AltaRica Examples

Proof of Peterson's algorithm for mutual exclusion*

A. Griffault, G. Point
LaBRI - CNRS - Université Bordeaux

December 16, 2018

Contents

Mutual exclusion (or critical section access) is a classical problem in the domain of distributed algorithms. This problem is encountered in the domain of operating systems where several processes compete to get resources that are not shareable. It is also a common issue in concurrent programming.

*This example is extracted from **A**ltaRica **H**andbook.

1 The critical section problem

We consider N processes whose only allowed instructions are read or write of a memory word (i.e. these actions can not be interrupted by the processor) execute the following program:

```
1  begin
2    while true do
3     begin
4          // Enter critical section
5          execute critical code
6          // Leave critical section
7          execute non critical code
8     end
9  end
```

The problem to solve is to give access to critical section (CS), line 5, to all processes. A correct solution must satisfy at least the first two following requirements. The third one is more difficult to obtain.

Mutual exclusion One and only one process execute the CS;

Liveness If two or more processes try to enter their CS then one of them eventually enters its CS.

Freedom from starvation If a process is trying to enter its critical section, it will eventually succeed.

2 Peterson's algorithm

A well-known solution to mutual exclusion problem has been proposed by Gary Peterson[2]. Intuitively, a level between 0 and $N-1$, is associated to each process. To obtain the access to the CS, processes have reach the highest level.

Let N be the number of running processes. The algorithm uses two arrays of integers:

`Q[1..`N`]` indicates the current level of each process. Elements of `Q` belong to the range `[0..`$N-1$`]`.

`Turn[1..`$N-1$`]` stores the last process that has reach a level. Elements of `Turn` belong to the range `[1..`N`]`.

```
const N
shared variables
  Q : array[1..N] of [0..N-1];
  turn : array[1..N-1] of [1..N];
```

In this algorithm, processes *climb* a ladder with $N-1$ steps. When several processes try to pass the same step, at least one of them do not move. The algorithm for the i-th process is the following:

```
begin
  while true do
  begin
    for j from 1 to N-1 do
    begin
      Q[i] = j;
      Turn[j] = i;
```

```
        wait until ((for all k ≠ i, (Q[k] < j)) ou (Turn[j]≠i))
      end
      execute critical code
      Q[i] := 0;
      execute non critical code
    end
  end
```

3 The model

3.1 What kind of modelling ?

As indicated by the title of this *exercice*, we have to prove the correctness of an algorithm. Properties proved on the model will be satisfied by the algorithm only if we have reproduced accurately the algorithm in the model.

Thus, the description is very close from the algorithm and one of the major issue is to determine are store shared variables in the AltaRica hierarchy.

3.2 Shared variables

The algorithm uses two kind of shared variables:

read/write access The variable `Q[`i`]` which indicates current level of process i is modified only by the i-th process; other processes just read the variable.

write/write access The variable `Turn[`j`]` which indicates the number of the last process that reached level j is modified by all processes.

`Q[`i`]` can be handled using a state variable of the i-th process. The value of this variable is then made available to other processes with a flow variable.

The variable `Turn[`j`]` can not be handled in the same manner as a state variable of the i-th process because other processes could not modify it. This variable is thus a state variable of the hierarchical level that embeds all processes and its changes will be described using synchronizations.

In the sequel variables will take their values in following domains:

```
const N = <?= $N ?>;
domain ProcessID = [1, N];
domain ProcessLevel = [0, N-1];
```

Note the use of PHP code. In the sequel of this example, `arc` requests PHP preprocessing for each loaded file. `php` is invoked using, for instance, following settings (here we have set the number of processes to 4):

```
set arc.shell.preprocessor.default.command \
  "php -d short_open_tag=On -r '$NB_PROCESSES = 4; include ($argv[1]); ?>'"
```

3.3 Model of a process

AltaRica tools, among which `arc`, does not support parameterized models. However, using PHP preprocessor, one can describe in a generic way Peterson'algorithm for any number of processes $N > 1$.

To describe the algorithm by means of AltaRica transitions we identify some of its program points. In the model we will use the following domain:

```
domain ProgramCounter = { A, B, C, D };
```

Intuitively, these program points are located in the algorithm as follows:

```
1  begin
2    while true do
3    begin
4      for j from 1 to N−1 do
5      begin
6        // program counter = A
7        Q[i] = j;
8        // program counter = B
9        Turn[j] = i;
10       // program counter = C
11       wait until ((for all k ≠ i, (Q[k] < j)) ou (Turn[j]≠i))
12     end
13     // program counter = D
14     execute critical code
15     Q[i] := 0;
16     execute non critical code
17   end
18 end
```

The AltaRica model of the algorithm can be generated "easily" for any integer $N > 1$ using PHP preprocessing. Since PHP code is neither readable nor very interesting we present in next sections the model for $N = 3$.

