Decision Procedures and Verification

Martin Bлича

Charles University

30.4.2018
Program analysis
Introduction

- Detection of software defects
  - Traditionally → Testing (specific inputs)
- **Software verification**
  - Goal: to decide whether the specification is satisfied *for all possible inputs*
  - Specification: No division-by-0, \( x < y \) for program variables \( x \) and \( y \), . . .
- **Reachability problem**
  - Problem of checking whether a given program state occurs in any execution of the program
- Undecidable in general
  - Unbounded allocation of memory
  - Partial solutions exist
Introduction

- Partial solutions
  - Work for a subset of programs
  - Testing - declares program incorrect if an input is found that violates the specification.
  - Core of the solution = reasoning engine with decision procedure (SMT solver)

- dynamic program vs static decision procedure
  - simultaneous assignment of all variables satisfying given formula

- static single assignment (SSA) form

- Approximations:
  - Underapproximation - considers a subset of possible paths
  - Overapproximation - considers superset of possible paths
Terminology

- An *execution path* is a sequence of program instructions executed during a run of a program.
  - Can be partial.
- An *execution trace* is a sequence of states that are observed along an execution path.
  - Many traces along a single path are possible, corresponding to different inputs.
- *Symbolic simulation* - technique using symbolic representation of traces.
  - automatic test generation, detection of dead code, verification of properties
- *Assertion* is a program instruction that takes a condition as argument, and if the condition evaluates to false, it reports an error and aborts.
  - Verifying an assertion means proving that for all inputs the condition of the assertion evaluates to true.
Checking Feasibility of a Single Path

1. void ReadBlocks(int data[], int cookie)
2. {
3.    int i = 0;
4.    while (true)
5.    {
6.        int next;
7.        next = data[i];
8.        if (!(i < next && next < N)) return;
9.        i = i + 1;
10.       for (; i < next; i = i + 1)
11.          {
12.              if (data[i] == cookie)
13.                  i = i + 1;
14.              else
15.                  Process(data[i]);
16.          }
17.    }
18. }

- Artificial, but useful low-level example
- N denotes number of elements in the array
- Specification: no access out-of-bounds
- Consider single path (generalization later)
  1. Run through for loop once, take the else branch.
  2. Exit the while loop in the second iteration on Line 8.
Checking Feasibility of a Single Path

- Consider the sequence of instructions corresponding to this execution
- Record the branching conditions corresponding to the branches taken
- Rewrite instructions and conditions into the static single assignment representation
  - timestamped versions of variables
  - new version of a variable for each write (assignment)
- Translate SSA into logical formula \( \Rightarrow \) *path constraint*
  - Replace assignments with equalities and conjunct everything including branch conditions.
Checking Feasibility of a Single Path

<table>
<thead>
<tr>
<th>Line</th>
<th>Kind</th>
<th>Instruction or condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Assignment</td>
<td>i = 0;</td>
</tr>
<tr>
<td>7</td>
<td>Assignment</td>
<td>next = data[i];</td>
</tr>
<tr>
<td>8</td>
<td>Branch</td>
<td>i &lt; next &amp;&amp; next &lt; N</td>
</tr>
<tr>
<td>9</td>
<td>Assignment</td>
<td>i = i + 1;</td>
</tr>
<tr>
<td>10</td>
<td>Branch</td>
<td>i &lt; next;</td>
</tr>
<tr>
<td>11</td>
<td>Branch</td>
<td>data[i] != cookie;</td>
</tr>
<tr>
<td>14</td>
<td>Function call</td>
<td>Process(data[i]);</td>
</tr>
<tr>
<td>10</td>
<td>Assignment</td>
<td>i = i + 1</td>
</tr>
<tr>
<td>10</td>
<td>Branch</td>
<td>!(i &lt; next)</td>
</tr>
<tr>
<td>7</td>
<td>Assignment</td>
<td>next = data[i];</td>
</tr>
<tr>
<td>8</td>
<td>Branch</td>
<td>!(i &lt; next &amp;&amp; next &lt; N)</td>
</tr>
</tbody>
</table>

