IFM 2010 - TLAPS tutorial

Peterson algorithm

In this tutorial, we shall prove that Peterson's algorithm satisfies the mutual exclusion property, using the TLA Proof System and the Toolbox IDE.

1 Toolbox and TLAPS install

If you have not done it yet, you can download the Toolbox from

`http://www.tlaplus.net/tools/tla-toolbox/`

and dowload the TLA Proof System from

`http://www.msr-inria.inria.fr/~doligez/tlaps/`

Binaries are available for Windows, MacOS and Linux distributions. Notice that Cygwin (version 1.7 or higher) is required to install TLAPS on Windows.

2 Peterson Algorithm

This algorithm, formulated by Gary L. Peterson in 1981, is a concurrent programming algorithm for mutual exclusion that allows two processes to share a single-use resource without conflict, using only shared memory for communication.

2.1 Description

The algorithm allows two processes to share some resource with the three following properties :

– Mutual exclusion : the two processes cannot access the resource at the same time
– Progress : a process cannot immediately re-access the resource if the other process wants also to access the resource.
– Bounded waiting : there exists a limit on the number of times that the other process is allowed to access the resource after a process has made a request to access the resource.

We shall focus on the mutual exclusion property in the TLA+ specification we are to build.

We call *critical section* the moment when some process is accessing the shared resource. The Mutual Exclusion property ensures that the two processes are not both in the critical section at the same time.

The idea of the algorithm is the following : we use two variables, a function called `flag`, which says for each of the processes, if it wants to access the resource or not. And a simple variable `turn` which holds the process whose turn it is to enter the critical section.
At the beginning of the algorithm, `flag[0]` and `flag[1]` are equal to `FALSE` (none of the processes wants to access the resource), and `turn` is equal to `0` (process 0 has priority to access the resource).
Then, for making process `0` (for example) access the shared resource, we do the following :

(a1) turn the value of `flag[0]` to `TRUE`
(process `0` does want to enter the critical section)

(a2) turn the value of `turn` to `1`
(give the priority to the other process)

(a3) wait for `flag[1]` to be equal to `FALSE` and `turn` to be equal to `1`
(process `1` does not want to access the resource and process `0` has priority to access it)

(cs) enter the critical section
(access the resource)

(a4) turn the value of `flag[0]` to `FALSE`
(process `0` does not want to access the resource anymore)

2.2 PlusCal version

Peterson's algorithm can be expressed in the PlusCal language as follows :

```
Not(i) == IF i = 0 THEN 1 ELSE 0

--algorithm Peterson {
   variables flag = [i \in {0, 1} |-> FALSE], turn = 0;
   process (proc \in {0,1}) {
     a0: while (TRUE) {
     a1:   flag[self] := TRUE;
     a2:   turn := Not(self);
     a3a:  if (flag[Not(self)]) {goto a3b} else {goto cs} ;
     a3b:  if (turn = Not(self)) {goto a3a} else {goto cs} ;
     cs:   skip;  \* critical section
     a4:   flag[self] := FALSE;
     } \* end while
    } \* end process
  }
```

Notice that, for simplicity, the TLA+ specification we describe in the next section differs a little bit from the one that can be directly translated from the algorithm above, by the PlusCal translator.

3 The specification

Peterson's algorithm for two processes can already be verified by the Toolbox model-checker. But we use it as an exercise to make you learn ho to use TLA Proof Sytem.

Peterson's TLA+ specification can be downloaded at

`http://www.msr-inria.inria.fr/~doligez/tlaps/IFM2010/Peterson.tla`

Let us describe what you can find in this file.

