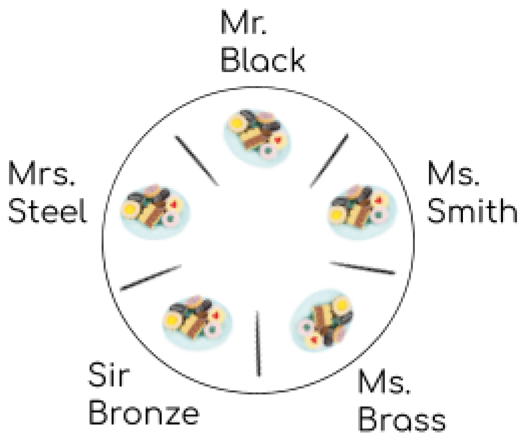
Chopstick Lab

Part 1

Five hungry blacksmiths have gathered for their evening meal. Help these hungry workers eat so they can get back to forging some alloys!

Their dining situation this week is a little unusual, however. Our five blacksmiths are seated at a round table, and there is a single chopstick placed between each adjacent pair of blacksmiths. In order to take a bite of their meal, each blacksmith must pick up both the chopstick on their left and the chopstick on their right. After eating for some amount of time, the blacksmith can put their chopsticks down for others to use.



The default implementation of the “Dining Blacksmiths” problem leads to livelock. If everyone picks up a left chopstick, then no one can pick up their right chopstick to enjoy their delicious souffle. Read over the following implementation of this naive algorithm and confirm that it livelocks.

Implementation: rude_smiths.pml

The array chopState stores the state of each chopstick: either available or taken. The array smithState stores the state of each blacksmith: whether they are holding no chopsticks, their left chopstick, their right chopstick, or both.

You can find the descriptions for LeftOk, RightOk, DropOk, TakeLeft, TakeRight, and DropChops in the instructions in the code, and below.

1. LeftOk, RightOk: Checks that the blacksmith can take their left/right chopstick, meaning the right chopstick is available and this blacksmith is not already holding it.
2. DropOk: Checks that the blacksmith can drop their holding chopsticks, this can only occur when they have both left and right chopstick in hand.
3. TakeLeft, TakeRight: Models the procedure of a random and valid blacksmith—a blacksmith that passed LeftOk/RightOk-—taking their left/right chopstick.
4. DropOk: Models the procedure of a random and valid blacksmith—a blacksmith that passed DropOk—dropping both of their chopsticks.

“Fed”-lock

Verify that leaving these blacksmiths alone to pick up utensils leads to livelock and starving blacksmiths. You will resolve this livelock in part 2.

Running from the Command Line

To verify that the provided program livelocks, use the following commands, in order:

- `spin -a <file>.pml` : generates the verifier C program as pan.c
- `gcc -o pan pan.c` : compiles the verifier C program (with many options available!)
- `./pan -a -m2000000`: checks for deadlocks and violated liveness properties over all simulations. `-m2000000` sets the maximum depth to `2000000` steps.

In the output, the phrase “acceptance cycle” indicates that the non_starvation liveness property has been violated.

Part 2

Next up, you’ll implement a solution to this problem with the non_starvation property. This property removes the possibility of livelock, ensuring that every blacksmith will never be starved (prevented from eating for an infinitely long period of time). The solution is as follows:

1. Every chopstick is always being held by some blacksmith. That is, chopsticks no longer lie on the table at any point. In addition, every chopstick is always either clean or dirty.
2. All blacksmiths start out holding their right chopstick, except blacksmith 0 and blacksmith (`NUM_SMITH - 1`). Blacksmith 0 has both of their chopsticks and blacksmith (`number of` (`NUM_SMITH - 1`) has neither. All chopsticks start out dirty.
3. Rather than just take an available chopstick, a blacksmith must request the utensil that they want from its current owner, adding their request to a list of pending requests.
4. The holder of a requested utensil should hand it to the requester if it is dirty. Before handing over the chopstick, its owner should clean it.
5. When a blacksmith has two clean chopsticks, they may eat. This action dirties both chopsticks.

Your task is to implement the algorithm described above, using the following code stencil:

Stencil: polite_smiths.pml

The array 'chopState' should store the state of each chopstick: either clean or dirty. The array 'smithState' should store the state of each blacksmith: whether they are holding no chopsticks, their left chopstick, their right chopstick, or both. Lastly, the array 'requests' should store whether each chopstick is requested or not.

