Applications and Challenges in Satisfiability Modulo Theories

WING/ETAPS 2009 – York

Leonardo de Moura
Microsoft Research
Verification/Analysis tools need some form of Symbolic Reasoning
Applications and Challenges in Satisfiability Modulo Theories

Applications

- Test case generation
- Verifying Compilers
- Predicate Abstraction
- Invariant Generation
- Type Checking
- Model Based Testing
Some Applications @ Microsoft

Applications and Challenges in Satisfiability Modulo Theories
unsigned GCD(x, y) {

  requires(y > 0);

  while (true) {
    unsigned m = x % y;
    if (m == 0) return y;
    x = y;
    y = m;
  }
}

(y_0 > 0) and
(m_0 = x_0 \% y_0) and
not (m_0 = 0) and
(x_1 = y_0) and
(y_1 = m_0) and
(m_1 = x_1 \% y_1) and
(m_1 = 0)

x_0 = 2
y_0 = 4
m_0 = 2
x_1 = 4
y_1 = 2
m_1 = 0

We want a trace where the loop is executed twice.
Is formula $F$ satisfiable modulo theory $T$?

SMT solvers have specialized algorithms for $T$. 

Applications and Challenges in Satisfiability Modulo Theories
$b + 2 = c \quad \text{and} \quad f(\text{read}(\text{write}(a,b,3), c-2) \neq f(c-b+1)$
Arithmetic

\[ b + 2 = c \quad \text{and} \quad f(\text{read}(\text{write}(a,b,3), c-2)) \neq f(c-b+1) \]
Satisfiability Modulo Theories (SMT)

\[ b + 2 = c \quad \text{and} \quad f(\text{read}(\text{write}(a,b,3), c-2)) \neq f(c-b+1) \]

Array Theory
$b + 2 = c$ and $f(read(write(a,b,3), c-2)) \neq f(c-b+1)$
A Theory is a set of sentences

Alternative definition:

A Theory is a class of structures
Z3 is a new solver developed at Microsoft Research.
Development/Research driven by internal customers.
Free for academic research.
Interfaces:

- C/C++
- .NET
- OCaml
- Text

http://research.microsoft.com/projects/z3
For some theories, SMT can be reduced to SAT

Higher level of abstraction

$bvmul_{32}(a,b) = bvmul_{32}(b,a)$
For most SMT solvers: \( F \) is a set of ground formulas

Many Applications

Bounded Model Checking

Test-Case Generation
Applications and Challenges in Satisfiability Modulo Theories
Guessing (case-splitting)

\[ p \quad \mid \quad p \lor q, \neg q \lor r \]

\[ p, \neg q \quad \mid \quad p \lor q, \neg q \lor r \]
Deducing

\[ p \mid p \lor q, \lnot p \lor s \]

\[ p, s \mid p \lor q, \lnot p \lor s \]
Backtracking

\[ p, \neg s, q \mid p \lor q, s \lor q, \neg p \lor \neg q \]

\[ p, s \mid p \lor q, s \lor q, \neg p \lor \neg q \]
Efficient indexing (two-watch literal)
Non-chronological backtracking (backjumping)
Lemma learning
...

Applications and Challenges in Satisfiability Modulo Theories
Efficient decision procedures for conjunctions of ground atoms.

\[ a = b, \ a < 5 \ \mid \ \neg a = b \lor f(a) = f(b), \ a < 5 \lor a > 10 \]

Efficient algorithms

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Difference Logic</td>
<td>Belmann-Ford</td>
</tr>
<tr>
<td>Uninterpreted functions</td>
<td>Congruence closure</td>
</tr>
<tr>
<td>Linear arithmetic</td>
<td>Simplex</td>
</tr>
</tbody>
</table>
a = b, a > 0, c > 0, a + c < 0 | F

backtrack
Naïve recipe?

SMT Solver = DPLL + Decision Procedure

Standard question:
Why don’t you use CPLEX for handling linear arithmetic?
Efficient SMT solvers

Decision Procedures must be:
Incremental & Backtracking
Theory Propagation

\[ a=b, \ a<5 \ | \ ... \ a<6 \lor f(a) = a \]

\[ a=b, \ a<5, \ a<6 \ | \ ... \ a<6 \lor f(a) = a \]
Decision Procedures must be:

- Incremental & Backtracking
- Theory Propagation
- Precise (theory) lemma learning

Given:

\[ a = b, \ a > 0, \ c > 0, \ a + c < 0 \ | \ F \]

Learn clause:

\[ \neg(a=b) \lor \neg(a > 0) \lor \neg(c > 0) \lor \neg(a + c < 0) \]

Imprecise!

