CS510 Assignement #4 Solution

May 1, 2015

1 Symbolic Model Checking (30p)

For this problem, you will need to convert a program into a set of constraints. The constraints should be such that every satisfying assignment corresponds to an execution of the program that violates an assertion. The program is given below:

```
void main (void)
{
  int    i = input();
  char * buf = input2();
  char   next = 0;

  if (buf [i] == '\0')
    {
      int start = 0;
      while (start < i)
        {
          buf [start] = '.';
          start++;
          next = buf [start];
        }
    }
}
```

(a) Show the result of unrolling the loop twice (2) and adding an unwinding assertion.

(b) Convert the loop-free program (with assertion) from part (a) into Single Static Assignment (SSA) form.

(c) Generate a constraint system C from the SSA program in part (a) such that C is satisfiable if and only if the unwinding assertion can be violated. Note that you don't need to turn it into boolean formula.

(d) Is the constraint system C from part (c) satisfiable? If yes, show a satisfying assignment.

Answer:

(a)
```
void main (void)
{
  int    i = input();
  char * buf = input2();
  char   next = 0;

  if (buf [i] == '\0')
    {
      int start = 0;
      if (start < i)
        {
          buf [start] = '.';
          start++;
          next = buf [next];
          if (start < i)
           {
             buf [start] = '.';
             start++;
             next = buf [start];
             assert (!(start<i));
           }
        }
    }
}
```

(b)
```
void main (void)
{
  ...
  next0 = 0;
  if (buf0 [i] == '\0')
    {
      int start0 = 0;
      if (start0 < i)
        {
          buf1 = lamda x: x==start0? '.', buf0[x];
          start1=start0 + 1;
          next1 = buf1 [start1];
          if (start1 < i)
           {
             buf2 = lamda x: x==start1? '.', buf1[x];
             start2=start1 + 1;
             next2 = buf2 [start2];
             assert (!(start2<i));
           }
           start3= (start1<i)? start2: start1;
```

```
            next3= (start1<i)? next2: next1;
            buf3 = (start1 <i)? buf2: buf1;
        }
        start4= (start0<i)? start3: start0;
        next4= (start0<i)? next3: next0;
        buf4 = (start0 <i)? buf3: buf0;
    }
    next5= (buf0[i]=='\0')? next4: next0;
    buf5 = (buf0[i]=='\0')? buf4: buf0;
}
```

Note that the assignments at the join points are not really needed as they are not used. I put it here for the completeness purpose. The lambda expression makes the program not legal for execution, but good for the sat solver.

(c)

$$
\begin{aligned}
next0 &= 0 \wedge \\
start0 &= 0 \wedge \\
buf1 &= \lambda x : x == start0 ?\ '.', buf0[x] \wedge \\
start1 &= start0 + 1 \wedge \\
next1 &= buf1[start1] \wedge \\
buf2 &= \lambda x : x == start1 ?\ '.', buf1[x] \wedge \\
start2 &= start1 + 1 \wedge \\
next2 &= buf2[start2] \wedge \\
&\neg(\neg(start2 < i))
\end{aligned}
$$

Note that the $buf1 = \lambda x : x == start0 ?\ '.', buf0[x]$ is equivalent to $\ldots((x == start0 - 1) \wedge (buf1[x] == buf0[x])) \vee ((x == start0) \wedge (buf1[x] ==' .')) \vee \ldots$.

(d) Yes. The following assignments satisfies the constraints.

```
i=10
buf0[...]=0
start0=0
buf1[0]='.'
start1=1
next1='A'
buf2[1]='.'
start2=2
next2='A'
```

Several values, including the ones for next1 and next2 are arbitrarily chosen by the solver.

2 Predicate Abstraction (30p)

In order to mitigate state explosion in explicit state model checking, predicate abstraction is often used to reduce state space. Counter-example guided refinement may be needed during the process.

```
void main (void)
{
   int a, b;
   a = 2;
   b = 5;
   if (a > b) {
      a--;
   } else {
      a+=4;
   }

   assert(a>b);
}
```

(a) Starting with predicate `a>b`, apply predicate abstraction to the above program.

(b) Perform explicit state model checking on the abstract program, present your execution tree and the counter example, if there is one.

(c) If there is a counter example in (b), test if it is a counter example in the original program.

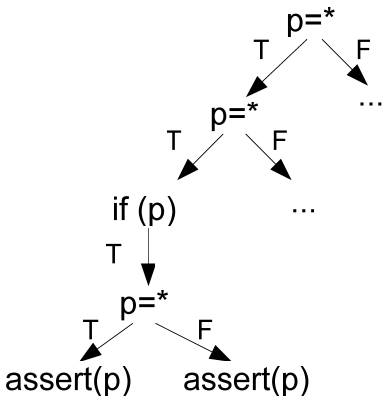
(d) If the counter example is bogus, refine your abstraction so that either you find a real counter example or show the correctness of the program.

