

CS766: Analysis of concurrent programs (first half) 2021

Lecture 19: Practical model checking

Instructor: Ashutosh Gupta

IITB, India

Compile date: 2021-03-11

Limited verification

Full verification is a **very hard goal**.

Soundness: 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

$reach : L \rightarrow \Sigma(V) := \lambda x. \perp$; $worklist := \{(\ell_0, \top, 0)\}$;

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

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

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 $reach(\ell_e) \neq \perp$ **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

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;
    }
}
```

Example 19.1

Original program

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


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

Program after SSA transformation

Example 19.2

Original program

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

```
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

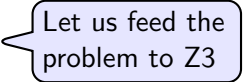
```
(assert (= x1 (bvadd x0 y0) ) )  
(assert (= x2 #x00000002) )  
(assert (= x3 (bvadd x1 #x00000001)) )  
(assert (= path_b (distinct x1 1))  
(assert (ite path_b (= x4 x2) (= x4 x3)) )  
(assert (not (bvsle x4 3) ) )
```

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))
(declare-fun path_b () (Bool))
(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)
```



Let us feed the
problem to 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

Concurrent BMC

BMC for concurrent programs

- ▶ Full verification of concurrent programs is hard.
- ▶ Most tools use some form of Bounded verification
- ▶ Let us see how to do BMC for C program

Set of Events

An execution of program generates a set of read/write events E .

We define a relation po over E as follows.

Definition 19.1

For $e_1, e_2 \in E$, $(e_1, e_2) \in \text{po}$ if e_1 was generated before e_2 by the same thread.

Memory operation relation

The read write operations create the following relations $\subseteq E \times E$.

- ▶ rf : every read reads from exactly one write
- ▶ ws : all writes on a global are totally ordered
- ▶ fr : no other write comes between the write-read pairs in rf

Execution relations and condition

Memory model has conditions on the relations, which are

- ▶ po program order
- ▶ rf read from
- ▶ ws write serialization
- ▶ fr from read

Theorem 19.1

In a valid execution, $po \cup rf \cup ws \cup fr$ is acyclic

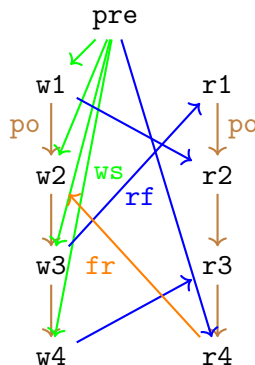
Example: execution

```
pre: m1 := s1 := m2 := s2 := 0

thread T1:                thread T2:
w1: m1 := v                r1: a1 := s1
w2: m2 := v                r2: c1 := m1
w3: s1 := 1                r3: a2 := s2
w4: s2 := 1                r4: c2 := m2

post: (a1=1 && a2=1) ⇒ c1+c2=2*v
```

Invalid execution:



Constraints generation

If we want to do bounded model checking, we check satisfiability of the following formula.

The formula $F := F_1 \wedge F_2 \wedge F_3 \wedge F_4$ that encodes violating executions has four parts

1. $F_1 = \text{SSA formula}$
2. $F_2 = \text{well-formed rf}$
3. $F_3 = \text{write serialization}$
4. $F_4 = \text{fr constraints}$

We will present some of the above constraints.

SSA formula(F_1)

Reads/writes on the globals, and writes to locals get fresh names.

Let us consider our example again.

```
pre: m1 := s1 := m2 := s2 := 0

thread T1:                thread T2:
w1: m1 := v                r1: a1 := s1
w2: m2 := v                r2: c1 := m1
w3: s1 := 1                r3: a2 := s2
w4: s2 := 1                r4: c2 := m2