Precise clause:

\[ \neg a > 0 \lor \neg c > 0 \lor \neg a + c < 0 \]
Verifying Compilers

Annotated Program $\rightarrow$ Verification Condition $F$

pre/post conditions
invariants
and other annotations
class C {
    private int a, z;
    invariant z > 0

    public void M() {
        requires a != 0
        {
            z = 100/a;
        }
    }
}
Modeling execution traces

terminates

diverges

goes wrong
States and execution traces

State
- Cartesian product of variables

Execution trace
- Nonempty finite sequence of states
- Infinite sequence of states
- Nonempty finite sequence of states followed by special error state

(x: int, y: int, z: bool)
Command language

- \( x := E \)
  - \( x := x + 1 \)
  - \( x := 10 \)
- havoc \( x \)
- assert \( P \)
- assume \( P \)
- \( \neg P \)
Command language

- \( x := E \)
  - \( x := x + 1 \)
  - \( x := 10 \)
- \( \text{havoc } x \)
- \( S ; T \)

- \( \text{assert } P \)
- \( \text{assume } P \)
- \( \neg P \)

P

...
Command language

- $x := E$
  - $x := x + 1$
  - $x := 10$
- $\text{havoc } x$
- $S ; T$
- $\text{assert } P$
- $\text{assume } P$
- $S \square T$
Reasoning about execution traces

Hoare triple \( \{ P \} S \{ Q \} \) says that every terminating execution trace of S that starts in a state satisfying P does not go wrong, and terminates in a state satisfying Q.
Hoare triple \{ P \} S \{ Q \} says that every terminating execution trace of S that starts in a state satisfying P does not go wrong, and terminates in a state satisfying Q. Given S and Q, what is the weakest P' satisfying \{ P' \} S \{ Q \}? P' is called the weakest precondition of S with respect to Q, written \textit{wp}(S, Q) to check \{ P \} S \{ Q \}, check P \Rightarrow P'
Weakest preconditions

- \( \text{wp}( x := E, Q ) = Q[E/x] \)
- \( \text{wp}( \text{havoc} x, Q ) = (\forall x \cdot Q) \)
- \( \text{wp}( \text{assert} P, Q ) = P \land Q \)
- \( \text{wp}( \text{assume} P, Q ) = P \implies Q \)
- \( \text{wp}( S ; T, Q ) = \text{wp}( S, \text{wp}( T, Q )) \)
- \( \text{wp}( S \Box T, Q ) = \text{wp}( S, Q ) \land \text{wp}( T, Q ) \)
Structured if statement

if E then S else T end =

assume E; S

assume \neg E; T
Dijkstra's guarded command

if  E \rightarrow S  \mid  F \rightarrow T \ fi  =

assert E \lor F;

(  
assume E;  S

assume F;  T
)


While loop with loop invariant

while E
    invariant J
do
    S
end

where x denotes the assignment targets of S

= assert J;
  havoc x; assume J;
  ( assume E; S; assert J; assume false
  □ assume ¬E
  )

check that the loop invariant holds initially

“fast forward” to an arbitrary iteration of the loop

check that the loop invariant is maintained by the loop body
Verification conditions: Structure

∀ Axioms (non-ground)

Control & Data Flow

+ BIG and-or tree (ground)
**Meta OS**: small layer of software between hardware and OS

**Mini**: 60K lines of non-trivial concurrent systems C code

**Critical**: must provide functional resource abstraction

**Trusted**: a verification grand challenge
VCs have several Mb
Thousands of non ground clauses
Developers are willing to wait at most 5 min per VC
Partial solutions

- Automatic generation of: Loop Invariants
- Houdini-style automatic annotation generation
Challenge

- Quantifiers, quantifiers, quantifiers, ...
- Modeling the runtime

∀ h,o,f:

\[ \text{IsHeap}(h) \land o \neq \text{null} \land \text{read}(h, o, \text{alloc}) = t \]

⇒

\[ \text{read}(h,o,f) = \text{null} \lor \text{read}(h, \text{read}(h,o,f),\text{alloc}) = t \]
Quantifiers, quantifiers, quantifiers, ...

