Proof of Peterson’s algorithm for mutual exclusion*

A. Griffault, G. Point
LaBRI - CNRS - Université Bordeaux
January 29, 2016

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 AltaRica Handbook.
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:

```
begin
  while true do
    begin
      // Enter critical section
      execute critical code
      // Leave critical section
      execute non critical code
    end
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 \( [0..N-1] \).
- \( \text{Turn}[1..N-1] \) stores the last process that has reach a level. Elements of \( \text{Turn} \) belong to the range \( [1..N] \).

```
const N = shared variables
Q : array[1..N] of [0..N-1];
\text{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
          \( Q[i] = j;\)
          \( \text{Turn}[j] = i;\)
```
wait until \((\text{for all } k \neq i, (Q[k] < j) \text{ ou } (\text{Turn}[j] \neq i))\)
end

critical code

\(Q[i] := 0;\)

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 \(\text{Turn}[j]\) 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 \(\text{Turn}[j]\) 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** \(N = \{0, 1, \ldots, N\};\)
- **domain** \(\text{ProcessID} = [1, N];\)
- **domain** \(\text{ProcessLevel} = [0, N-1];\)

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):

```plaintext
set arc.shell.preprocessor.default.command \n"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** \(\text{ProgramCounter} = \{ A, B, C, D \};\)
Intuitively, these program points are located in the algorithm as follows:

```plaintext
begin
while true do
begin
  for j from 1 to N-1 do
  begin
    // program counter = A
    Q[i] = j;
    // program counter = B
    Turn[j] = i;
    // program counter = C
    wait until ((for all k ≠ i, Q[k] < j) or (Turn[j] ≠ i))
  end
  // program counter = D
execute critical code
  Q[i] := 0;
execute non critical code
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 Processes 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 15 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 17. 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 Processes, 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.

