CAS CS 512
Surya Vajjhala {vajjhala @ bu.edu}

Project Report

Formal Verification and Implementation of Peterson's Algorithm for Mutual Exclusion

The Problem

In a distributed program, a number of different components are simultaneously executed, but the critical section is only accessible to one component at a time.

The problem is to find a *protocol* to allow only one of the competing components access to the critical section.
Peterson's mutual exclusion algorithm is one of the many algorithms that solves this problem.

The properties we need to check for are:

1. **Safety**: Only one process is in the critical section at any time.
2. **Liveness**: Every requesting process must eventually get access to the critical section.

Approach

I will start with the primitive algorithm for mutual exclusion, which is the the Safe Sluice algorithm, show how it fails to address deadlock, then extract the Peterson's algorithm from the *safety* condition.

The algorithm I will be arguing about will be a variant of the Peterson's algorithm that synchronises only two components, but the general version extends this to n components.

At the end of the report I have presented the pseudo-code for the **N**-component variant without proof. It is this version that I have implemented.

Notation

The pseudo-code used Dijkstra's Guarded Command Language

skip does nothing

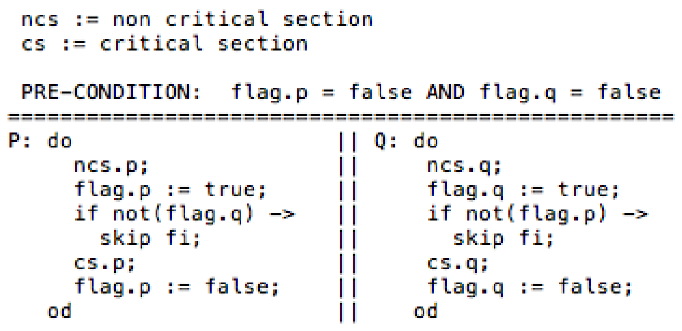
x := E Assignment of evaluated expression E to x

if B0 $\rightarrow$ S0 $\|$ B1 $\rightarrow$ S1 fi "Alternative Construct" Execute S0 if B0 is `true`, execute S1 if B1 `true`; terminate only if one of the guarded conditions is satisfied

do B0 $\rightarrow$ S0 $\|$ B1 $\rightarrow$ S1 od "Repetitive Construct" or "Loop", keep executing S0 or S1 as long as the corresponding guards evaluate to `true`. Terminates when both are `false`

Safe Lock or Safe Sluice algorithm

The idea behind the safe lock is based on the analogy with the locks that sit in dikes where boats sail in opposite direction. The captain of one boat uses a flag to signal to the other boat. The other boat will only sail when no flag is signalled.



```
 ncs := non critical section
 cs := critical section

 PRE-CONDITION:  flag.p = false AND flag.q = false
====================================================
P: do                      || Q: do
     ncs.p;                ||      ncs.q;
     flag.p := true;       ||      flag.q := true;
     if not(flag.q) ->     ||      if not(flag.p) ->
       skip fi;            ||        skip fi;
     cs.p;                 ||      cs.q;
     flag.p := false;      ||      flag.q := false;
   od                      ||    od
```

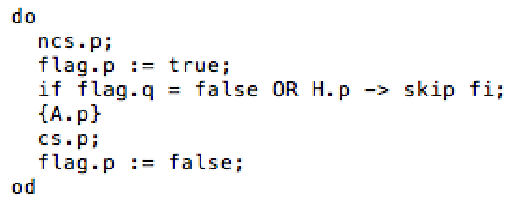
The safe lock ensures only one component can have access to the critical section. But you can have a situation where both the **flag.p** and **flag.q** are both set to **true** and you can have a deadlock.

Extracting Peterson's

For this reason we introduce a new condition $H.p$ into the **if** statement. The idea here is to weaken the guard so that total deadlock does not occur.