The node for `Process`es is depicted on listing 1. It starts with the declaration of parameters, variables and events.

The parameter `ident` is used by the top-level node to specify to each process its identifier. This parameter is compiled as a constant value and add no cost to the model.

Then we find shared variables of the algorithm `Q` and `Turn`. As we mention in above section each process is allowed to modified its associated level in `Q` and only this variable. The level of the process is represented by the state variable `j`. Assertion at line 10 makes current level readable by other processes by enforcing the quality of variables `Q`$[i]$ and j. `Turn` array is not constrained by `Process` nodes. We use a private variable, `wait`, to store the truth value of the waiting condition of the algorithm; this truth value is defined by assertion at line 12. Finally variable `pc` stores current program counter of the algorithm.

Events describe following actions:

- `reqCS` models the fact that the process wants to access its CS; this event corresponds to the assignment of `Q`$[i]$.
- `setTurn`$[l]$ indicates that this process is the last one having reached level l. Since `Turn` variable can not be modified by `Process`es, `setTurn`$[l]$ is synchronized by upper node.
- `enterCS` labels transitions that enter the critical section at program counter `D`.
- `run` models the execution of critical section and the loop to the entrypoint of the algorithm.

The declaration of events is followed by transitions between program points. Note transitions labelled by `setTurn`$[l]$ that have to be synchronized with transitions of the main node that assigns `Turn` variable.

```
1  node Process
2    param ident : ProcessID;
3    flow  Q    : ProcessLevel[N];
4          Turn : ProcessID[N−1];
5          wait : bool : private;
6    state pc : ProgramCounter;
7          j : ProcessLevel;
8    init  pc := A, j := 0;
9    assert
10     // export current level
```

```
11      Q[ident-1] = j;
12      // waiting condition
13      wait = not ((     ((1 != ident) => Q[0] < j)
14                    and ((2 != ident) => Q[1] < j)
15                    and ((3 != ident) => Q[2] < j))
16              or  (j = 1 & Turn[0] != ident)
17              or  (j = 2 & Turn[1] != ident));
18
19    event reqCS, setTurn[N-1], enterCS, run;
20    trans
21      pc = A |- reqCS -> pc := B, j := j + 1;
22      pc = B and j = 1 |- setTurn[0] ->  pc := C;
23      pc = B and j = 2 |- setTurn[1] ->  pc := C;
24      pc = C and j < 2 and not wait |- run -> pc := A;
25      pc = C and j = 2 and not wait |- enterCS -> pc := D;
26      pc = D |- run ->  pc := A, j := 0;
27  edon
```

Listing 1: Model of a process for $N = 3$

3.4 Putting all processes together

Listing 2 is the model of environment of processes. It mainly describes interactions between the three processes.

After the declaration of processes[1] P_is, the `Main` node assigns (line 4) identifiers to `Process` nodes by defined the value of `ident` parameters. The shared variables are then declared. As mentionned above, only `Turn` contains state variables while `Q` gathers shared flow variables.

Assertions describe the sharing of variables `Q` and `Turn`.

The assignments of `Turn` variables is then described. First the model, line 22, enumerates events that represents the assignment of each cell: an event `turn`l`To`i models the assigment `Turn[`l`] := `i. Corresponding transitions follow.

Finally line 34assignments of `Turn` variables are synchronized according to the event `setTurn[`i`]` of each process.

```
1   node Main
2     sub
3       P1, P2, P3 : Process;
4     param_set // assigning identifiers to processes
5       P1.ident := 1,
6       P2.ident := 2,
7       P3.ident := 3;
8     flow
9       Q : ProcessLevel[N];
10    state
11      Turn : ProcessID[N-1];
12    init
13      Turn[0] := 1,
14      Turn[1] := 1;
15    assert
16      P1.Turn = Turn;
17      P1.Q = Q;
18      P2.Turn = Turn;
19      P2.Q = Q;
20      P3.Turn = Turn;
21      P3.Q = Q;
22    event // Assignments of Turn variables
23      turn1To1, turn1To1, turn1To2, turn1To3;
24      turn2To1, turn2To1, turn2To2, turn2To3;
25    trans
26      true |- turn1To1 -> Turn[0] := 1;
27      true |- turn1To2 -> Turn[0] := 2;
28      true |- turn1To3 -> Turn[0] := 3;
```

