y + 2z

\[
\begin{align*}
(\text{mod } 8) \\
2y + 4z + 2 &= 0 \\
4y + 6z &= 0
\end{align*}
\]
Algorithm for Solving Linear Bit-vector Equations

(mod 8)
\[ 2y + 4z + 2 = 0 \]
\[ 4y + 6z = 0 \]

All Coeffs Even
No Inverse

(mod 4)
\[ y[1:0] + 2z[1:0] + 1 = 0 \]
\[ 2y[1:0] + 3z[1:0] = 0 \]

Key Idea: Solve for bits of variables

Divide by 2
Algorithm for Solving Linear Bit-vector Equations

\begin{align*}
(y[1:0] + 2z[1:0] + 1) \mod 4 &= 0 \\
2y[1:0] + 3z[1:0] &= 0
\end{align*}

Solve for $y[1:0]$

\begin{align*}
y[1:0] &= 2z[1:0] + 3 \\
3z[1:0] + 2 &= 0
\end{align*}

Substitute $y[1:0]$
Algorithm for Solving Linear Bit-vector Equations

\[(\text{mod 4})\]
\[3z[1:0] + 2 = 0\]

Solve for \(z[1:0]\)

Solution (mod8, 3 bits)
\[x = 4(y' @ 3) + 2(z' @ 2)\]
\[y = y' @ 3\]
\[y[1:0] = 3\]
\[z = z' @ 2\]
\[z[1:0] = 2\]

\[(\text{mod 4})\]
\[z[1:0] = 2\]
## Experimental Results: Solver for Linear Equations

<table>
<thead>
<tr>
<th>Testcase</th>
<th>Result</th>
<th>Solver On (sec)</th>
<th>Solver Off (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test15 (0.9M)</td>
<td>Sat</td>
<td>66</td>
<td>MO</td>
</tr>
<tr>
<td>Test16 (0.9M)</td>
<td>Sat</td>
<td>67</td>
<td>MO</td>
</tr>
<tr>
<td>Thumb1 (2.7M)</td>
<td>Sat</td>
<td>840</td>
<td>MO</td>
</tr>
<tr>
<td>Thumb2 (3.2M)</td>
<td>Sat</td>
<td>115</td>
<td>MO</td>
</tr>
<tr>
<td>Thumb3 (4.3M)</td>
<td>Sat</td>
<td>1920</td>
<td>MO</td>
</tr>
</tbody>
</table>

3.2 GHz Pentium, 512 Kb cache, 32 bit machine, MO @ 3.2 GB, TO @ 35 minutes

Examples courtesy David Molnar (Berkeley)
1. **Design and Architecture of STP**

2. **Abstraction-Refinement based heuristics for Deciding Arrays**

3. **Solver Algorithm for deciding Linear Bit-vector Arithmetic O(n^3)**

4. **Experimental Results**
## STP v. Existing Tools
**(Hardest Examples: SMT Comp, 2007)**

<table>
<thead>
<tr>
<th>Testcase (# of Unique Nodes)</th>
<th>Result</th>
<th>STP (sec)</th>
<th>Z3 (sec)</th>
<th>Yices (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>610dd9c (15k)</td>
<td>Sat</td>
<td>37</td>
<td>TO</td>
<td>MO</td>
</tr>
<tr>
<td>Grep65 (60k)</td>
<td>UnSat</td>
<td>4</td>
<td>0.3</td>
<td>TO</td>
</tr>
<tr>
<td>Grep84 (69k)</td>
<td>Sat</td>
<td>18</td>
<td>176</td>
<td>TO</td>
</tr>
<tr>
<td>Grep106 (69k)</td>
<td>Sat</td>
<td>227</td>
<td>130</td>
<td>TO</td>
</tr>
<tr>
<td>Blaster4 (262k)</td>
<td>UnSat</td>
<td>10</td>
<td>MO</td>
<td>MO</td>
</tr>
<tr>
<td>Testcase20 (1.2M)</td>
<td>Sat</td>
<td>56</td>
<td>MO</td>
<td>MO</td>
</tr>
<tr>
<td>Testcase21 (1.2M)</td>
<td>Sat</td>
<td>43</td>
<td>MO</td>
<td>MO</td>
</tr>
</tbody>
</table>

3.2 GHz Pentium, 512 Kb cache, 32 bit machine, MO @ 3.2 GB, TO @ 35 minutes

Examples courtesy Dawn Song (CMU) and David Molnar (Berkeley)
Lessons Learnt

- Abstraction Refinement will remain important for DPs for many applications
- Reduction to Boolean SAT
- Identify polynomial pieces and nail them
- Successful DPs highly application driven
Future Work

- Make STP more efficient for
  - Disjunctions
  - Non-linear Arithmetic (\(*\), \(/\), \%)  
- Quantifiers
- Boolean SAT tuning for structured input
- More theories
  - Uninterpreted Functions, Datatypes, Reals, Integers, …
Other Projects at Stanford

- Software
  - CVC
    - Decision Procedure for Mixed Real and Integer Linear Arithmetic
  - CVC Lite
    - Decision Procedure for Bit-vectors
  - Collaborated on EXE
    - STP, Capturing C semantics in STP

- Theory
  - Lifted Ghilardi’s Combination Result to Many-Sorted Logic
Acknowledgements

- Prof. David L. Dill, Stanford CS Department (Ph.D. Advisor)
- STP users and Stanford community
- Prof. Martin Rinard (Host)
- Lincoln Labs and Tim Leek (Support)
QUESTIONS

http://people.csail.mit.edu/vganesh/stp.html