So the current guarded **if** statement is replaced with `if not flag.q ∨ H.p → skip fi`

Lets call the post condition after this **if** statement terminates as **A.p**



```
 do
   ncs.p;
   flag.p := true;
   if flag.q = false OR H.p -> skip fi;
   {A.p}
   cs.p;
   flag.p := false;
 od
```

$A.p \equiv flag.p \wedge (\neg flag.q \vee H.p)$

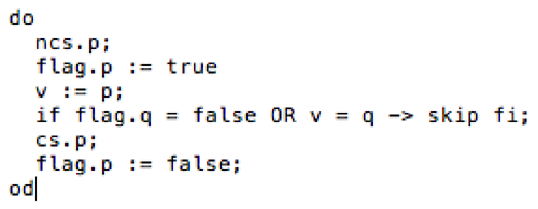
Recall that the *safety* condition states that the "Only one process is in the critical section at any time".
Therefore, what we want is $\Box A.p \wedge A.q \Rightarrow false$, which is the strongest statement we can make about the system state.

$$
\begin{array}{ll}
 & A.p \wedge A.q \\
\equiv & \{\text{definition of A}\} \\
 & flag.p \wedge (\neg flag.q \vee H.p) \wedge flag.q \wedge (\neg flag.p \vee H.q) \\
\equiv & \{\text{predicate calculus}\} \\
 & flag.p \wedge H.p \wedge flag.q \wedge H.q \\
\Rightarrow & \{\text{we cannot infer anything about } \mathit{flag}.\text{p or } \mathit{flag}.\text{q}\} \\
 & H.p \wedge H.q \\
\Rightarrow & \{\text{which is to be proven}\} \\
 & false
\end{array}
$$

Now because $H.p$ occurs as a disjunct in the guard, progress is best ensured by choosing $H.p$ as weak as possible, or $\neg H.p$ as strong as possible. A choice we can make for this is to introduce a new variable v as the following:

$\neg H.p : v = p,$ or
$H.p : v \neq p \Rightarrow v = q$

Now by replacing this new definition into the pseudo-code, we arrive at:



```
do
  ncs.p;
  flag.p := true
  v := p;
  if flag.q = false OR v = q -> skip fi;
  cs.p;
  flag.p := false;
od
```

This is the Peterson's Algorithm which is executed by each concurrent process P.

Check for Deadlock

In the case of the safe lock algorithm, when both **flags** were set to **true**, we had a deadlock.

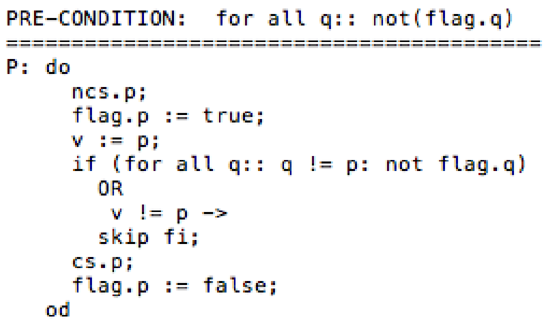
We now follow the order of execution dictated by the Peterson's algorithm and see if in case both the **flags** are set to **true** whether or not we have a deadlock.

$$
\begin{array}{l}
\{\neg \text{flag}.p \wedge \neg \text{flag}.q\} \\
\quad Q : v := q \\
\{\neg \text{flag}.p \wedge \neg \text{flag}.q \wedge v = q\} \\
\quad P : v := p \\
\{\neg \text{flag}.p \wedge \neg \text{flag}.q \wedge v = p\} \\
\quad P : \text{flag}.p := true \\
\{\text{flag}.p \wedge \neg \text{flag}.q \wedge v = p\} \\
\quad P : \text{if} \neg \text{flag}.q \vee v = q \rightarrow \text{skip fi} \\
\{\text{flag}.p \wedge \neg \text{flag}.q \wedge v = p\} \\
\quad Q : \text{flag}.q := true \\
\{\text{flag}.p \wedge \text{flag}.q \wedge v = p\} \\
\quad P : \text{if} \neg \text{flag}.p \vee v = p \rightarrow \text{skip fi} \\
\{\text{flag}.p \wedge \text{flag}.q \wedge v = p\} \\
\quad Q : cs.q
\end{array}
$$

Thus we have shown that even though, both the **flags** are set to **true** only one of them gets access to the critical section. Thereby holding the safety condition.