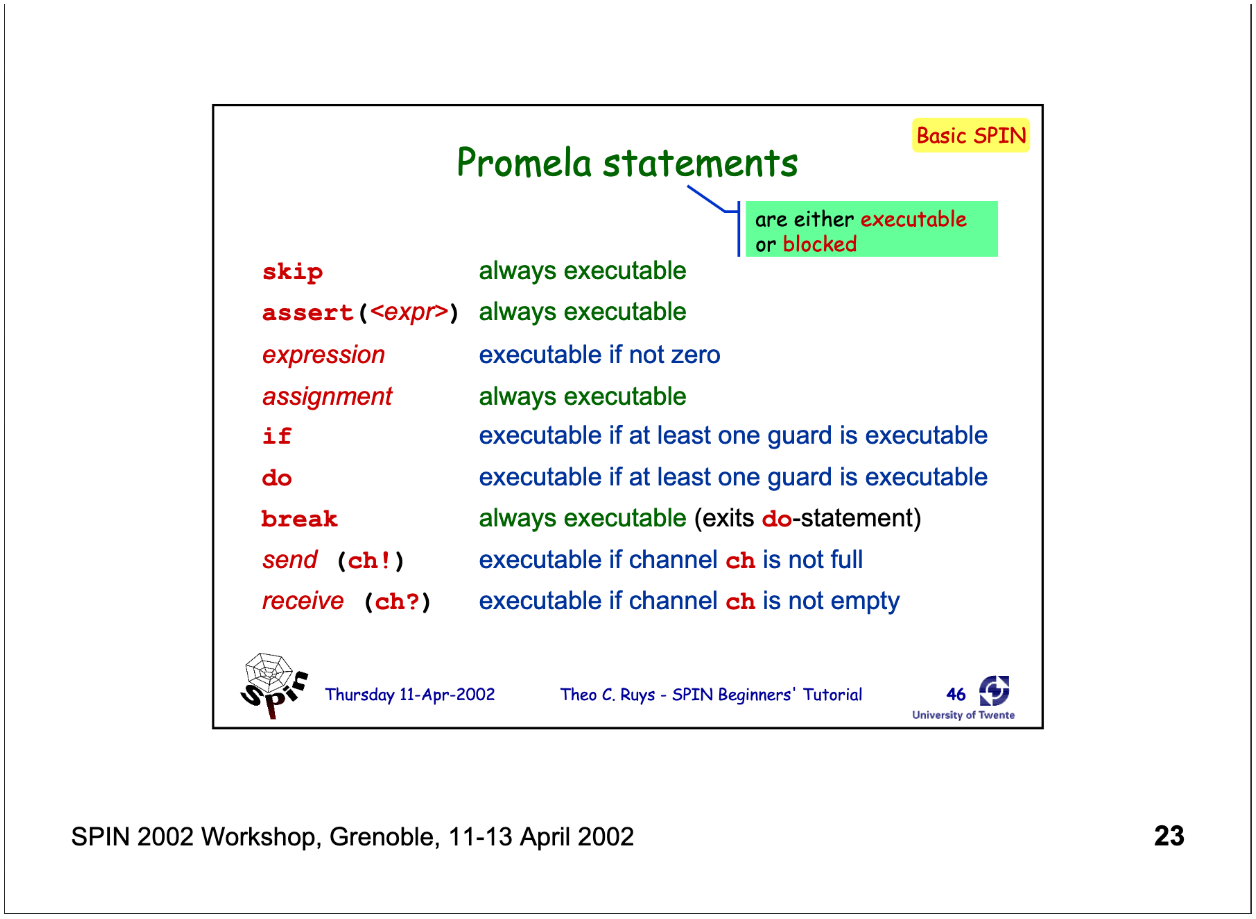
Implement LeftOk, RightOk, DirtyOk, RequestOk, RequstLeft, RequestRight, DirtyChops, and HonorRequest based on the instructions in the stencil and below. Additionally, create a non_starvation property to check that no blacksmiths are going hungry.

1. LeftOk, RightOk: Check whether a blacksmith does not already have their left/right chopstick in hand, and the left/right chopstick is not already requested.
2. DirtyOk: Check whether a blacksmith is holding both chopsticks and both the chopsticks are clean.
3. RequestOk: Check whether a chopstick is currently dirty and is requested.
4. RequestLeft, RequestRight: Model the left/right chopstick of a valid blacksmith getting requested.
5. DirtyChops: Model the blacksmith using chopsticks in both hands to eat making the chopsticks dirty.
6. HonorRequest: Model cleaning and transferring a 'requested' 'dirty' chopstick from one blacksmith to another. Then, clean the chopstick and remove the request for it.

Spin tip:
Be careful with statements that can block! For example, "if statements" with no executable guards will block, halting the current process.
For reference:

- **do** http://spinroot.com/spin/Man/do.html
- **if** http://spinroot.com/spin/Man/if.html

- **ltl logical operators** http://spinroot.com/spin/Man/ltl.html



Basic SPIN

Promela statements

are either executable or blocked

`skip`	always executable
`assert(`*<expr>*`)`	always executable
expression	executable if not zero
assignment	always executable
`if`	executable if at least one guard is executable
`do`	executable if at least one guard is executable
`break`	always executable (exits `do`-statement)
send `(ch!)`	executable if channel `ch` is not full
receive `(ch?)`	executable if channel `ch` is not empty

Thursday 11-Apr-2002 Theo C. Ruys - SPIN Beginners' Tutorial 46 University of Twente

SPIN 2002 Workshop, Grenoble, 11-13 April 2002 **23**

Verify!

Run this model with the non_starvation property to make sure that every blacksmith gets to enjoy their meal! Have a TA check you off when you've reached this point.

To verify that your algorithm is correct (or not) use the following commands, in order:

- `spin -a <file>.pml` : generates the verifier C program as pan.c
- `gcc -o pan pan.c` : compiles the verifier C program (with many options available!)
- `./pan -a -m2000000`: checks for deadlocks and violated liveness properties over all simulations. `-m2000000` sets the maximum depth to `2000000` steps.

If your program is correct, you should see "errors: 0" in the output. If you instead are getting acceptance cycles, use the following commands to debug:

- `spin -t -p <file>.pml` : produces a counterexample trace
- `spin -t <file>.pml` : traces the trail file but does not print line by line output. This is useful for debugging with printf statements.