Mini Project: Dining Philosophers

Peter Palfrader

July 30, 2010

1 Introduction

The Dining Philosophers scenario is one of the classical examples of computer science to illustrate certain aspects of concurrent systems.

The description of the setup varies slightly from source to source, but they all are similar to the following scenario:

A certain number of philosophers, often five, sit around a table. A philosopher's only purpose in life is to think, yet in order to sustain their thinking body they occasionally have to take some time off to eat. For this reason the table has in its center a large bowl of pasta. Unfortunately, owning to the tough budget situation of their university department, there are only as many forks as there are philosophers. However, since they are not of Italian descent, none of them has mastered the art of eating their spaghetti with only one fork; each philosopher requires exactly two forks to eat.

They agree to place each fork between two philosophers so that every fork is shared between these two. If a philosophers gets hungry he tries to acquire, one at a time, the forks that are on the left and on the right of him. Which side he starts on is non-deterministic. Once equipped with two utensils they will eat for some time and then replace their forks and return to thinking.

The behavior of one such thinker is shown in figure 1, the entire company is sketched it figure 2.

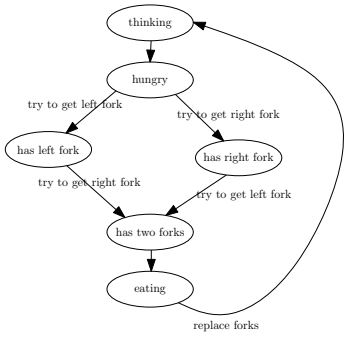

Figure 1: One philosopher's behavior.

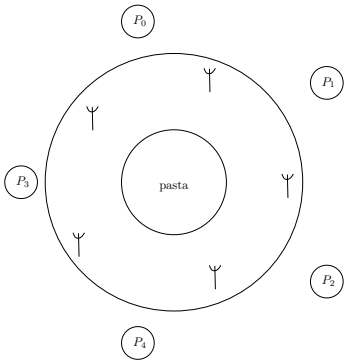

Figure 2: Philosophers sitting around food, thinking.

2 Modeling in Promela

In order to examine the setup with SPIN we have to model it in Promela.

We only show an abbreviated version of the philosopher proc here. The complete Promela spec for the system can be found in the accompanying source directory in the file called `din.prm`.

```
bool pthinking[NUM_PHIL], phungry[NUM_PHIL], peating[NUM_PHIL] = false;
int forks[NUM_PHIL] = -1;

proctype P(int i) {
        int right = i; int left = (i + 1) % NUM_PHIL;
Think:  atomic { peating[i] = false; pthinking[i] = true; };
        /* Nothing */
Hungry: atomic { pthinking[i] = false; phungry[i] = true; };
        if :: skip ;
           atomic { forks[left]  == -1 -> forks[left]  = i };
           atomic { forks[right] == -1 -> forks[right] = i };
          :: skip ;
           atomic { forks[right] == -1 -> forks[right] = i };
           atomic { forks[left]  == -1 -> forks[left]  = i };
        fi;
Eating: atomic { phungry[i] = false; peating[i] = true; };
Done:   forks[right] = -1; forks[left] = -1;
        goto Think;
}
```

First we declare three arrays of booleans that will record in which state each philosopher is. These state variables are not used in the program itself but instead are used later on when we try to verify properties, given in LTL, of the system.

We also declare an array of n forks. These too could be booleans but we chose to make them integers so we can record *who* holds a fork instead of just noting that it's taken. This will make reading traces slightly more comfortable.

When a process gets initiated it gets told which philosopher it is (0 through $n-1$). The philosopher i uses the forks i and $i+1$ (modulo n).

The first state a philosopher, let's call him Phil, enters is the thinking state in which he does nothing that is visible to external observers. Presumably he thinks a lot.

Once Phil get hungry he picks non-deterministically which fork to try to acquire first. This is modeled by the select/alternative statement with two guards, both of which are enabled.

When the philosopher has control over both forks he can eat and once done returns the forks and resumes thinking.

The main procedure, not shown above, launches a number of these philosophers.