N variant of Peterson's Algorithm

This is the pseudo-code for the N-variant of the Peterson's Algorithm where you have N components trying to get access to the critical section. I merely state the pseudo-code here, [1, p. 275] but the proof is similar to the 2 component case shown above.



```
PRE-CONDITION:  for all q:: not(flag.q)
=========================================
P: do
     ncs.p;
     flag.p := true;
     v := p;
     if (for all q:: q != p: not flag.q)
       OR
        v != p ->
       skip fi;
     cs.p;
     flag.p := false;
   od
```

Reasoning about liveness

Now we address liveness, which means that every system after a finite number of steps will enter the critical section. Unfortunately there is no formal liveness argument that has been published.[1, p. 277] And I have not made an attempt to formally prove it myself. Hence I will try to reason about it informally.

Here we come back to the use of the extra variable *v* introduced. At most one component can be blocked in the guarded skip, thanks to the disjunct $\mathsf{v} \neq p$. Now let a component **P** be blocked indefinitely in this guarded skip. We show that within a finite number of steps the guards will become stably **true**.

There are two different computations that can be generated by the system:

1. Some component **Q** initiates it's *repetitive construct*(**do** { } **od**). It will make $\mathsf{v} \neq p$ **true** through $\mathsf{v} := q$ - and hence component **P**'s *guarded skip* will be **true**.
2. No component **Q** initiates it's *repetitive construct* while component **P** is blocked. In this case the system will generate stable truth of $\langle \forall q : q \neq p : \neg \mathrm{flag}.q \rangle$. The argument is as follows, any component that is outside the *repetitive construct* would have $\neg \mathrm{flag}.q$ through $\mathrm{flag}.q := \mathrm{false}$ which is set before finishing the program fragment.

Through this informal argument liveness is guaranteed.

We have now show that both *safety* and *liveness* are guaranteed by the Peterson's mutual exclusion algorithm.