[1] Here we do not use an array of `Process` nodes to permit be able to assign distinct values to `ident` parameters.

```
29
30      true |- turn2To1 -> Turn[1] := 1;
31      true |- turn2To2 -> Turn[1] := 2;
32      true |- turn2To3 -> Turn[1] := 3;
33
34    sync //Synchronization of assignments of Turn variables
35      <turn1To1,P1.setTurn[0]>;
36      <turn1To2,P2.setTurn[0]>;
37      <turn1To3,P3.setTurn[0]>;
38
39      <turn2To1,P1.setTurn[1]>;
40      <turn2To2,P2.setTurn[1]>;
41      <turn2To3,P3.setTurn[1]>;
42
43  edon
```

Listing 2: Model of environment of processes for $N = 3$

4 Checking the model

4.1 Validation

Before checking that our model possess expected properties we make some validation tests. First of all we use the **validate** command of **arc** to verify that all elements of the models are actually used when computing its semantics. For $N = 2$, the tools produces the results below.

arc creates, for each parameter, a state variable that stores the value of the parameter. These state variables are never modified. This explains the warnings related to parameters. Note that for $N = 2$ our PHP code produces also unused transitions that are mentionned by the command **validate**.

```
basic properties checking for node 'Main'
 there is 20 configurations.
 usage of variables
  state variable 'P2.ident' is never used (asserts and trans).
  state variable 'P2.ident' is not assigned by a transition.
  state variable 'P1.ident' is never used (asserts and trans).
  state variable 'P1.ident' is not assigned by a transition.

 uniqueness of initial configuration
  The system has only one initial configuration

 coverage of domains / configurations
  Domains of variables are covered by the set of configurations.

 coverage of domains / reachables
   State variable 'P2.ident' does not cover its domain.
   Missing values (restricted to configurations) verify:
    (P2.ident = 1)
   State variable 'P1.ident' does not cover its domain.
   Missing values (restricted to configurations) verify:
    (P1.ident = 2)

 usage of macro-transitions
  not P2.wait and ((P2.j < 1)) and ((P2.pc = C)) |- P2.run -> P2.pc
      := A is never triggered.
  not P1.wait and ((P1.pc = C)) and ((P1.j < 1)) |- P1.run -> P1.pc
      := A is never triggered.
```

After this first checking, we compute some elementary properties of the algorithm:

Absence of deadlock We check that there is no sink configuration. Such configuration has no output transition except the one labelled by ϵ event. We have to check that the following set is empty:

```
deadlock := any_s - src (any_t - self_epsilon);
```

Resetability We check that the initial state is reachable from any other configuration. In other words, any configuration is co-accessible from the initial one. We have to check that the following set is empty:

```
nonreset := any_s - coreach (initial, any_t);
```

In order to make formula clearer now and in the sequel, we define the following sets of configuration for each process identifier i:

```
PiReqCS := any_s and tgt (label Pi.reqCS);
PiInCS := any_s and tgt (label Pi.enterCS);
```

P$_i$ReqCS is the set of configuration frtom which process P$_i$ has requested to enter its CS. P$_i$InCS is the set of configuration where process P$_i$ is in its critical section.

Then sets, `ReqCS` and `InCS`, defined below are the sets where, respectively, some process has requested access to its CS and some process is in its CS.

```
ReqCS := P1ReqCS or ... or PNReqCS;
InCS := PNInCS or ... or PNInCS;
```

4.2 Mutual exclusion

As usual, we check properties by verifying the emptyness of the set of configurations that do not satisfy the property. The *mutual exclusion* property is an invariant that must be fulfilled by all configurations. A configuration does not satisfy the invariant if at least two processes are in their respective CS. The property is thus falsified if for some $i \neq j$ the intersection of PiInCS and PjInCS is not empty.

We request `arc` to compute the union of all such intersections. For instance, for $N = 4$ we have specified:

```
mutex :=
   (P1InCS and (P2InCS or P3InCS or P4InCS)) or
   (P2InCS and (P3InCS or P4InCS)) or
   (P3InCS and (P4InCS));
test (mutex, 0) >> "peterson4.res";
```

4.3 Liveness

To prove liveness property we enumerate set of processes that try to access their critical section and forbid others to have a concurrent access. For each set of concurrent processes we compute the set of configurations such that:

- only these processes have requested access to their CS; others stay at program point `A`;
- there exists a path that infinitely avoid entering in CS by one of these processes.