3 Deadlocks

The first property we want to ascertain is if the system is free of deadlocks.

```
$ ./1-check-dl
+ spin -a -DSTRATEGY=0 din.prm
+ gcc -O2 -g -o pan pan.c -DSAFETY
+ ./pan -m1000000
pan:1: invalid end state (at depth 2117)
pan: wrote din.prm.trail
[..]
```

SPIN tells us that the system actually is *not* free of deadlocks. (`invalid end state` means first that the system actually came to a halt at some point, and that the states the various processes hold are not explicitly marked as valid end states.)

Fortunately the tool also provided a trail that let's us investigate what went wrong exactly.

```
+ spin -t din.prm
[..]
                          Philosopher 4 is thinking
                          Philosopher 4 is hungry
                          Philosopher 4 is trying to pick up l fork 0
                          Philosopher 4 is trying to pick up r fork 4
                     Philosopher 3 is trying to pick up r fork 3
               Philosopher 1 is trying to pick up r fork 1
spin: trail ends after 2118 steps
#processes: 6
                 forks[0] = 4
                 forks[1] = 0
                 forks[2] = 1
                 forks[3] = 2
                 forks[4] = 3
[.. pthinking are all 0, phungry all 1 and peating all 0]
```

Either by looking at the end state, or by following the trail step-by-step it becomes apparent what happened. All the great thinkers got hungry at more or less the same time, and each of them equipped a fork. Fork #0 is held by philosopher #4, fork #1 by philosopher #0, fork #2 by thinker #1 etc.

Each philosopher holds one fork and is unable to get a second one to finally eat and return the forks to the table for somebody else to eat. Their lack of communications or agreement on a means to avoid just such a situation has doomed them all to starvation.

4 A smarter philosopher

One way to solve such a deadlock problem is to ensure that all processes acquire their respective resources in the same order. For our philosophers that means that they number all the forks from 0 to $n-1$ and always try to get the fork with the lower number first.

If our model of dining philosophers in `din.prm` is built with `-DSTRATEGY=1` then they will try this new approach. This is the part where they pick up their forks:

```
if :: left < right;
     atomic { forks[left]  == -1 -> forks[left]  = i };
     atomic { forks[right] == -1 -> forks[right] = i };
   :: right < left;
     atomic { forks[right] == -1 -> forks[right] = i };
     atomic { forks[left]  == -1 -> forks[left]  = i };
fi;
```

Note how the guards which previously were always enabled (`skip;`) are now conditional statements of which exactly one holds and is therefore enabled. The order in which a specific philosopher picks up their fork is now deterministic.

Let's check this new system for deadlocks:

```
$ ./2-check-dl-different-strategy
+ rm -f pan.b pan.c pan.h pan.m pan.t pan din.prm.trail din-coord.prm.trail buechi
+ spin -a -DSTRATEGY=1 din.prm
+ gcc -O2 -g -o pan pan.c -DSAFETY
+ ./pan -m1000000
[.. nothing about invalid end states and no .trail file.]
```

So apparently this new strategy is better; there are no more deadlocks. Clearly this must mean that now every philosopher can have some food whenever they get hungry. Let's verify this statement (it'll turn out to be wrong).

SPIN allows us to check so-called Never-claims. A Never claim is a property that we claim does not hold. Should SPIN find a counter example showing that it does in fact hold we can infer that the negation of our claim does indeed hold.

The property we want to deduce eventually is that *whenever philosopher #0 gets hungry, he will eat eventually.* In LTL that's `[](phungry -> <>peating)` where `phungry` is the property indicating that Phil is hungry and `peating` is true when he is eating. The negated never claim would then be `! [] (phungry -> <>peating)`.

```
$ ./3-check-starvation
+ spin -f '!([](phungry -> <>peating))'
+ spin -a -DSTRATEGY=1 -N buechi din.prm
+ gcc -O2 -g -o pan pan.c
+ ./pan -a -n -m1000000
pan:1: acceptance cycle (at depth 9586)
pan: wrote din.prm.trail
[..]
```