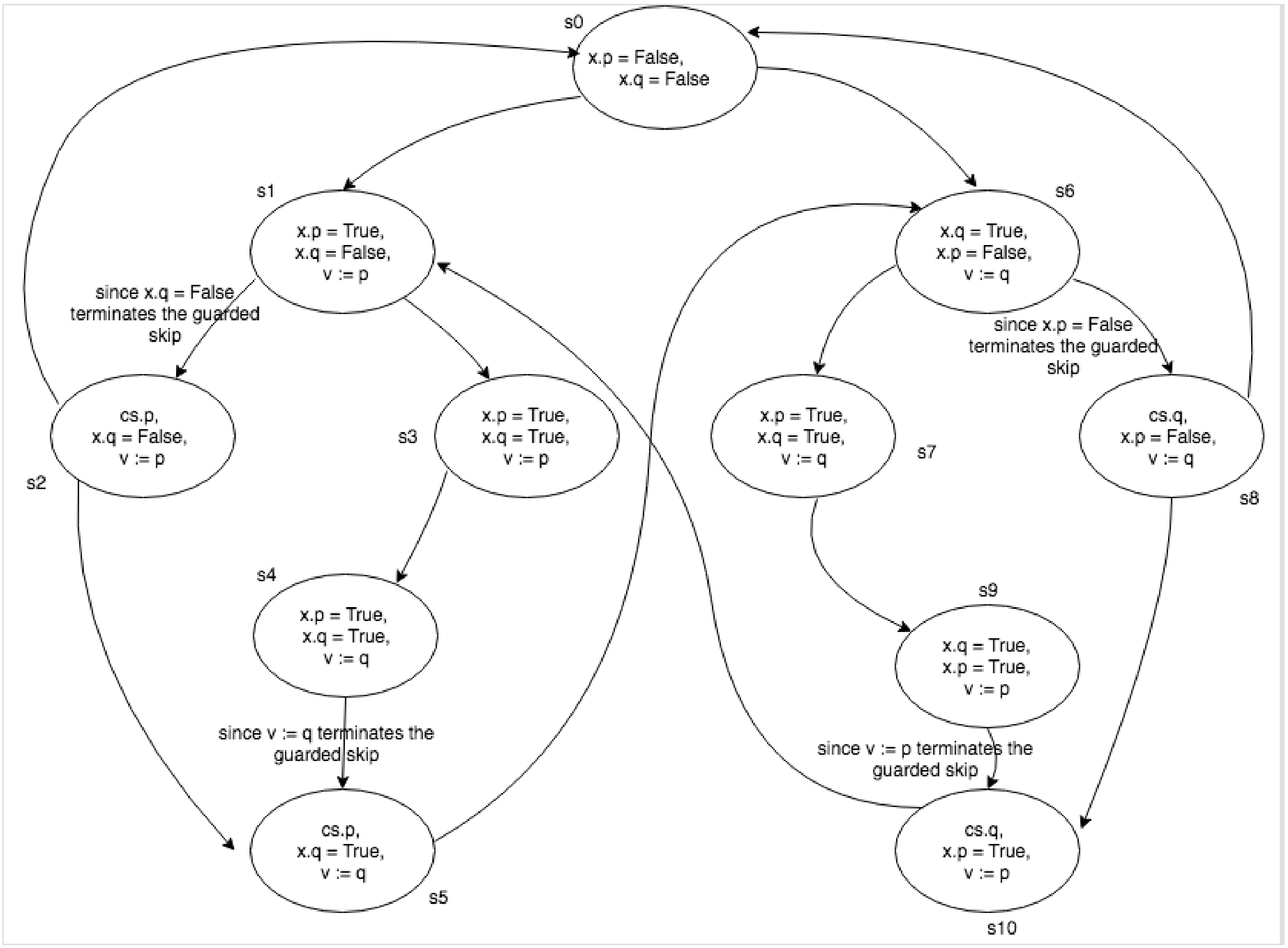
Transition System for the two component case

Instead of *flag.p* and *flag.q* I will be using *x.p* and *x.q* respectively.

There are three variables in question here, which determine the state, *x.p*, *x.q* and *v*. *cs.p* refers to P being in the critical system and *cs.q* refers to Q being in the transition system.

From the transition system we can reason about safety and liveness[2]:

1. **Safety**: $\Box\neg(\text{cs.p} \wedge \text{cs.q})$: We can see that there is no state where both *cs.p* and *cs.q* are satisfied.
2. **Liveness**: The liveness of component **P** is expressed as $\Box\ \text{x.p} \rightarrow \Diamond\ \text{cs.p}$. We can see that starting at state s_0, when *x.p* is *true*, we reach state s_1 and from here both the possible paths lead to *cs.p* via paths $s_1 \rightarrow s_2$ or $s_1 \rightarrow s_3 \rightarrow s_4 \rightarrow s_5$ Even when we take the path $s_0 \rightarrow s_6$, we will eventually reach s_0 or s_1 and hence we can again satisfy the liveness of component**P**. Symmetrically we can reason about component **Q** as well.



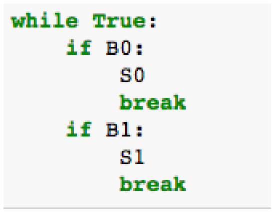
Implementation Details

In this section I will discuss various details about my implementation and then mention the test cases along with the outputs.

GCL to Python3

There are two main constructs in GCL, the ”Alternate Construct” and the ”Repetitive Construct”.