The specification of this property in `arc` uses the function `unav(T, S)`; a mnemonic for *unavoidable*. This builtin function computes the set of configurations U from which all paths that follow transitions on T eventually pass by a configuration in S. In other words, configurations in U can not avoid the set S when using only transitions in T.

Let $I \subseteq \{1, \ldots, N\}$ be identifiers of processes. The set of configurations where only these processes try to access their CS is:

$$\bigcap_{i \in I} \mathtt{P}i\mathtt{ReqCS} \cap \bigcap_{i \notin I} [\mathtt{P}i.\mathtt{pc} = \mathtt{A} \wedge \mathtt{P}i.\mathtt{j} = 0]$$

N	S	T	deadlock	nonreset	mutex	liveness	starvation
2	20	34	0	0	0	0	0
3	417	945	0	0	0	0	186
4	9272	25792	0	0	0	0	5620
5	223105	741065	0	0	0	0	157175

Table 1: Results computed by `arc` for 2 to 5 processes.

Liveness property is falsified if from this set of configurations the set $\cup_{i \in I}$PiInCS is avoided by at least one path along which processes that are not identified by I do not try to enter their CS.

For $N = 3$ the request sent to `arc` is the following:

```
liveness :=
  ((P1ReqCS and P2ReqCS and [P3.pc = A & P3.j = 0])
    - unav (any_t-(self_epsilon or label P3.reqCS),
            P1InCS or P2InCS)) or
  ((P1ReqCS and [P2.pc = A & P2.j = 0] and P3ReqCS)
    - unav (any_t-(self_epsilon or label P2.reqCS),
            P1InCS or P3InCS)) or
  (([P1.pc = A & P1.j = 0] and P2ReqCS and P3ReqCS)
    - unav (any_t-(self_epsilon),
            P1InCS or P2InCS or P3InCS)) or
  ((P1ReqCS and P2ReqCS and P3ReqCS)
    - unav (any_t-(self_epsilon),
            P1InCS or P2InCS or P3InCS));
test (liveness, 0) >> "peterson3.res";
```

4.4 Freedom from starvation

Starvation occurs if there exists a process that tries to enter in CS but is always bypassed by others. To check if a process, say Pi, is starved we have to verify that from each configuration from which Pi has started to access CS all paths pass by a configuration where Pi has entered its CS.

Yet we use `unav` operation to filter from PiReqCS configurations that can avoid PiReqCS. Obviously we disallow self-loops on ϵ event. For each process Pi the following set should be empty:

```
PiReqCS - unav (any_t-self_epsilon, PiInCS)
```

To check freedom from starvation when, for instance, $N = 3$, `arc` check emptyness following set:

```
starvation :=
 (P1ReqCS - unav (any_t-(self_epsilon), P1InCS)) or
 (P2ReqCS - unav (any_t-(self_epsilon), P2InCS)) or
 (P3ReqCS - unav (any_t-(self_epsilon), P3InCS));
```

4.5 Results

We have checked properties of Peterson's algorithm from 2 to 5processes. Following table summarizes results. Columns S and T give, respectively, the number of configurations and transitions of the model. Next columns give the number of states of computed properties presented in above sections.

Last column seems indicate that for $N \geq 3$ starvation exists. If we request `arc` to generate a counter-example[2] that shows existence of a starved process for $N = 3$, we obtain graph on figure 1 page 10.

[2]Here we have use CTL* engine to produce this counter-example.

The counter example show that processes `P2` and `P3` alternatively enter their respective CS. The infinite loop depicted on figure 1 shows that `P1` can not trigger any `setTurn` transition because as soon as `P2` (or `P3`) terminates its critical section it requests a new access to its CS.

Actually it is known that Peterson's algorithm does not guaranteed bounded waiting[1] which means that without fairness constraint some process, like `P1` in our example, can be postponed an unbounded amount of time until it enters its CS. However if we constraint processes to not try to reenter CS just after leaving it then freedom from starvation is obtained.