So SPIN tells us that our never claim does not hold (there is an acceptance cycle).

```
$ spin -p -t din.prm
[..]
              Philosopher 0 is hungry
                              Philosopher 4 is trying to pick up l fork 0
              Philosopher 0 is trying to pick up r fork 0
  <<<<<START OF CYCLE>>>>>
                              Philosopher 4 is trying to pick up r fork 4
                              Philosopher 4 is eating
                              Philosopher 4 is done eating
                              Philosopher 4 is thinking
                              Philosopher 4 is hungry
                              Philosopher 4 is trying to pick up l fork 0
spin: trail ends after 9606 steps
```

Here we can see what can happen. Even tho philosopher #0 is already quite hungry his neighbor, philosopher #4, gets the shared fork first, picks up the other fork and starts eating. When done and thinking again he immediately becomes hungry once more and again claims the fork before Phil can pick it up. Repeat until Phil has starved to death.

5 Central Coordinator

The previous system we tried was free of deadlocks but still did not prevent starvation. One approach to fix this is a new process that coordinates when a process may acquire certain resources, that is processes have to request their resources from the coordinator that will queue these requests and grant them if not immediately then at least eventually.

We have modeled a coordinator for our philosophers. Whenever a philosopher gets hungry they send a request with their number to the coordinator. The request channel is long enough to hold at least as many requests as there are philosophers.

Once the request has been sent a philosopher waits until it has been granted before they pick up both forks. Once done eating a philosopher signals that to the coordinator.

The coordinator in our model is kept deliberately simple. Only one request is granted at any one time, i.e. only one philosopher can eat at any point in time. This is achieved by the coordinator continuously doing the following: read a request from the request channel, grant that request to the philosopher in question and then wait until that philosopher is done eating.

In Promela this looks thus (again, only an abbreviated version is shown here; the complete source code can be found in `din-coord.prm`):

```
proctype Coordinator() {
        int who, who2;
start:   request?who;      // got a request from $who
         comm[who]!who;    // granted request to $who
         comm[who]?who2;   // $who2 is done eating
         assert(who == who2);  // basic correctness assertion.
```

```
        goto start;
}

proctype P(int i) {
        int right = i; int left = (i + 1) % NUM_PHIL; int ack;
Think:  atomic { pthinking[i] = true; peating[i] = false; };
Hungry: request!i;  // sends a request to the coordinator
        atomic { phungry[i] = true; pthinking[i] = false; };
        comm[i]?ack; // wait until the request is granted.
        if :: skip ;
           atomic { forks[left]  == -1 -> forks[left]  = i };
           atomic { forks[right] == -1 -> forks[right] = i };
          :: skip ;
           atomic { forks[right] == -1 -> forks[right] = i };
           atomic { forks[left]  == -1 -> forks[left]  = i };
        fi;
Eating: assert ( forks[left] == i && forks[left] == i );
        atomic { peating[i] = true;  phungry[i] = false; };
Done:   forks[right] = -1; forks[left] = -1;
        comm[i]!ack; // return token
        goto Think;
}
```

First we again check for deadlocks:

```
$ ./6-coord-check-dl
+ spin -a -DSTRATEGY=0 din-coord.prm
+ gcc -O2 -g -o pan pan.c -DSAFETY
+ ./pan -m1000000 -w21
[.. nothing about invalid end states and no .trail file.]
```

So that's a good first step. Now check if Phil can still starve like in the previous setup:

```
$ ./7-coord-check-starvation
+ spin -f '!([](phungry -> <>peating))'
+ spin -a -N buechi din-coord.prm
+ gcc -O2 -g -o pan pan.c
+ ./pan -a -n -m1000000 -w24
[.. and no .trail file]
```

So we finally found a way to keep our thinking elite fed.

6 Summary

We modeled the original dining philosophers scenario and showed that the system is not free of deadlocks. We modified the setup slightly and this second version was without deadlocks but still did not ensure that each philosopher could eat after becoming hungry.

Lastly we presented one means to ensure that each philosopher can always satisfy their hunger. While our coordinatior in this last system was quite inefficient we did show the system to be free of deadlocks and starvation.