post: (a1=1 && a2=1)  $\Rightarrow$  c1+c2=2*v
```

The SSA encoding of the above is

$$W.pre.m1 = 0 \wedge W.pre.s1 = 0 \wedge W.pre.m2 = 0 \wedge W.pre.s2 = 0 \quad (\text{pre})$$

$$W.w1.m1 = v \wedge W.w2.m2 = v \wedge W.w3.s1 = 1 \wedge W.w4.s2 = 1 \quad (\text{T1})$$

$$a1 = R.r1.s1 \wedge c1 = R.r2.m1 \wedge a2 = R.r3.s2 \wedge c2 = R.r4.m2 \quad (\text{T2})$$

$$\neg((a1 = 1 \wedge a2 = 1) \Rightarrow c1 + c2 = 2v) \quad (\text{post})$$

Well-formed $rf(F_2)$

Every read reads from exactly one write and the write happens before the read.

We need to introduce a few variables.

We use clock variables for the timing of the events.

- ▶ Integer $t_{w.w3.s1}$ encodes the time when the write at $w3$ occurred.

We also create a Boolean variable for each potential write-read pair.

- ▶ Boolean $b_{pre.r1.s1}$ indicates that the read at $r1$ of $s1$ is reading from the write at pre .

Well-formed $rf(F_2)$ (contd.)

pre: $m1 := s1 := m2 := s2 := 0$

thread T1:		thread T2:
w1: $m1 := v$		r1: $a1 := s1$
w2: $m2 := v$		r2: $c1 := m1$
w3: $s1 := 1$		r3: $a2 := s2$
w4: $s2 := 1$		r4: $c2 := m2$

post: $(a1=1 \ \&\& \ a2=1) \Rightarrow c1+c2=2*v$

Consider the read of $s1$ at $r1$. It may read from two writes, which are the initialization and the write at $w3$.

This is encoded as follows.

$$(b_{\text{pre}.r1.s1} \vee b_{w3.r1.s1}) \wedge (b_{\text{pre}.r1.s1} \Rightarrow W.\text{pre}.s1 = R.r1.s1) \wedge (b_{w3.r1.s1} \Rightarrow W.w3.s1 = R.r1.s1)$$

Constrains for encoding the happens-before condition

$$(b_{w3.r1.s1} \Rightarrow t_{W.w3.s1} < t_{R.r1.s1})$$

Similar constraints are generated for each read.

Write serialization and *fr* condition (F_3)

```
pre: m1 := s1 := m2 := s2 := 0

thread T1:                thread T2:
w1: m1 := v                r1: a1 := s1
w2: m2 := v                r2: c1 := m1
w3: s1 := 1                r3: a2 := s2
w4: s2 := 1                r4: c2 := m2

post: (a1=1 && a2=1)  $\Rightarrow$  c1+c2=2*v
```

Here are the *fr* constraints.

$$(b_{\text{pre.r1.s1}} \Rightarrow t_{\text{R.r1.s1}} < t_{\text{W.w3.s1}}) \wedge (b_{\text{pre.r2.m1}} \Rightarrow t_{\text{R.r2.m1}} < t_{\text{W.w1.m1}}) \wedge \\ (b_{\text{pre.r3.s2}} \Rightarrow t_{\text{R.r3.s2}} < t_{\text{W.w4.s2}}) \wedge (b_{\text{pre.r4.m2}} \Rightarrow t_{\text{R.r4.m2}} < t_{\text{W.w2.m2}})$$

po condition (F_4)

We need to encode the intra-thread order of events that is preserved by the model. Recall, writes on different globals are relaxed.

```
pre: m1 := s1 := m2 := s2 := 0
```

```
thread T1:                thread T2:
w1: m1 := v                r1: a1 := s1
w2: m2 := v                r2: c1 := m1
w3: s1 := 1                r3: a2 := s2
w4: s2 := 1                r4: c2 := m2
```

```
post: (a1=1 && a2=1)  $\Rightarrow$  c1+c2=2*v
```

po for T1: ($t_{W.pre_} < t_{W.w1.m1} < t_{W.w2.m2} < t_{W.w3.s1} < t_{W.w4.s2}$)

po for T2: ($t_{W.pre_} < t_{R.r1.s1} < t_{R.r2.m1} < t_{R.r3.s2} < t_{R.r4.m2}$)

End of Lecture 19