CS766: Analysis of concurrent programs 2022

Lecture 19: Bounded model-checking for concurrent programs

Instructor: Ashutosh Gupta

IITB, India

Compile date: 2022-02-09

Limited verification

Full verification is a very hard goal.

Soundiness: May be reduced objectives give us reasonable guarantees.

We will look at a popular method that have been widely used.

Bounded model checking(BMC)

Topic 19.1

Basics of BMC

BMC

We get a program and a property as input.

We verify that the program does not violate the property in a given number of steps.

Only only consider safety. No liveness properties such as starvation or deadlock. (why?)

Implementing BMC

The program goes via several transformation steps.

1. Loop unrolling

2. SSA renaming

3. Translation to a giant formula

4. Use a sat solver to check the property.

Step 1: 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 program, 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);
}
```

Let us unroll the loop three times.

The program transformation does not pay attention to the logic of the program. It simply unrolls even if there are fewer iterations.

```
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;
    }
  }
}
```

Step 2 : 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 );
```

Step 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_BV 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

Step 4: 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

An effective technology

- There are very successful BMC tools, *e. g.*, CBMC
- Not a full verification method, but somewhat better than testing

Topic 19.2

Concurrent BMC

BMC for concurrent programs

- Full verification of concurrent programs is even more hard.
- Most tools use some form of Bounded verification
- Let us see how to do BMC for a concurrent 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 $e_1, e_2 \in E$, $(e_1, e_2) \in \mathrm{po}$ 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`

Recall: Execution relations and condition

We have the following relations over events.

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

Theorem 19.1

In a valid execution, $\texttt{po} \cup \texttt{rf} \cup \texttt{ws} \cup \texttt{fr}$ *is acyclic*

We need to encode the acyclic requirement using integer. Every event will get a distinct integer time point.

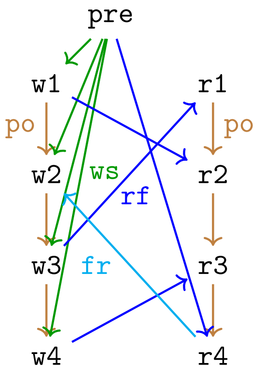
Example: execution

```
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)$\Rightarrow$c1+c2=2*v

Invalid execution:



Constraints generation

If we want to do bounded model checking, we not only need to encode the behavior of a thread but also the concurrent interaction.

The formula F that encodes violating executions has following parts

1. $F_{\mathtt{po}}$ = program ordering constraints
2. F_{ssa} = SSA formula
3. $F_{\mathtt{rf}}$ = well-formed `rf`
4. $F_{\mathtt{fr}}$ = `fr` constraints

Let us discuss the constraints in detail.

po condition (F_{po})

We need to encode the intra-thread order of events.

We use integer variables to encode the timing of the events. Integer variable

$$t_{\texttt{w3}}$$

encodes the time when the write at `w3` occurred.

$$\begin{tikzcd}
\texttt{w1.Wx} = 0 \arrow[d, "\texttt{po}"] \\
\texttt{r1.Ry} = 0
\end{tikzcd}$$

$$t_{\texttt{w1}} < t_{\texttt{r1}}$$

Example: F_{po}

Example 19.4

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)⇒c1+c2=2*v
```

F_{po} consists of the following formulas.

po for T1:

$$t_{\texttt{pre.m1}} < t_{\texttt{w1}} \land t_{\texttt{pre.s1}} < t_{\texttt{w1}} \land t_{\texttt{pre.m2}} < t_{\texttt{w1}} \land t_{\texttt{pre.s2}} < t_{\texttt{w1}} \land$$

$$t_{\texttt{w1}} < t_{\texttt{w2}} < t_{\texttt{w3}} < t_{\texttt{w4}}$$

po for T2:

$$t_{\texttt{pre.m1}} < t_{\texttt{r1}} \land t_{\texttt{pre.s1}} < t_{\texttt{r1}} \land t_{\texttt{pre.m2}} < t_{\texttt{r1}} \land t_{\texttt{pre.s2}} < t_{\texttt{r1}} \land$$

$$t_{\texttt{r1}} < t_{\texttt{r2}} < t_{\texttt{r3}} < t_{\texttt{r4}}$$

SSA formula(F_{ssa})

We translate each loop free thread into single static assignment(SSA) form.