```plaintext
node Process
param ident : ProcessID;
flow Q : ProcessLevel[N];
  Turn : ProcessID[N-1];
  wait : bool : private;
state pc : ProgramCounter;
  j : ProcessLevel;
init pc := A, j := 0;
assert // export current level
```
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's, the Main node assigns (line 37) identifiers to Process nodes by defining the value of ident parameters. The shared variables are then declared. As mentioned 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 55, enumerates events that represents the assignment of each cell: an event turnToN models the assignment Turn[i] := i. Corresponding transitions follow.

Finally line 67 assignments of Turn variables are synchronized according to the event setTurn[i] of each process.

\(^1\)Here we do not use an array of Process nodes to permit be able to assign distinct values to ident parameters.
true | turn2To1 -> Turn[1] := 1;
true | turn2To2 -> Turn[1] := 2;
true | turn2To3 -> Turn[1] := 3;

sync // Synchronization of assignments of Turn variables
<turn1To1, P1.setTurn[0]>
<turn1To2, P2.setTurn[0]>
<turn1To3, P3.setTurn[0]>
<turn2To1, P1.setTurn[1]>
<turn2To2, P2.setTurn[1]>
<turn2To3, P3.setTurn[1]>

eden

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 mentioned 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 \( \epsilon \) 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:
\[
\text{nonreset} := any.s \subseteq \text{coreach } (\text{initial, any.t});
\]
In order to make formula clearer now and in the sequel, we define the following
sets of configuration for each process identifier \(i\):
\[
P_i\text{ReqCS} := any.s \text{ and tgt (label } P_i.\text{reqCS)};
\]
\[
P_i\text{InCS} := any.s \text{ and tgt (label } P_i.\text{enterCS)};
\]
\(P_i\text{ReqCS}\) is the set of configuration from which process \(P_i\) has requested to enter
its CS, \(P_i\text{InCS}\) is the set of configuration where process \(P_i\) is in its critical section.
Then sets, \(\text{ReqCS}\) and \(\text{InCS}\), defined below are the sets where, respectively, some
process has requested access to its CS and some process is in its CS.
\[
\text{ReqCS} := \bigcup_{i=1}^{N} P_i\text{ReqCS} \text{ or } \ldots \text{ or } P_N\text{ReqCS};
\]
\[
\text{InCS} := P_N\text{InCS} \text{ or } \ldots \text{ or } P_N\text{InCS};
\]

### 4.2 Mutually exclusion
As usual, we check properties by verifying the emptiness 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 \(P_i\text{InCS}\) and \(P_j\text{InCS}\) is not empty.
We request \texttt{arc} to compute the union of all such intersections. For instance, for
\(N = 4\) we have specified:
\[
\text{mutex} := \\
(P1\text{InCS and (P2InCS or P3InCS or P4InCS)}) \text{ or } \\
(P2\text{InCS and (P3InCS or P4InCS)}) \text{ or } \\
(P3\text{InCS and (P4InCS)});
\]
\texttt{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 \(k\);
- there exists a path that infinitely avoid entering in CS by one of these processes.

The specification of this property in \texttt{arc} uses the function \texttt{unav(T, S)}; a mnemonic
for unavoidable. This built-in 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:
\[
\bigcap_{i \in I} P_i\text{ReqCS} \cap \bigcap_{i \in I} \big[ P_i.\text{pc} = A \land P_i.j = 0 \big]
\]
Table 1: Results computed by arc for 2 to 5 processes.

<table>
<thead>
<tr>
<th>N</th>
<th>S</th>
<th>T</th>
<th>deadlock</th>
<th>nonreset</th>
<th>mutex</th>
<th>liveness</th>
<th>starvation</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>20</td>
<td>34</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>417</td>
<td>945</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>186</td>
</tr>
<tr>
<td>4</td>
<td>9272</td>
<td>25792</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5620</td>
</tr>
<tr>
<td>5</td>
<td>223105</td>
<td>741065</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>157175</td>
</tr>
</tbody>
</table>

Liveness property is falsified if from this set of configurations the set $\bigcup_{i \in I} P_i \text{InCS}$ 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:

\[
\text{liveness} :=
\begin{align*}
&((\text{P1ReqCS and P2ReqCS and [P3.pc = A & P3.j = 0]}) \\
&- \text{unav (any.$\tau$-(self.$\epsilon$ision or label P3.reqCS), P1InCS or P2InCS)) or} \\
&((\text{P1ReqCS and [P2.pc = A & P2.j = 0] and P3ReqCS}) \\
&- \text{unav (any.$\tau$-(self.$\epsilon$ision or label P2.reqCS), P1InCS or P3InCS)) or} \\
&((\{P1.pc = A & P1.j = 0\} and P2ReqCS and P3ReqCS) \\
&- \text{unav (any.$\tau$-(self.$\epsilon$ision), P1InCS or P2InCS or P3InCS)) or} \\
&((\text{P1ReqCS and P2ReqCS and P3ReqCS}) \\
&- \text{unav (any.$\tau$-(self.$\epsilon$ision), P1InCS or P2InCS or P3InCS))};
\end{align*}
\]

\[
\text{test (liveness, 0) \gg \"peterson3.res\"};
\]

\subsection{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 $P_i$, is starved we have to verify that from each configuration from which $P_i$ has started to access CS all paths pass by a configuration where $P_i$ has entered its CS.

Yet we use $\text{unav}$ operation to filter from $P_i\text{reqCS}$ configurations that can avoid $P_i\text{reqCS}$. Obviously we disallow self-loops on $\epsilon$ event. For each process $P_i$ the following set should be empty:

\[
P_i\text{reqCS} - \text{unav (any.$\tau$-(self.$\epsilon$ision, P1InCS)}
\]

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

\[
\text{starvation} :=
\begin{align*}
&((\text{P1ReqCS} \quad \text{unav (any.$\tau$-(self.$\epsilon$ision), P1InCS}) \quad \text{or}) \\
&((\text{P2ReqCS} \quad \text{unav (any.$\tau$-(self.$\epsilon$ision), P2InCS}) \quad \text{or}) \\
&((\text{P3ReqCS} \quad \text{unav (any.$\tau$-(self.$\epsilon$ision), P3InCS)}));
\end{align*}
\]

\subsection{4.5 Results}

We have checked properties of Peterson’s algorithm from 2 to 5 processes. 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\footnote{Here we have use CTL* engine to produce this counter-example.} that shows existence of a starved process for $N = 3$, we obtain graph on figure 1 page 10.
The counter example show that processes $P_2$ and $P_3$ alternatively enter their respective CS. The infinite loop depicted on figure 1 shows that $P_1$ can not trigger any `setTurn` transition because as soon as $P_2$ (or $P_3$) terminates its critical section it requests a new access to its CS.

Actually it is known that Peterson’s algorithm does not guarantee bounded waiting[1] which means that without fairness constraint some process, like $P_1$ 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 $P_i.reqCS$ that triggered just after a transition $P_i.run$ i.e. that leaves the CS. For $N = 3$ this set is specified as follows:

$$\text{ReqAfterRun} := (\text{label } P_1.\text{reqCS} & \text{rsrc(tgt (label } P_1.\text{run}) )) \text{ or (label } P_2.\text{reqCS} & \text{rsrc(tgt (label } P_2.\text{run}) )) \text{ or (label } P_3.\text{reqCS} & \text{rsrc(tgt (label } P_3.\text{run}) ));$$

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

$$\text{fixedstarvation} := (P_1\text{ReqCS} - \text{unav (any } t - (\text{self } \epsilon | \text{ReqAfterRun}, P_1\text{InCS}) )) \text{ or (P_2\text{ReqCS} - \text{unav (any } t - (\text{self } \epsilon | \text{ReqAfterRun}, P_2\text{InCS}) )) \text{ or (P_3\text{ReqCS} - \text{unav (any } t - (\text{self } \epsilon | \text{ReqAfterRun}, P_3\text{InCS}) )); test \ (\text{fixedstarvation, 0}) \gg "peterson3.res";$$

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


Figure 1: Counter-example of freedom from starvation