# Software Verification – assignment 1: Peterson's algorithm

> If you are working at LIACS or remotely through SSH, the tools and libraries needed for this course have already been installed. To access them, type:
>
> ```
> $ source /vol/share/groups/liacs/scratch/softveri/init.sh
> ```
>
> Your shell prompt will now display `(softveri)` to show the tools are accessible.

In this lab assignment, you'll model Peterson's algorithm for N processes (also known as the filter algorithm). The task is to implement the transition function (or "next-states" function) of this algorithm, so that its behavior can be explored by a model checker.

Assignments will be based on a model checking framework. This framework is written in Python, and supports both models written in Python as well as models written in C (using the PINS interface). More information can be found in the `doc` directory.

A good start is to try playing around with the models interactively:

```
$ python
Python 3.4.3 (default, Nov 12 2018, 22:20:49)
[GCC 4.8.4] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import minipor;
>>> mdl = minipor.Model();
>>> mdl.initialState
{'pcA': 0, 'pcB': 0, 'a': 0, 'x': 0, 'y': 0}
>>> mdl.initialState.labels
{StateLabel('pcB==0'), StateLabel('pcA==0')}
```

In Python, you can get "man page"-style help about objects or classes by using the `help` command. For instance, try `help(minipor.Model)` or `help(mdl)`.

In the framework, models are represented by subclasses of the class `Model`. The `nextStates` function of a model should return the successor states to a given state `src`. This function is implemented as a generator.

> A generator is a function that generates a sequence of values, returning one value per call. They can be used in `for ... in` loops. Instead of `return value`, you use `yield value`. When the caller comes back to the generator for the next value, the function continues after the last `yield`. The sequence of values ends when the generator function returns.

A very simple generator looks like this:

```
def onetwothree():
    i = 1;
    while i <= 3:
        yield i;
        i += 1;

for v in onetwothree():
    print(v);
```

Notice how the value of `i` is preserved in between calls of `onetwothree()`.

The function `nextStates(src)` should give all transitions from the state `src`. A transition is represented as a tuple consisting of the state, and the action used to get there.

To see it in action for the `minipor` example:

```
>>> for state, action in mdl.nextStates(mdl.initialState):
...     print(state, action);
...
```

In the constructor of the model, all possible actions of the model have been created in `actions`, which maps names to `Model.Action` objects. Similarly, state labels have been created in `labels`, which contains `Model.StateLabel` objects. These objects can contain additional information about the actions and state labels that will be useful later.

To return a given `Action`, which you'll need to do in the `nextStates` implementation, simply return `self.actions[name]` with that action's name.

# 1 Peterson's algorithm

There are N processes, running simultaneously. Each process has a "critical section" , and only one process may be in this section at the same time. Peterson's algorithm is a way to enforce this mutual exclusion property.

The complete algorithm is given in Algorithm 1. In the version given here, the content of the critical section is simply a no-op; in a real program, this is where the process would do something useful, such as accessing a shared resource.

In each process i, the following code is executed. Loop ranges do not include the upper bound.

```
// try to enter critical section
for level[i] from 0 to (N − 1):
    last[level[i]] := i;

    for k from 0 to N:
        if k = i: continue;
        await last[level[i]] ≠ i or level[k] < level[i];

// critical section
nop;

// leave critical section
level[i] := 0; goto 0;
```

Algorithm 1: Filter algorithm for N processes.

### 1.1 Task 1

**Draw the deterministic program graph** of a single Peterson process as shown in Algorithm 1. In your drawing, include the following:

- For each transition, indicate the action(s) that occur.
- For each transition, indicate what conditions (if any) need to be true for the transition to be taken.
- Indicate which state(s) make up the critical section.

**Implement Peterson's algorithm.** The file `peterson.py` is a template for the model. The state vector (the class `State`) is already worked out. To implement the algorithm, modify the `nextStates` function in `peterson.py`. Running `python peterson.py` will run a simple reachability analysis.

### 1.2 Task 2

With Peterson's algorithm implemented, you'll want to check if the model does indeed guarantee mutual exclusion. If you look at the `main` function of `peterson.py`, you'll see that right now it does nothing more than go through all reachable states (given by `mdl.reach()`).

**Add a check to verify that there is never more than one process in the critical section.** Each state has a set of state labels (`labels`) that show which process(es) are in the critical section. Use this to verify that mutual exclusion is implemented by the model.

### 1.3 Task 3

In addition to verifying mutual exclusion, you might want to check the model for invalid modeling artifacts such as deadlocks. How do we do this?

A deadlock is a state $s$ that has no successors. So, to detect all deadlocks, we can look at the `nextStates` of every reachable state. If it returns nothing, that state is a deadlock.

We've seen that generators (like `nextStates`) can be used in a `for ... in` loop. But they can also be stepped through "manually" by using the `next()` function. You can try it out interactively:

```
>>> it = mdl.nextStates(mdl.initialState);
>>> # 'it' is now a reference to the generator object:
>>> it
<generator object nextStates at 0xbeef0000>
>>> # every time the 'next' function is called, the next value is returned
>>> next(it)
```

If there are no more states, the `next` function will raise a `StopIteration` exception. So to see if a state has no successors, we can check whether we catch that exception on the very first call to `next`.

Use this to **add a counter for deadlocks** to the main function.

## 2 Results

Write a short report answering the above questions and make a compressed archive of the report together with your source code.

- The archive must be in `.zip` or `.tar.gz` format.
- Be sure to include your name and student number in the report.
- Please run `make clean` before making the archive.