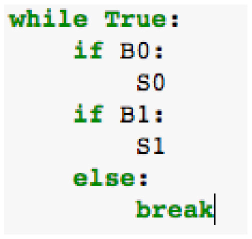
1. **Alternate Construct**: **if B0 → S0 ‖ B1 → S1 fi**



```
while True:
    if B0:
        S0
        break
    if B1:
        S1
        break
```

Figure 1: Python 3 implementation of Alternate Construct

2. **Repetitive Construct do B0 → S0 ‖ B1 → S1 od**



```
while True:
    if B0:
        S0
    if B1:
        S1
    else:
        break
```

Figure 2: Python 3 implementation of Repetitive Construct

Implementation

The base class called `Component` implements an abstract class with methods that the all the subclasses must implement. The file `petersons.py` contains it. In each of the test classes, a component is defined as an instance of a sub class of `Component`. The method `peterson` implemented by each component is run on a separate thread.
Two test cases we investigated.

1. Test Case 1, implemented in `test1.py`. Here each component is trying to modify a shared variable, by re-assigning a new value to it.
2. Test Case 2, implemented in `test2.py`. Here each component is trying to mutate a shared python `list`, by adding and removing elements from it.

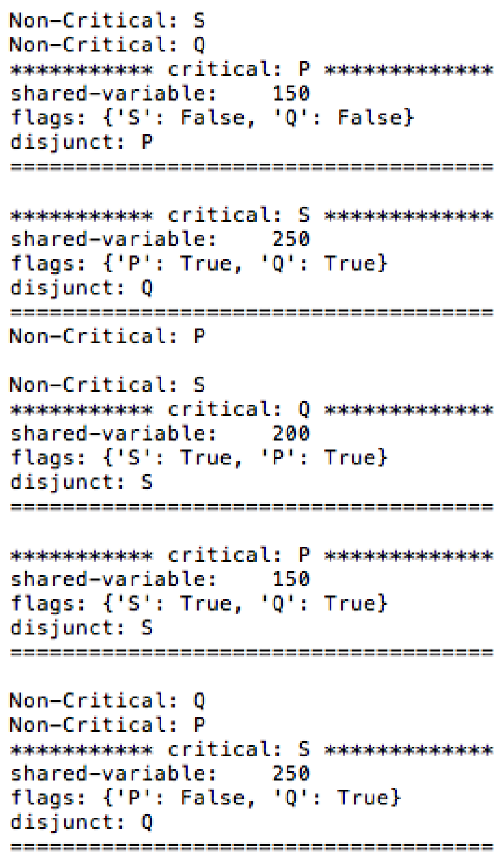
Further details about the implementation are provided as in-line comments in the scripts.

Run

No external libraries, so any machine with `Python 3` installed should be able to run the scripts.
To run `test1.py`: `python3 test1.py`
To run `test2.py`: `python3 test2.py`
Sample outputs are provided in the “outputs” folder.

Test Case 1: Change Variable

Three components called `s`, `p` and `q` are defined. Each tries to re-asign a new value to a shared variable.
You can see that when any component is in the critical section, either the flags of the other components are `False` or the `disjunct` is not equal to the component in the critical section. This shows that both guards in the guarded skip, *alternate construct* are held. The `disjunct` is the variable *v* in the psuedo-code.