- fresh names to the local variables after every assignment
- add instructions at join points to feed in correct values
- give a fresh name at each read and write of global variables

Example : F_{ssa}

Example 19.5

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)⇒c1+c2=2*v
```

Locals are not given fresh names because they are not modified.

F_{ssa} consists of the following formulas.

The SSA encoding of pre condition.

$$\mathtt{W.pre.m1} = 0 \wedge \mathtt{W.pre.s1} = 0 \wedge \mathtt{W.pre.m2} = 0 \wedge \mathtt{W.pre.s2} = 0$$

SSA encoding of thread T1.

$$\mathtt{W.w1.m1} = \mathtt{v} \wedge \mathtt{W.w2.m2} = \mathtt{v} \wedge \mathtt{W.w3.s1} = 1 \wedge \mathtt{W.w4.s2} = 1$$

SSA encoding of thread T2.

$$\mathtt{a1} = \mathtt{R.r1.s1} \wedge \mathtt{c1} = \mathtt{R.r2.m1} \wedge \mathtt{a2} = \mathtt{R.r3.s2} \wedge \mathtt{c2} = \mathtt{R.r4.m2}$$

SSA encoding of post condition.

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

$F_{\texttt{rf}}$: Well-formed $\texttt{rf}$

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

We need to introduce a Boolean variable for each potential write-read pair.Boolean

$$b_{\texttt{w}.\texttt{r}.\texttt{s1}}$$

indicates that the read at location $\texttt{r}$ of variable $\texttt{x}$ is reading from the write at $\texttt{w}$.

$$\begin{tikzcd} \texttt{w} : \texttt{x} := .. \arrow[d, "\texttt{rf}"'] \\ \texttt{r} : .. := \texttt{x} \end{tikzcd}$$

$$b_{\texttt{w}.\texttt{r}.\texttt{x}} \Rightarrow (\texttt{w}.\texttt{x} = \texttt{r}.\texttt{x} \land t_{\texttt{w}} < t_{\texttt{r}})$$

Example: $F_{\texttt{rf}}$

Example 19.6

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)⇒c1+c2=2*v
```

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

This is encoded as follows.

- Read from exactly one.
  $(b_{\texttt{pre.r1.s1}} \vee b_{\texttt{w3.r1.s1}})$
- If reads from `pre`
  $b_{\texttt{pre.r1.s1}} \Rightarrow (\texttt{W.pre.s1} = \texttt{R.r1.s1} \wedge t_{\texttt{pre.s1}} < t_{\texttt{r1}})$
- If reads from `w3`
  $b_{\texttt{w3.r1.s1}} \Rightarrow (\texttt{W.w3.s1} = \texttt{R.r1.s1} \wedge t_{\texttt{w3}} < t_{\texttt{r1}})$

Similar constraints are generated for each read.

Exercise 19.1

Is exactly one write encoding correct?

$F_{\mathtt{fr}}$: relation between ws and *fr*

If

- read r is reading from write w1 and
- write w2 is serialized after w1

then w2 is after read r.

$$\begin{tikzcd}
\mathtt{w1 : x := ..} \arrow[dr, "\mathtt{rf}"] \arrow[dd, "\mathtt{ws}"'] & \\
& \mathtt{r : .. := x} \arrow[dl, "\mathtt{fr}"] \\
\mathtt{w2 : x := ..} &
\end{tikzcd}$$

$$b_{\mathtt{w1.r.x}} \wedge t_{\mathtt{w1}} < t_{\mathtt{w2}} \Rightarrow t_{\mathtt{r}} < t_{\mathtt{w2}}$$

Example:

Here are the *fr* constraints.

Example 19.7

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)⇒c1+c2=2*v
```

- When r1 reads from pre
  $(b_{\texttt{pre.r1.s1}} \land t_{\texttt{pre.s1}} < t_{\texttt{w3}} \Rightarrow t_{\texttt{r1}} < t_{\texttt{w3}})$
- When r1 reads from w3
  $(b_{\texttt{w3.r1.s1}} \land t_{\texttt{w3}} < t_{\texttt{pre.s1}} \Rightarrow t_{\texttt{r1}} < t_{\texttt{pre.s1}})$

End of Lecture 19