Modeling Concurrent Activities and Resource Sharing in Modelica

Håkan Lundvall, Peter Fritzson
Dept. of Computer and Information Science,
Linköping University, SE-581 83 Linköping, Sweden
(haklu,petfr)@ida.liu.se

Abstract
Modelica [1,2,4,6] is an equation based, object oriented modeling language for modeling of complex multi-domain systems. Modelica is an inherently concurrent language in the sense that objects with behavior specified by equations evolve in parallel, concurrently in time. Equations specify how objects evolve and interact, as well as giving constraints for the parallel behavior. We can view each object as a kind of parallel process. As always in parallel activities, concurrent access of shared common resources can lead to problems. In this paper we will, as an example of a well-known concurrence problem, present a Modelica implementation of the Dining Philosophers problem [7]. The presented solution guarantees freedom from both deadlock and starvation. To accomplish this a general mutex class is introduced.

Keywords: Modelica, Resource Sharing, Concurrency

Introduction
Modelica is an object oriented modeling language capable of describing heterogeneous physical systems through the use of hybrid differential algebraic equations. One of the great advantages of the Modelica language is the possibility to model systems from different physical disciplines in the same model. It is, for example, possible to model the control software in the same model as the system it controls. Often, in this kind of systems, there is a need to model several concurrent processes that share a limited amount of mutual resources.

Related work in the discrete systems area includes a Modelica library for modeling Petri Nets [5].

The Dining Philosophers
The dining philosophers problem [7] is considered a classic synchronization problem. This is not because of its practical importance – philosophers are not very common in real life. The reason this problem is interesting is that it represents a large class of concurrency problems. It is a simple example of allocation of resources among several concurrent processes. This needs to be done in a deadlock and starvation free manner. Let us now describe our dining philosophers; there are five dining philosophers who are sitting around a table thinking and eating during their whole life. Five plates and five forks are available on the table. A philosopher is either thinking or eating, and can only eat if he has two forks in his hands. Now and then a philosopher gets hungry and tries to pick up the two forks closest to him. If one of the forks is already in the hand of neighbor, he cannot take it. When a hungry philosopher has both forks in his hands he eats without releasing the forks. When he has finished eating, he puts down both forks and starts thinking again. The crucial observation is that the philosopher can only eat if he has access to two neighboring forks out of the common resource of five forks.

There are two potential problems associated with the dining philosophers:

- Deadlock – philosophers wait indefinitely on each other without ever been able to grab two forks.
- Starvation – certain philosophers are never able to eat because their neighbors always grab the forks ahead of them.

Solution
If all philosophers acted entirely autonomous, it is easy to see how deadlock could occur. Let us assume that when a philosopher gets hungry, he always picks up the fork to the right if it is available and then the other one. If all philosophers becomes hungry simultaneously they all pick up the fork to the right, but no one can eat since they all wait for their left fork. In order to avoid this situation a mutual exclusion (mutex) mechanism is introduced so that no one can pick up a fork without first getting hold of the mutex.
**First attempt:**
When a philosopher gets hungry, he waits until both forks are available and then he requests the mutex. If, when he receives the mutex, the forks are still available he grabs both forks and start eating. If some other philosopher got the mutex first one of the forks may not be available any longer, in which case the hungry philosopher starts over and waits for both forks to be free again.

When he is done eating he puts both forks back on the table.

It is easy to convince one self that deadlock can no longer occur, since if someone is hungry but cannot eat his neighbor must be eating (no one has only one fork, and if he has two he is eating) and he will eventually stop eating. But nothing is stopping four of the philosophers to starve the fifth to death. To make sure that cannot happen the algorithm must be changed a little.

**Second attempt:**
We give the philosophers the opportunity to tip their neighbors off when they put down the forks. When a philosopher is done eating, he first puts down his left fork and gives his left neighbor a chance to pick it up if he wants it. Not until the left neighbor has acknowledged the tip, does he put down the right fork. In case he immediately gets hungry again he also waits for an acknowledgement from his right neighbor, which he has tipped off in the same way, before he tries to pick the forks up himself again.

The second alteration to our first algorithm is that a philosopher no longer must pick both forks up simultaneously. Instead, when a philosopher is hungry he always picks his right fork up if it is available.

**Proof:**
If there is deadlock all philosophers must be hungry and holding on to their right fork, since no one ever holds just the left fork, and if some fork is still on the table or if someone has two forks there would not be deadlock, since the fork on the table could be picked up and the one having two forks would eventually be done eating.

Let us assume that deadlock has occurred. The last thing that happened before deadlock occurred must have been that one philosopher picked up his right fork, so the state of the dining table just before the deadlock must have been one fork on the table and the others in a philosophers right hand. There are only three ways in which this state can occur:

1) The philosopher to the right of the fork just put it down, in which case he is no longer hungry and he will soon put his other fork down as well, hence there will not be deadlock.

