02913
Introduction to SMT
Tuesday 3\textsuperscript{rd} January 2012

Flemming Nielson
Literature

SMT and SAT

- SAT: satisfiability of propositional formulae
- SMT: satisfiability modulo theories
  - the propositional variables (up to $10^6$) are really queries to an underlying theory
  - a theory solver is supposed to determine satisfiability of a conjunction of literals (variables or negated variables)
  - the SAT solver is responsible for the search (of which conjunct to choose in the equivalent DNF)
Example: Scheduling

• precedence:
  – complete a phase before starting the next
  – arbitrary delays between phases allowed

• resources:
  – each having a limited capacity

• example:

<table>
<thead>
<tr>
<th>$d_{ij}$</th>
<th>Machine 1</th>
<th>Machine 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job 1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Job 2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Job 3</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

max = 8

Solution

$t_{1,1} = 5, \quad t_{1,2} = 7, \quad t_{2,1} = 2, \quad t_{2,2} = 6, \quad t_{3,1} = 0, \quad t_{3,2} = 3$
Example: Translation to SMT

• Encoding using Difference Arithmetic
  – precedence constraints

\[
\begin{align*}
(t_{1,1} \geq 0) & \land (t_{1,2} \geq t_{1,1} + 2) & \land (t_{1,2} + 1 \leq 8) \\
(t_{2,1} \geq 0) & \land (t_{2,2} \geq t_{2,1} + 3) & \land (t_{2,2} + 1 \leq 8) \\
(t_{3,1} \geq 0) & \land (t_{3,2} \geq t_{3,1} + 2) & \land (t_{3,2} + 3 \leq 8)
\end{align*}
\]

  – resource constraints

\[
\begin{align*}
((t_{1,1} \geq t_{2,1} + 3) & \lor (t_{2,1} \geq t_{1,1} + 2)) \\
((t_{1,1} \geq t_{3,1} + 2) & \lor (t_{3,1} \geq t_{1,1} + 2)) \\
((t_{2,1} \geq t_{3,1} + 2) & \lor (t_{3,1} \geq t_{2,1} + 3)) \\
((t_{1,2} \geq t_{2,2} + 1) & \lor (t_{2,2} \geq t_{1,2} + 1)) \\
((t_{1,2} \geq t_{3,2} + 3) & \lor (t_{3,2} \geq t_{1,2} + 1)) \\
((t_{2,2} \geq t_{3,2} + 3) & \lor (t_{3,2} \geq t_{2,2} + 1))
\end{align*}
\]
Theory: Difference Arithmetic

- Difference Arithmetic
  - fragment of Linear Arithmetic
  - only variable – variable ≤ constant
  - trick for variable ≤ constant (use variable for zero)
- Theory Solver for Difference Arithmetic
  - Construct graph
    - variables are nodes
    - constraint v1 – v2 ≤ c is edge labelled c from v2 to v1
  - Solvable iff no cycle with negative sum of labels
Theory: Difference Arithmetic

• Example Graph

\[
\begin{align*}
    z - t_{1,1} & \leq 0 \\
    z - t_{2,1} & \leq 0 \\
    z - t_{3,1} & \leq 0 \\
    t_{3,2} - z & \leq 5 \\
    t_{3,1} - t_{3,2} & \leq -2 \\
    t_{2,1} - t_{3,1} & \leq -3 \\
    t_{1,1} - t_{2,1} & \leq -2
\end{align*}
\]

• Negative Cycle
SMT operation

• SAT solver with Theory Solver as subroutine
  – SAT proceeds to find satisfying assignments
    • if not satisfiable we are done (no solution)
  – these are checked by Theory Solver
    • if true we are done (found a solution)
    • if false
      – the negation of the assignment provides a theory lemma
      – the theory lemma is added as a new conjunct in the CNF
  – Correctness and Termination
  – Optimization
    • find minimal part of assignment that is not true
Example: Theory Lemma

• negative cycle

• gives theory lemma

\[ \neg (t_{3,1} - t_{3,2} \leq -2) \lor \neg (t_{2,1} - t_{3,1} \leq -3) \lor \neg (t_{1,1} - t_{2,1} \leq -2) \lor \neg (z - t_{1,1} \leq 0) \lor \neg (t_{3,2} - z \leq 5) \]
SMT improvements

• SAT part
  – (we showed lazy off-line integration)
  – online integration: also check partial assignments
  – theory propagation: given partial assignment, difference arithmetic may induce transitive edge, which might be variable that is now implied

• Theory Solver part
  – incremental: cheap to add constraints
  – backtrackable: cheap to remove constraints
Applications of SMT

• Dynamic Symbolic Execution
  – find test inputs for reaching certain paths / bugs
  – consider SSA representation of some chosen finite straight line path through program

```c
int GCD (int x, int y) {
  while (true) {
    int m = x % y;
    if (m == 0) return y;
    x = y;
    y = m;
  }
}
```

```c
int GCD (int x₀, int y₀) {
  int m₀ = x₀ % y₀;
  assert (m₀ != 0);
  int x₁ = y₀;
  int y₁ = m₀;
  int m₁ = x₁ % y₁;
  assert (m₁ == 0);
}
```
Applications of SMT