Modeling the runtime

Frame axioms

\( \forall o, f:\)

\( o \neq \text{null} \land \text{read}(h_0, o, \text{alloc}) = t \Rightarrow\)

\( \text{read}(h_1, o, f) = \text{read}(h_0, o, f) \lor (o, f) \in M \)
Challenge

- Quantifiers, quantifiers, quantifiers, ...
- Modeling the runtime
- Frame axioms
- User provided assertions
  \[ \forall i, j : i \leq j \Rightarrow \text{read}(a, i) \leq \text{read}(b, j) \]
Challenge

- Quantifiers, quantifiers, quantifiers, ...
- Modeling the runtime
- Frame axioms
- User provided assertions

Theories

- $\forall x: p(x,x)$
- $\forall x,y,z: p(x,y), p(y,z) \Rightarrow p(x,z)$
- $\forall x,y: p(x,y), p(y,x) \Rightarrow x = y$
Quantifiers, quantifiers, quantifiers, ...
Modeling the runtime
Frame axioms
User provided assertions
Theories
Solver must be fast in satisfiable instances.

We want to find bugs!
There is no sound and refutationally complete procedure for linear integer arithmetic + free function symbols
Many Approaches

- Heuristic quantifier instantiation
- Combining SMT with Saturation provers
- Complete quantifier instantiation
- Decidable fragments
- Model based quantifier instantiation
SMT solvers use **heuristic quantifier instantiation**.

**E-matching** (matching modulo equalities).

**Example:**

$$\forall \ x : f(g(x)) = x \ \{ f(g(x)) \}$$

- $a = g(b)$,
- $b = c$,
- $f(a) \neq c$
SMT solvers use heuristic quantifier instantiation.

E-matching (matching modulo equality).

Example:
\[ \forall x : f(g(x)) = x \{ f(g(x)) \} \]
\[ a = g(b), \]
\[ b = c, \]
\[ f(a) \neq c \]

Equalities and ground terms come from the partial model \( M \)
E-matching: why do we use it?

- Integrates smoothly with DPLL.
- Efficient for most VCs
- Decides useful theories:
  - Arrays
  - Partial orders
  - ...

Applications and Challenges in Satisfiability Modulo Theories
E-matching is NP-Hard.

In practice

<table>
<thead>
<tr>
<th>Problem</th>
<th>Indexing Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast retrieval</td>
<td>E-matching code trees</td>
</tr>
<tr>
<td>Incremental E-Matching</td>
<td>Inverted path index</td>
</tr>
</tbody>
</table>

Applications and Challenges in Satisfiability Modulo Theories
E-matching code trees

Trigger:
\[ f(x_1, g(x_1, a), h(x_2), b) \]

Instructions:
1. init(f, 2)
2. check(r4, b, 3)
3. bind(r2, g, r5, 4)
4. compare(r1, r5, 5)
5. check(r6, a, 6)
6. bind(r3, h, r7, 7)
7. yield(r1, r7)

Similar triggers share several instructions.

Combine code sequences in a code tree

Applications and Challenges in Satisfiability Modulo Theories
Challenge: modeling runtime

- Is the axiomatization of the runtime consistent?
  - False implies everything
  - E-matching doesn’t work
    - No ground terms to instantiate clauses
- Partial solution: SMT + Saturation Provers
- Found many bugs using this approach
Tight integration: DPLL + Saturation solver.
Inference rule:

\[
\frac{C_1 \ldots C_n}{C}
\]

DPLL(\(\Gamma\)) is parametric.

Examples:
- Resolution
- Superposition calculus
- ...
Challenge: Robustness

- Standard complain
  “I made a small modification in my Spec, and Z3 is timing out”
- This also happens with SAT solvers (NP-complete)
- In our case, the problems are undecidable
- Partial solution: parallelization
Joint work with Y. Hamadi (MSRC) and C. Wintersteiger
Multi-core & Multi-node (HPC)
Different strategies in parallel
Collaborate exchanging lemmas
Non-linear arithmetic is necessary for verifying embedded and hybrid systems.

Non-linear integer arithmetic is undecidable.

Many approaches for non-linear real arithmetic:
- Cylindrical Algebraic Decomposition
  - Doubly exponential procedure
- Grobner Basis + “extensions”
- Heuristics
Predicate Abstraction & Invariant Generation
Overview

- [http://research.microsoft.com/slam/](http://research.microsoft.com/slam/)
- **SLAM/SDV** is a software model checker.
- Application domain: *device drivers*.
- Architecture:
  - **c2bp** C program → boolean program (*predicate abstraction*).
  - **bebop** Model checker for boolean programs.
  - **newton** Model refinement (check for path feasibility)
- SMT solvers are used to perform predicate abstraction and to check path feasibility.
- **c2bp** makes several calls to the SMT solver. The formulas are relatively small.
Given a C program $P$ and $F = \{p_1, \ldots, p_n\}$.