Sequence of statements along a path
## Checking Feasibility of a Single Path

<table>
<thead>
<tr>
<th>Line</th>
<th>Kind</th>
<th>Instruction or condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Assignment</td>
<td>( i_1 = 0; )</td>
</tr>
<tr>
<td>7</td>
<td>Assignment</td>
<td>( \text{next}_1 = \text{data}_0[\text{i}_1]; )</td>
</tr>
<tr>
<td>8</td>
<td>Branch</td>
<td>( i_1 &lt; \text{next}_1 &amp;&amp; \text{next}_1 &lt; \text{N}_0 )</td>
</tr>
<tr>
<td>9</td>
<td>Assignment</td>
<td>( i_2 = i_1 + 1; )</td>
</tr>
<tr>
<td>10</td>
<td>Branch</td>
<td>( i_2 &lt; \text{next}_1; )</td>
</tr>
<tr>
<td>11</td>
<td>Branch</td>
<td>( \text{data}_0[\text{i}_2] \neq \text{cookie}_0; )</td>
</tr>
<tr>
<td>14</td>
<td>Function call</td>
<td>( \text{Process(data}_0[\text{i}_2]); )</td>
</tr>
<tr>
<td>10</td>
<td>Assignment</td>
<td>( i_3 = i_2 + 1 )</td>
</tr>
<tr>
<td>10</td>
<td>Branch</td>
<td>( !(i_3 &lt; \text{next}_1) )</td>
</tr>
<tr>
<td>7</td>
<td>Assignment</td>
<td>( \text{next}_2 = \text{data}_0[\text{i}_3]; )</td>
</tr>
<tr>
<td>8</td>
<td>Branch</td>
<td>( !(i_3 &lt; \text{next}_2 &amp;&amp; \text{next}_2 &lt; \text{N}_0) )</td>
</tr>
</tbody>
</table>

**SSA form of the trace**
Checking Feasibility of a Single Path

\[
ssa \iff i_1 = 0 \\
next_1 = data_0[i_1] \\
(i_1 < next_1 \land next_1 < N_0) \\
i_2 = i_1 + 1 \\
i_2 < next_1 \\
data_0[i_2] \neq cookie_0 \\
i_3 = i_2 + 1 \\
!(i_3 < next_1) \\
next_2 = data_0[i_3] \\
!(i_3 < next_2 \land next_2 < N_0)
\]

- All evaluations of inputs \(data_0\) and \(cookie_0\) satisfying this formula correspond to a trace for the chosen path
Assertion checking

1. Consider path leading the an assertion.
2. Take the path constraint of that path.
3. Add *negation* of the assertion to the path constraint.

- Satisfying assignment correspond to trace leading to assertion with its condition violated.
- Problem of verifying corectness of a path in a program is reduced to checking the satisfiability of a formula.
Checking Feasibility of All Paths in a Bounded Program

- Number of paths can grow exponentially in the number of branches.
- Approach described previously would need to solve exponential number of decision problems.
- Better approach ⇒ generate SSA for \textit{bounded} program with branches as a whole.
- SSA is converted to a formula that encodes \textit{all} possible paths.
Checking Feasibility of All Paths in a Bounded Program

SSA for the whole program

1. Unfold loops pre-specified number of times.
2. Assign the condition of each if statement to a new variable.
   ▶ \( \gamma \) (for guard)
3. Identify points where control-flow reconverges.
4. Add \( \phi \)-instructions setting the correct values of variables.
   ▶ For variables that has been changed in either branch.
5. Translate to formula as before.
   ▶ If-then-else operator to represent \( \phi \) instructions
6. Satisfying assignment corresponds to one trace (of one path).
   ▶ Assignment of guard variables determines the branches taken.

▶ Example: for-loop from ReadBlocks unrolled 2 times
Checking Feasibility of All Paths in a Bounded Program