3.1 Variables

We declare three variables, `flag`, `turn` and `pc` (which gives the current step of the algorithm). We define `vars` as the tuple composed by those three variables, and `ProcSet` as the set $\{0, 1\}$.

```
VARIABLES flag, turn, pc
vars == << flag, turn, pc >>
ProcSet == ({0,1})
```

3.2 The `Init` state

We then express the initial state of the algorithm as follows :

```
Init == /\ flag = [i \in {0, 1} |-> FALSE]
        /\ turn = 0
        /\ pc = [self \in ProcSet |-> CASE self \in {0,1} -> "a0"]
```

`flag[0]` and `flag[1]` are equal to `FALSE`.
`turn` is equal to `0`.
Both processes are entering the while spin.

3.3 The `Next` action

In order to define the `Next` action of Peterson's algorithm, we have to describe, for each process, every possible step of the algorithm. For example the `a0` step describes the fact that the considered process (`self`) is going from line `a0` to line `a1` in the algorithm (in that case, values of variables `flag` and `turn` don't change).

```
a0(self) == /\ pc[self] = "a0"
            /\ pc' = [pc EXCEPT ![self] = "a1"]
            /\ UNCHANGED << flag, turn >>
```

The second line above expresses the fact that the next value of the function `pc` is the same as the current one, except that its value on `self` is now `"a1"`.

We then do the same for each possible step of the algorithm :

```
a1(self) == /\ pc[self] = "a1"
            /\ flag' = [flag EXCEPT ![self] = TRUE]
            /\ pc' = [pc EXCEPT ![self] = "a2"]
            /\ UNCHANGED turn

a2(self) == /\ pc[self] = "a2"
            /\ turn' = Not(self)
            /\ pc' = [pc EXCEPT ![self] = "a3a"]
            /\ UNCHANGED flag

a3a(self) == /\ pc[self] = "a3a"
             /\ IF flag[Not(self)]
                   THEN /\ pc' = [pc EXCEPT ![self] = "a3b"]
                   ELSE /\ pc' = [pc EXCEPT ![self] = "cs"]
             /\ UNCHANGED << flag, turn >>
```

```
a3b(self) == /\ pc[self] = "a3b"
             /\ IF turn = Not(self)
                   THEN /\ pc' = [pc EXCEPT ![self] = "a3a"]
                   ELSE /\ pc' = [pc EXCEPT ![self] = "cs"]
             /\ UNCHANGED << flag, turn >>

cs(self) == /\ pc[self] = "cs"
            /\ TRUE
            /\ pc' = [pc EXCEPT ![self] = "a4"]
            /\ UNCHANGED << flag, turn >>

a4(self) == /\ pc[self] = "a4"
            /\ flag' = [flag EXCEPT ![self] = FALSE]
            /\ pc' = [pc EXCEPT ![self] = "a0"]
            /\ UNCHANGED turn
```

Now we can define `proc(self)` as the fact that one of the previous actions is being accomplished :

```
proc(self) == \/ a0(self) \/ a1(self) \/ a2(self) \/ a3a(self)
              \/ a3b(self) \/ cs(self) \/ a4(self)
```

Finally, we define the `Next` action, as the fact that either `proc` is accomplished for one of the processes, or the algorithm has finished (to prevent deadlock on termination).

```
Next == \E self \in {0,1}: proc(self)
```

3.4 Specification

The specification of the algorithm is given by the facts that the initial state is satisfied and that at every step either the action `Next` is satisfied or the variables in `vars` keep their values.

```
Spec == Init /\ [][Next]_vars
```

3.5 Mutual Exclusion

The property we want the algorithm to satisfy can be defined as the fact that `pc[0]` and `pc[1]` have not both value `"cs"` at the same time.

```
MutualExclusion == ~ (pc[0] = "cs" /\ pc[1] = "cs")
```

3.6 The invariant

Let us first define the property that ensures well-typedness of the variables .

```
TypeOK == /\ pc \in [{0,1} -> {"a0","a1","a2","a3a","a3b","cs","a4"}]
          /\ turn \in {0,1}
          /\ flag \in [{0,1} -> BOOLEAN]
```