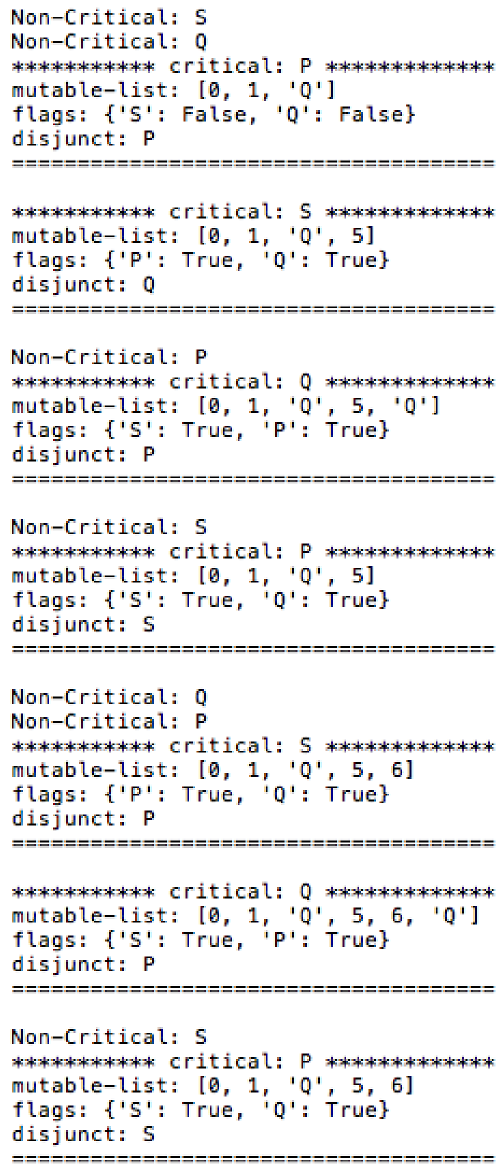


```
Non-Critical: S
Non-Critical: Q
*********** critical: P *************
shared-variable:    150
flags: {'S': False, 'Q': False}
disjunct: P
=====================================

*********** critical: S *************
shared-variable:    250
flags: {'P': True, 'Q': True}
disjunct: Q
=====================================
Non-Critical: P

Non-Critical: S
*********** critical: Q *************
shared-variable:    200
flags: {'S': True, 'P': True}
disjunct: S
=====================================

*********** critical: P *************
shared-variable:    150
flags: {'S': True, 'Q': True}
disjunct: S
=====================================

Non-Critical: Q
Non-Critical: P
*********** critical: S *************
shared-variable:    250
flags: {'P': False, 'Q': True}
disjunct: Q
=====================================
```

Figure 3: Sample Output for test1.py

Test Case 2: Mutate a list

Three components called `s`, `p` and `q` are defined. Each tries to mutate a shared list through appending or removing elements from the list. Similar to the previous test case it can be verified that the guarded skip conditions are satisfied within the critical section of each component.



```
Non-Critical: S
Non-Critical: Q
*********** critical: P *************
mutable-list: [0, 1, 'Q']
flags: {'S': False, 'Q': False}
disjunct: P
=====================================

*********** critical: S *************
mutable-list: [0, 1, 'Q', 5]
flags: {'P': True, 'Q': True}
disjunct: Q
=====================================

Non-Critical: P
*********** critical: Q *************
mutable-list: [0, 1, 'Q', 5, 'Q']
flags: {'S': True, 'P': True}
disjunct: P
=====================================

Non-Critical: S
*********** critical: P *************
mutable-list: [0, 1, 'Q', 5]
flags: {'S': True, 'Q': True}
disjunct: S
=====================================

Non-Critical: Q
Non-Critical: P
*********** critical: S *************
mutable-list: [0, 1, 'Q', 5, 6]
flags: {'P': True, 'Q': True}
disjunct: P
=====================================

*********** critical: Q *************
mutable-list: [0, 1, 'Q', 5, 6, 'Q']
flags: {'S': True, 'P': True}
disjunct: P
=====================================

Non-Critical: S
*********** critical: P *************
mutable-list: [0, 1, 'Q', 5, 6]
flags: {'S': True, 'Q': True}
disjunct: S
=====================================
```

Figure 4: Sample Output for test2.py

References

[1] WHJ Feijen and AJM van Gasteren, Monographs in Computer Science, On a method of multiprogramming.

[2] Michael Huth and Mark Ryan, Logic in Computer Science: Modelling and Reasoning about Systems.