CS615: Formal Specification and Verification of Programs 2019

Lecture 19: Practical model checking

Instructor: Ashutosh Gupta

IITB, India

Compile date: 2019-11-05

Limited verification

Full verification is a very hard goal.

Soundiness: May be reduced objectives give us reasonable guarantees.

We will look at two popular methods that have been widely used.

1. Bounded model checking
2. Concolic testing

Topic 19.1

Bounded Model checking

Avoid complete fixed point computation

- For many programs symbolic model checking does not terminate
- Lets compromise in computing fixed point
- We can symbolically execute up to a fixed depth
- Very useful tool in falsification(bug finding)

Bounded model checking(BMC)

Algorithm 19.1: Bounded model checking

Input: $P = (V, L, \ell_0, \ell_e, E)$ and bound b

$reach : L \to \Sigma(V) \ := \ \lambda x.\bot;$

$worklist \ := \ \{(\ell_0, \top, 0)\};$

while $worklist \neq \emptyset$ **do**

  choose $(l, F, d) \in worklist;$

  $worklist \ := \ worklist \setminus \{(\ell, F, d)\};$

  if $d \leq b$ *and* $\neg(F \Rightarrow reach(\ell))$ *is sat* **then**

    $reach \ := \ reach[\ell \mapsto reach(\ell) \vee F];$

    foreach $(\ell, \rho(V, V'), \ell') \in E$ **do**

      $worklist \ := \ worklist \ \cup \ \{(\ell', sp(F, \rho), d + 1)\};$

if $reach(\ell_e) \neq \bot$ **then**

  return UNSAFE

else

  return SAFE *up to depth b*

Implementing BMC

A BMC tool is not implemented as discussed earlier

The program is turned into a giant satisfiability problem and solved using a satisfiability solver.

Bounding using loop unrolling

- Unroll the loops a fixed number of times, say n, and add appropriate if-conditions for early exists from the loop.
- Modify recursive function calls similarly

In some execution of the original programs, if a loop executes more than n times then the modified program will reach a dead end.

Example: bounded loop unrolling

Example 19.1

Original program

```
x=0;
while (x < 2) {
  y=y+x;
  x++;
  assert( y < 5);
}
```

Unrolled the loop three times

```
x=0;
if(x < 2) {
   y=y+x;
   x++;
   assert( y < 5);
   if(x < 2) {
      y=y+x;
      x++;
      assert( y < 5);
      if(x < 2) {
         y=y+x;
         x++;
         assert( y < 5);
      }
      if( !(x < 2) ) goto DEAD_END;
   }
}
```

SSA encoding and SMT formula

The loop free program is translated into single static assignment(SSA) form.

- After every assignment fresh names are given to the variables
- At join points instructions are added to feed in correct values

Example 19.2

Original program

```
foo(x,y) {
  x=x+y;
  if (x!=1)
    x=2;
  else
    x++;
  assert(x<=3);
}
```

Program after SSA transformation

```
foo(x0,y0) {
  x1 = x0 + y0;
  if( x1 != 1 )
    path_b = 1
    x2 = 2;
  else
    path_b = 0
    x3 = x1 + 1;
  x4 = path_b ? x2 : x3;
  assert( x4 <= 3 );
}
```

SSA to SMT formula

An SSA program can be easily translated into a formula.

Example 19.3

Original program

```
foo(x0,y0) {
  x1 = x0 + y0;
  if( x1 != 1 )
    path_b = 1
    x2 = 2;
  else
    path_b = 0
    x3 = x1 + 1;
  x4=path_b?x2:x3;
  assert(x4 <= 3);
}
```

QF_LIA formula for the SSA program

```
(assert (= x1 (bvadd x0 y0) ) )
(assert (= x2 #x00000002) )
(assert (= x3 (bvadd x1 #x00000001))
(assert (= path_b (distinct x1 1))
(assert (ite path_b (= x4 x2)
                    (= x4 x3)) )
(assert (not (bvsle x4 3) ) )
```

If the above is sat, the program has a bug

SMT Input

The SMT input with all the needed declarations.

```
(set-logic QF_BV)
(declare-fun x0 () (_ BitVec 32))
(declare-fun x1 () (_ BitVec 32))
(declare-fun x2 () (_ BitVec 32))
(declare-fun x3 () (_ BitVec 32))
(declare-fun x4 () (_ BitVec 32))
(declare-fun y0 () (_ BitVec 32))
(assert (= x1 (bvadd x0 y0) ) )
(assert (= x2 #x00000002) )
(assert (= x3 (bvadd x1 #x00000001) ) )
(assert (= path_b (distinct x1 #x00000001) ) )
(assert (ite path_b (= x4 x2) (= x4 x3)) )
(assert (not (bvsle x4 #x00000003) ) )
(check-sat)
(get-model)
```

Let us feed the problem in Z3

CBMC

- Takes C/C++ programs as input and a loop unrolling bound k
- Returns an error execution or proves safety upto k unrolling of loops
- Robust tool, can take any input

Let us play with CBMC!

An effective technology

- There are very successful BMC tools, *e. g.*, CBMC
- Not a full verification method, but somewhat better than testing
- Implementations may unroll the program upto depth b and then generate path constraints for all the unrolled paths and solve the constraints

Topic 19.2

Concolic Testing

Concolic (Concrete+Symbolic) testing

Testing algorithm:
find a test suite that covers most of branches of a program

Concolic testing is one of the testing algorithm, which is aided by formal methods

- Execute the program both symbolically and concretely
  - Use symbolic constraints to guide the search
  - Use concrete values to simplify the constraints

Original paper: `http://dl.acm.org/citation.cfm?id=1065036`

Modeling input vector

In our formalism, havocs model inputs.

For ease of notation in the next algorithm, we assume that

- all the labels are guarded commands with a single assignment(havocs are allowed) and a conjunctive guard.
- all branches are mutually disjoint
- In the program, there are locations with no outgoing edges.

Concolic Testing

Algorithm 19.2: Concolic testing

Input: $P = (V, L, \ell_0, \ell_e, E)$

$\ell := \ell_0$ $v := randomVector()$;

$stack := \ < \{(\ell_0, \_, \_) \in E\} >$; $\pi := \ <>$;

while $stack.size() \neq 0$ **do**

  $Tr := stack$.peek();

  if $Tr \neq \emptyset$ **then**

    choose $(l, [F, x := exp], \ell') \in Tr$ such that $v \models F$;

    π.push($F \wedge x' = exp \wedge frame(x)$);

    $stack$.replaceTop($Tr \setminus \{(\ell, [F, x := exp], \ell')\}$);

    $stack$.push($\{(\ell', \_, \_) \in E\}$);

    $\ell := \ell'$; $v := v[x \mapsto exp(v)]$ // including the havoc ;

  else

    find min. $j \geq 0$ s.t. $\forall i > j.\ stack[i] = \emptyset$;

    $stack$.resize(j); π.resize($j - 1$);

    $v' :=$ solve($\pi, stack[j]$);

Exercise 19.1

a.Describe solve formally

b.add data structures to return test-suite

Coverity

End of Lecture 19