Now we can define the invariant we want the algorithm to satisfy, in order to prove the `MutualExclusion` property. It is defined as the fact that for each process `i`,

- if it is in step "a2", "a3a", "a3b", "cs" or "a4", then its flag is equal to TRUE (i.e. it does want to access to the resource)
- if it is in step "cs" or "a4", then the other process is not in one of these steps, and if the other process is in step "a3a" or "a3b" then process i has priority to access the resource.

```
I == \A i \in {0, 1} :
       /\ (pc[i] \in {"a2", "a3a", "a3b", "cs", "a4"} => flag[i])
       /\ (pc[i] \in {"cs", "a4"})
            => /\ pc[Not(i)] \notin {"cs", "a4"}
               /\ (pc[Not(i)] \in {"a3a", "a3b"}) => (turn = i)
```

Finally, we can define the actual invariant as the conjunction of the previous one and the fact that TypeOK is satisfied.

```
FullInv == TypeOK /\ I
```

4 Checking a simple proof from the Toolbox

As usual when we want to prove that a property is always satisfied (here MutualExclusion), we reason by induction and define a invariant property (here FullInv) such that :

1. the initial state (here Init) satisfies the invariant,
2. the invariant is preserved by the next-state relation (here [Next]_vars),
3. the invariant ensures the correctness property (MutualExclusion).

Proving the first fact is obvious. Let us check that by asking TLAPS to prove the following theorem (recall that we have to indicate explicitly which definitions we want to be usable in a proof) :

```
THEOREM InitFullInv == Init => FullInv
  BY DEF Init, FullInv, TypeOK, I, ProcSet
```

Now, in the Toolbox, put the cursor on the first line of that theorem and then right-click on "Prove step or module". The Toolbox then colors that theorem in green, which means that it is proved (in this case, Zenon has succeeded in finding a proof).

5 The main theorem

Let us now focus fact number 2 above. For proving that fact it is easier to an intermediate lemma, which states that property TypeOK is preserved by the next-state relation.

```
LEMMA TypeCorrect == TypeOK /\ Next => TypeOK'
  <1>1. ASSUME NEW i \in {0,1}
        PROVE  TypeOK /\ proc(i) => TypeOK'
    BY DEFS TypeOK, proc, a0, a1, a2, a3a, a3b, cs, a4, Not
  <1>. QED
    BY <1>1, NeverDone DEF Next
```

When asking TLAPS to prove the theorem above, you can notice that step <1>1 gets first colored black. That means that Zenon has been trying to prove it for more than three seconds. Then it gets colored red. That means that Zenon has failed to find a proof with the default timeout (ten seconds). Finally, it gets colored green since Isabelle succeeds in proving it.

Finally, let us prove the second item of section 4.

```
THEOREM Inductive == TypeOK /\ I /\ Next => I'
```

We first rephrase the sequent to be proved in a more perspicuous form :

```
<1>1. ASSUME TypeOK,
             I,
             NEW i \in {0,1},
             proc(i)
      PROVE  I'
```

We then assert `TypeOK'` (that will be useful in the following of the proof) :

```
  <2>0. TypeOK'
```

We then split the result we want to prove into three assertions :

```
  <2>1. ASSUME NEW j \in {0,1},
               pc'[j] \in {"a2", "a3a", "a3b", "cs", "a4"}
        PROVE  flag'[j]

  <2>2. ASSUME NEW j \in {0,1},
               pc'[j] \in {"cs", "a4"}
        PROVE pc'[Not(j)] \notin {"cs", "a4"}

  <2>3. ASSUME NEW j \in {0,1},
               pc'[j] \in {"cs", "a4"},
               pc'[Not(j)] \in {"a3a", "a3b"}
        PROVE turn' = j
```

Those three assertions can be proved by case on the step in which process `i` is. If you ask TLAPS to prove that theorem, you can notice that all those proof-steps are colored yellow. That means that the proof is missing. It's up to you now to make them green...