Answer:

(a) Assume `p` represents `a>b`

```
void main (void)
{
   bool p;
   p=*;
   p=*;
   if (p) {
      p=*;
   } else {
      p=*;
   }

   assert(p);
}
```



(b) The counter example is shown in the tree-like figure.

(c) However, note that the true branch in the original program cannot be taken. So this is a bogus counter example.

(d) As the contradiction occurs at the first three constraints, we refine our model with the first two constraints. Now we have four predicates: `p1` is for `a==2`, `p2` is for `b==5`, `p3` is for `a>b`.

```
void main (void)
{
   bool p1, p2, p3;
   p1_0= T;       // a=2
   p2_0= *;       // a=2
   p3_0= *;       // a=2

   p2_1= T;       // b=5
   p3_1=p1 & p2 ? F, *; // b = 5
   if (p3_1) {
      p1_1=p1_0 ? F, *;
      p3_2=p1_0 & p2_1 ? F, *;
   } else {
      p1_1=p1_0 ? F, *;
      p3_3=p1_0 & p2_1 ? T, *;
   }
   p3_4=p3_1 ? p3_2, p3_3;
   assert(p3_4);
}
```

The program always model-checks. Note that the refinement is not unique.

3 Concolic Execution (25p)

```
1  input (x,y,z);
2  input (A[]);
3  a=2*x;
4  b=y*y;
5  c=x+1;
6  if (a>b) {
7    c=c+1;
8    if (b>0)
9       c=c+1;
10   else
11      c=c+2;
12 }
13 if (A[z]>c) {
14   print ("case 1");
15 } else {
16   print ("case 2");
}
```

Apply concolic execution to the above program. The initial input is x=3, y=4, z=1, A[]={3,2,1}. Show the path constraints generated at each iteration of the algorithm and the generated input. What is the final statement coverage? Assume the solver can not solve non-linear or array index related constraints.

Answer:

After the first execution, the ideal path constraints are:

$$\neg 2 * x > y * y$$

$$\neg A[z] > x + 1$$

Since the nonlinear factors are replaced with concrete values, we get

$$\neg 2 * x > 16$$

$$\neg A[1] > x + 1$$

In the first round, the algorithm negates the predicate at 13. Solving the constraints gets $A[1] = 5$ (or any valuations as long as $A[1] > x + 1$ is satisfied).

By executing the program with the new input, it is confirmed that the program takes the desired path (the true branch of 13). Note that due to concretization, it may not be the case that the generated input does not lead to the desired path. Hence, a confirmation step is needed.

In the second round, the ideal path constraints are:

$$\neg 2 * x > y * y$$

$$A[z] > x + 1$$

The partially concretized constraints are:

$$\neg 2 * x > 16$$

$$A[1] > x + 1$$

This time, the algorithm tries to negate the predicate at 6. Solving the constraints gets $x = 9$ (or any valuations satisfying $2 * x > 16$). It is confirmed that with the new input, the program takes the desired path (the true branch of 6). So in the third round, the ideal path constraints are:

$$2 * x > y * y$$

$$y * y > 0$$

$$\neg A[z] > x + 3$$

After concretization, the constraints become

$$2 * x > y * y$$

$$16 > 0$$

$$\neg A[1] > x + 3$$

Negating the predicate at 13 gets $A[1] = 13$. So in the fourth round, the ideal path constraints become:

$$2 * x > y * y$$

$$y * y > 0$$

$$A[z] > x + 3$$

After concretization, the constraints become

$$2 * x > y * y$$

$$16 > 0$$

$$A[1] > x + 3$$

Since the condition of the predicate at 8 is concretized, it can not be switched. Since all negations have been tried, the algorithm terminates without covering statement 11.

The above solution has the goal of exploring all paths in the execution tree. In another possible solution, with the goal of covering all statements instead of all paths, the above fourth round is not tried.

4 Concurrency (15p)

Thread T1

```
0  r=input();
1  B=1000;
2  spawn(T2);
3  ...
4  acquire (L);
5  t=input ();
6  B=B+t;
7  release (L)
8  ...
9  join (T2);
```

Thread T2

```
10  acquire (L);
11  y=input ();
12  B=B-y;
13  release (L)
      ...
14  if (r>0) {
15     acquire (L);
16     B=B+y*r;
17     release (L);
18  }
```

One way to decide if a concurrent program has a bug is to run the program on the same input many times and observe if the same output is always produced. If not, the program is faulty as it does not produce stable output. Is the above program faulty? If so, present the input and the two schedules that produce different outputs.

For input r=0, t=10, y=20, how many races (pairs of accesses) are reported following the lockset algorithm? How many following the happens-before algorithm.

Answer:

The program is not faulty. If line 16 is changed to "B=B+(B+y)*r", the program becomes faulty due to an atomicity violation.

Two races are reported by lockset. They are statements 0 and 14, 1 and 12. No race is reported if the happens-before algorithm is used.