Produce a Boolean program $B(P, F)$
- Same control flow structure as $P$.
- Boolean variables $\{b_1, \ldots, b_n\}$ to match $\{p_1, \ldots, p_n\}$.
- Properties true in $B(P, F)$ are true in $P$.

Each $p_i$ is a pure Boolean expression.
Each $p_i$ represents set of states for which $p_i$ is true.
Performs modular abstraction.
Abstracting Expressions via $F$

- $\text{Implies}_F (e)$
  - Best Boolean function over $F$ that implies $e$.

- $\text{ImpliedBy}_F (e)$
  - Best Boolean function over $F$ that is implied by $e$.
  - $\text{ImpliedBy}_F (e) = \text{not} \ \text{Implies}_F (\text{not} \ e)$
Implies_{F}(e) and ImpliedBy_{F}(e)
Computing $\text{Implies}_F(e)$

- minterm $m = l_1 \land \ldots \land l_n$, where $l_i = p_i$, or $l_i = \text{not } p_i$.
- $\text{Implies}_F(e)$: disjunction of all minterms that imply $e$.
- Naive approach
  - Generate all $2^n$ possible minterms.
  - For each minterm $m$, use SMT solver to check validity of $m \Rightarrow e$.
- Many possible optimizations
Computing $\text{Implies}_F(e)$

- $F = \{ x < y, x = 2 \}$
- $e : y > 1$
- **Minterms over F**
  - $!x<y, !x=2$ implies $y>1$
  - $x<y, !x=2$ implies $y>1$
  - $!x<y, x=2$ implies $y>1$
  - $x<y, x=2$ implies $y>1$

$\text{Implies}_F(y>1) = b_1 y \land b_2 = 2$
Challenge: Rich API

- All-SAT
  - Better (more precise) Predicate Abstraction
- Unsatisfiable cores
  - Why the abstract path is not feasible?
- Fast Predicate Abstraction
Let $S$ be an unsatisfiable set of formulas.

$S' \subseteq S$ is an **unsatisfiable core** of $S$ if:
- $S'$ is also unsatisfiable, and
- There is not $S'' \subset S'$ that is also unsatisfiable.

Computing $\text{Implies}_F(e)$ with $F = \{p_1, p_2, p_3, p_4\}$
- Assume $p_1, p_2, p_3, p_4 \Rightarrow e$ is valid
- That is $p_1, p_2, p_3, p_4, \neg e$ is unsat
- Now assume $p_1, p_3, \neg e$ is the **unsatisfiable core**
- Then it is unnecessary to check:
  - $p_1, \neg p_2, p_3, p_4 \Rightarrow e$
  - $p_1, \neg p_2, p_3, \neg p_4 \Rightarrow e$
  - $p_1, p_2, p_3, \neg p_4 \Rightarrow e$
How to find loop invariant $I$?
Template based approach

- $I$ is a Boolean combination of $F = \{p_1, \ldots, p_n\}$
- Unknown invariant on the LHS constraints how weak $I$ can be
  \[ I(x) \land \neg c(x) \Rightarrow Post(x) \quad I(x) \Rightarrow \neg c(x) \Rightarrow Post(x) \]
- Unknown invariant on the RHS constraints how strong $I$ can be
  \[ \Theta(x) \Rightarrow I(x) \]
- More details: Constraint-based Invariant Inference over Predicate Abstraction, S. Gulwani et al, VMCAI 2009

Applications and Challenges in Satisfiability Modulo Theories
Bit-precise test case generation
unsigned GCD(x, y) {
  requires(y > 0);
  while (true) {
    unsigned m = x % y;
    if (m == 0) return y;
    x = y;
    y = m;
  }
}

(y₀ > 0) and
(m₀ = x₀ % y₀) and
not (m₀ = 0) and
(x₁ = y₀) and
(y₁ = m₀) and
(m₁ = x₁ % y₁) and
(m₁ = 0)

x₀ = 2
y₀ = 4
m₀ = 2
x₁ = 4
y₁ = 2
m₁ = 0
Most solvers use bit-blasting

\[ \text{bvmul}_{32}(a,b) \] is converted into a multiplier circuit

Solvers may run out of memory

“Smart” algorithms are usually less efficient than bit-blasting
I’m unaware of any SMT solver for floating point arithmetic

Approximate using Reals

Unsound!

Incomplete!
Logic as a platform
Most verification/analysis tools need symbolic reasoning
SMT is a hot area
Many applications & challenges
http://research.microsoft.com/projects/z3

Thank You!