- **Dynamic Symbolic Execution (cont.)**
  - create assertions enforcing this execution

```c
int GCD (int x_0, int y_0) {
  int m_0 = x_0 % y_0;
  assert (m_0 != 0);
  int x_1 = y_0;
  int y_1 = m_0;
  int m_1 = x_1 % y_1;
  assert (m_1 == 0);
  if (m_0 == y_0)
    return m_0;
  if (m_0 == 0)
    return x_0;
  if (m_0 == x_0 % y_0)
    return GCD(x_0, y_0);
  if (m_1 == 0)
    return m_0;
  return GCD(m_1, y_1);
}
```

- satisfying assignment gives required input
  \[ x_0 = 2, y_0 = 4, m_0 = 2, x_1 = 4, y_1 = 2, m_1 = 0 \]
- white-box fuzzing = fuzz (random input) + dynamic symbolic execution
Applications of SMT

- Program Model Checking / Static Analysis
  – prove the absence of bugs using abstraction and non-determinism
  – example: abstract “old_count = count” as $b$

```c
do {
  lock ();
  old_count = count;
  request = GetNextRequest();
  if (request != NULL) {
    unlock();
    ProcessRequest(request);
    count = count + 1;
  }
}
while (old_count != count);
unlock();
```

```c
do {
  lock ();
  b = true;
  request = GetNextRequest();
  if (request != NULL) {
    unlock();
    ProcessRequest(request);
    if (b) b = false; else b = *
  }
}
while (!b);
unlock();
```
Applications of SMT

• Program Model Checking / Static Analysis (cont.)
  – the transfer functions of the abstract model are proved as theorems in a SMT solver
  – we can now prove $b$ is true when unlock reached showing lock are balanced with unlock
    • prove using finite state model checker
    • prove using static analysis
  – Why SAT plus model check / static analysis?
Applications of SMT

- Program Model Checking / Static Analysis (cont.)
  - You need to be faithful to 32 bit arithmetic

```c
int binary_search(
    int[] arr, int low, int high, int key) {
    assert (low > high || 0 <= low < high);
    while (low <= high) {
        //Find middle value
        int mid = (low + high)/2;
        assert (0 <= mid < high);
        int val = arr[mid];
        //Refine range
        if (key == val) return mid;
        if (val > key) low = mid+1;
        else high = mid-1;
    }
    return -1;
}
```
Applications of SMT

• Extended Static Checking
  – Floyd Hoare triples with formulae in SMT
  – Application to Microsoft 10^5 line Viridian Hyper-Visor OS requiring 60 man years
  – Free theory of single inheritance used for arrays in Java and C# for determining inheritance:

\[
\begin{align*}
(\forall x: \text{sub}(x, x)) \\
(\forall x,y,z: \text{sub}(x, y) \land \text{sub}(y, z) \rightarrow \text{sub}(x, z)) \\
(\forall x,y: \text{sub}(x, y) \land \text{sub}(y, x) \rightarrow x = y) \\
(\forall x,y,z: \text{sub}(x, y) \land \text{sub}(x, z) \rightarrow \text{sub}(y, z) \lor \text{sub}(z, y)) \\
(\forall x,y: \text{sub}(x, y) \rightarrow \text{sub}(\text{array-of}(x), \text{array-of}(y)))
\end{align*}
\]
Theory: Free Functions

- Theory: Free Functions with Equality
  - free in the sense of free algebra
- Solver:
  - Construct DAG of terms
    - equalities indicate nodes to be shared
    - share nodes with same label and descendants
  - Check for absence of self-inequalities
    - unsatisfiable iff any such exist
Theory: Free Functions

- Example: \( a = b, b = c, f(a, g(a)) \neq f(b, g(c)) \)
Applications of SMT

• Software Modeling
  – modelling access and modification to arrays (McCarthy 1962)
  – model based design
    • bounded model checking on symbolic design
    • implementation through refinement
    • allows design with late choice of design parameters
Combining Theory Solvers

• Using Nelson Oppen
  – theories only share the equality (disjointness)
  – theories are stably infinite (do not require finite universe for quantifier free formula to be satisfied)

• Using Convexity
  – if $\land S \Rightarrow \lor E$ then $E e \in E: \land S \Rightarrow e$
  – polynomial time combinations exist

• Using Delayed Theory Combination
  – for guessing equalities between shared variables between theories
SMT: questions for today

Explain
① efficient indexing techniques for SAT (p.71)
② difference arithmetic: unsatisfiable iff neg.cycle (p.71)
③ incrementality and backtrackability (p.72)
④ “program model checking” (p.73)
  □ why use *, why not have old_count ≤ count
⑤ details of the Nelson and Oppen method (p.76)
⑥ details of convexity (p.76)
⑦ details of delayed theory combination (p.77)