In order to apply above constraint we define the set `ReqAfterRun` of transitions labelled by some `Pi.reqCS` that triggered just after a transition `Pi.run` i.e. that leaves the CS. For $N = 3$ this set is specified as follows:

```
ReqAfterRun :=
 (label P1.reqCS & rsrc(tgt (label P1.run))) or
 (label P2.reqCS & rsrc(tgt (label P2.run))) or
 (label P3.reqCS & rsrc(tgt (label P3.run)));
```

Then we forbid these transitions on paths that should not avoid each `Pi` to enter its CS:

```
fixedstarvation :=
 (P1ReqCS — unav (any_t—(self_epsilon|ReqAfterRun), P1InCS)) or
 (P2ReqCS — unav (any_t—(self_epsilon|ReqAfterRun), P2InCS)) or
 (P3ReqCS — unav (any_t—(self_epsilon|ReqAfterRun), P3InCS));
test (fixedstarvation, 0) >> "peterson3.res";
```

Under this constraint results are what was expected; for $N = 3$, `arc` displays:

```
TEST(deadlock,0)         [PASSED]
TEST(nonreset,0)         [PASSED]
TEST(mutex,0)    [PASSED]
TEST(liveness,0)         [PASSED]
TEST(starvation,0)       [FAILED] actual size = 186
TEST(fixedstarvation,0) [PASSED]
```

References

[1] K. Alagarsamy. Some myths about famous mutual exclusion algorithms. *SIGACT News*, 34(3):94–103, September 2003.

[2] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.

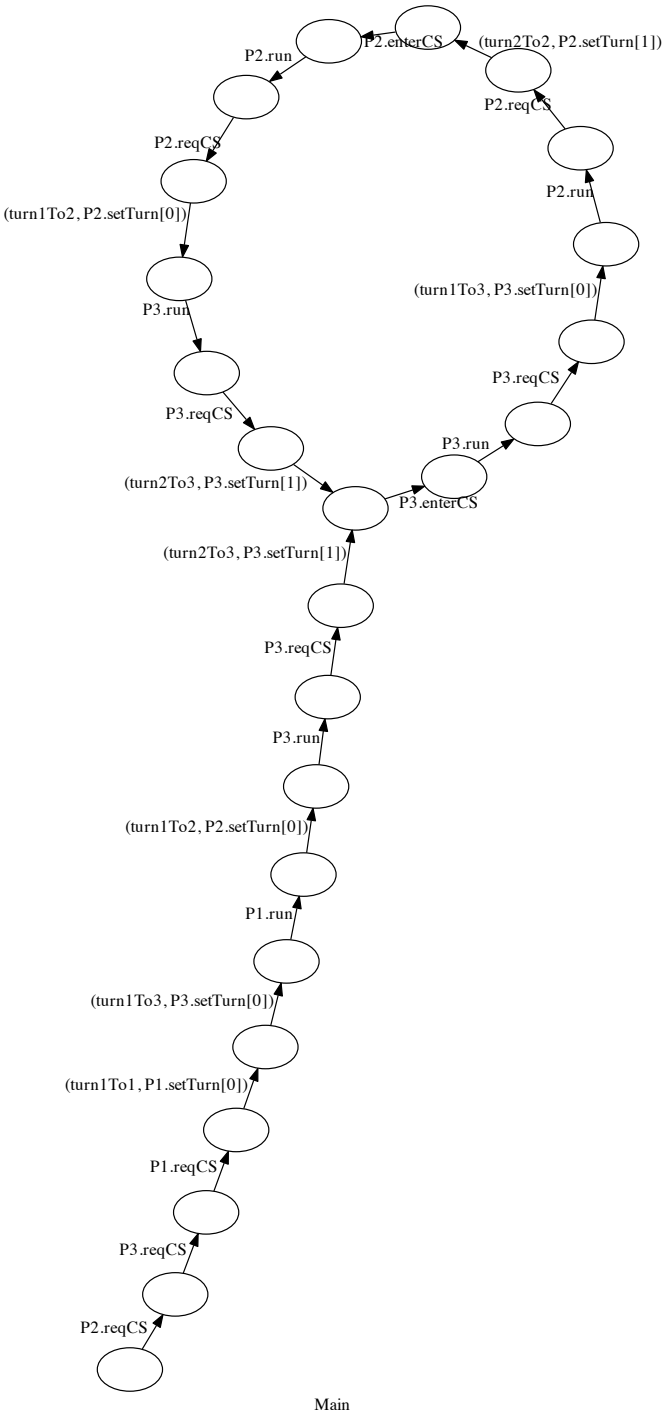

Figure 1: Counter-example of freedom from starvation