Example of SSA

```
if (i < next) {
  if (data[i] == cookie)
    i = i + 1;
  else
    Process(data[i]);
  i = i + 1;
}
if (i < next) {
  if (data[i] == cookie)
    i = i + 1;
  else
    Process(data[i]);
  i = i + 1;
}```
Under-approximation vs Over-approximation

► What we have seen:
  ► Transformation to loop-free program by unrolling loops
  ► Under-approximation technique
    ▶ Considers a *subset* of possible paths.
    ▶ If it detects a bug, it is real.
    ▶ Can declare program safe only up to given bound.

► What we will see:
  ► Transformation to loop-free program using non-determinism
  ► Over-approximation technique
    ▶ Considers a *superset* of possible paths.
    ▶ Detected bugs can be spurious
    ▶ If the over-approximation is safe, the original program is safe.
Over-approximating transformation

1. For each loop and each program variable that is modified by the loop, add an assignment at the beginning of the loop that assigns a nondeterministic value to the variable.

2. After each loop, add an assumption that the negation of the loop condition holds.
   ▶ An assumption is a program statement `assume(c)` that aborts any path that does not satisfy `c`.

3. Replace each while loop with an if statement using the condition of the loop as the condition of the if statement.
Over-approximating transformation

Example

Original program

```c
int i = 0;
int j = 0;

while (data[i] != '\n')
{
    i ++;
    j = i;
}
assert (i == j);
```

Transformed program

```c
int i = 0;
int j = 0;

if (data[i] != '\n')
{
    i = *;
    j = *;
    i ++;
    j = i;
}
assume (data[i] == '\n')
assert (i == j);
```
Checking over-approximating program

- Transformation to SSA/formula as before
  - Nondeterministic assignment modelled by incrementing variable counter.
  - Assumption translated by conjoining its condition to the formula.
- Formula is unsatisfiable.
- Program is safe for any number of iterations.
- Abstraction worked, because the assertion does not depend on previous iterations of the loop.
  - In other cases, the abstraction needs to be refined.
Loop invariant

- Key tool in any analysis of unbounded program.

**Definition**

A *loop invariant* is any predicate holds at the beginning of the body irrespective of how many times the loop iterates.

```plaintext
1  int i = 0;
2  while (i != 10) {
3      ...
4      i++;
5  }
⇒ 0 ≤ i < 10
```

- *Induction* is used to prove that a given formula is an invariant.
Proving loop invariant by induction

- Assume program in the following form where code fragments A,B are loop-free and condition C and invariant I are without side-effects.

- Prove that I is invariant by induction:
  1. Base case: Prove I is satisfied when entering the loop for the first time.
  2. Step case: Prove that from a state satisfying I, by executing the loop body once, we get to a state satisfying I.

Loop

1. A;
2. while (C){
3.   assert (I);
4.   B;
5. }

Base case

1. A;
2. assert (C => I);

Step case

1. assume (C & I);
2. B;
3. assert (C -> I);
### Proving loop invariant by induction

**Example**

<table>
<thead>
<tr>
<th>Loop</th>
<th>Base case</th>
<th>Step case</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int i = 0;</code></td>
<td><code>int i = 0;</code></td>
<td><code>assume(i != 10 &amp;&amp; i</code></td>
</tr>
<tr>
<td>`while(i != 10){</td>
<td><code>assert(i != 10 -&gt; (i</code></td>
<td><code>&gt;= 0 &amp;&amp; i &lt; 10));</code></td>
</tr>
<tr>
<td><code>++i;</code></td>
<td><code>&gt;= 0 &amp;&amp; i &lt; 10));</code></td>
<td><code>++i;</code></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td><code>assert(i != 10 -&gt; (i</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&gt;= 0 &amp;&amp; i &lt; 10));</code></td>
</tr>
</tbody>
</table>

By checking the base case program and step case program using techniques for loop-free programs, we verify that $0 \leq i < 10$ is an invariant of the loop.
Recall over-approximating transformation.

Assume that for each loop $l$ we have found a loop invariant $I_l$. For each loop add the following steps to the transformation.

4. Add an assertion that $I_l$ holds before the nondeterministic assignments to the loop variables.
   ▶ This establishes the base case.

5. Add an assumption that $I_l$ holds after the nondeterministic assignments to the loop variables.
   ▶ This is the induction hypothesis.

6. Add an assertion that $C \Rightarrow I_l$ holds at the end of the loop body.
   ▶ This proves the induction step.
Finding invariants

- The challenge is to find loop invariant that is strong enough to prove the property.
  - TRUE is always an invariant, but not very useful one.
- Finding loop invariants is an area of active research.
- Simple option: constructing candidates from predicates appearing in the code, or combining program variables with usual relational operators.
- Generalizing facts obtained from examining unrolling of the loop.
- ...