2) The philosopher to the left of the fork just put it down, but became hungry again immediately. However, according to the algorithm he must first offer the fork to his neighbor, who in this case would grab it and eat, since he is hungry. Thus, there would be no deadlock in this case either.

3) Some other fork was just picked up. In this case the philosopher who pick the fork up must have just become hungry and so must everyone because if there is at least one philosopher that is not hungry and at least one that is, then at least one can eat. When they all get hungry simultaneously, one of them gets the mutex first and grabs both forks, which does not lead to the sought state.

Now let us look at starvation. Assume that a philosopher is hungry, either his right neighbor is eating in which case the neighbor will soon offer the hungry philosopher his fork or the fork is already available. Now the hungry philosopher has his right fork. If the left fork is busy then his left neighbor is either eating or waiting for his left fork. If he is eating then our hungry philosopher will be offered the fork when the neighbor is done. If the neighbor in his turn is waiting, then somewhere around the table there must be an eating philosopher that will loose up the chain, otherwise all philosophers would be waiting for there left fork in a deadlock, which we just proved could not happen.

**Implementation**
In this section the implementation of the Modelica model is presented. Figure 1 shows the connection structure of a Modelica model with five philosophers, five forks, and a Mutex instance providing mutually exclusive access to two forks needed for a philosopher to eat.

Each philosopher in the DiningTable model is connected to the two forks on his left and right sides. He can only pick up those forks if his left and right neighbor philosophers do not occupy them. In the center there is a shared mutex that all the philosophers are connected to.
This connection is used to signal if a philosopher wants to get access to the forks in order to eat, so that no other philosopher can pick up the fork between the check for availability and the actual picking up.

A philosopher is connected to its left and right neighboring forks via ForkPhilosopher-Connection ports. This connector transfers information about the state of the fork. This includes if the fork is held by the connected philosopher, if it is busy or if it is available. It also contains flags that the philosophers use to signal their neighbors when they put down the forks.

The Fork class transfers the information between the left and right ports.
thinking, the left fork is put down and the neighbor is told of the fact that it is. Not until the neighbor acknowledges the tip is the right fork put down and that neighbor gets the tip. Note that the timeToGetHungry event does not fire until both neighbors have acknowledged.

```plaintext
model Philosopher "A Philosopher, connected to forks and a mutex"
import Random;
MutexPortOut mutexPort "Connection to the global mutex";
parameter Random.Seed startSeed = {1,2,3};
parameter Real mu = 20.0 "mean value";
parameter Real sigma = 5 "standard dev";
discrete Integer state "1==thinking, 2==hungry, 3==eating";
ForkPhilosopherConnection left;
ForkPhilosopherConnection right;
protected
constant Integer thinking=0;
constant Integer hungry=1;
constant Integer eating=2;
discrete Real T;
discrete Real timeOfNextChange;
discrete Random.Seed randomSeed;
Boolean canEat;
Boolean timeOfNextChange;
Boolean doneEating;
equation
  timeToChangeState = timeOfNextChange <= time;
  canEat = (state == hungry) and not (left.busy or right.busy);
  timeToGetHungry = (state == thinking) and timeToChangeState and not
                      (left.flagOut or right.flagOut);
  doneEating = (state == eating) and timeToChangeState;
algorithm
  when initial() then
    state := thinking;
    left.pickedUp := false;
    right.pickedUp := false;
    (T,randomSeed) := Random.normalvariate(mu, sigma, startSeed);
    timeOfNextChange := abs(T);
  end when;
  when pre(timeToGetHungry) then
    state := hungry;
  end when;
  // Request mutex to be able to grab the forks
  when pre(right.flagIn) then
    mutexPort.release := false;
    mutexPort.request := true;
  end when;
  when pre(left.flagIn) then
    mutexPort.release := false;
    mutexPort.request := true;
  end when;
  when pre(canEat) then
    mutexPort.release := false;
    mutexPort.request := true;
  end when;
when state == hungry and
  not pre(right.busy) then
  mutexPort.release := false;
  mutexPort.request := true;
end when;
// Got the mutex
when pre(mutexPort.ok) then
  if not pre(right.busy) then
    right.pickedUp := true;
  end if;
  // If the forks are still available
  // then grab them and decide for how long to eat
  if pre(canEat) then
    left.pickedUp := true;
    right.pickedUp := true;
    (T,randomSeed) := Random.normalvariate(mu, sigma, pre(randomSeed));
    timeOfNextChange := time + abs(T);
  state := eating;
  end if;
  // Release the mutex
  mutexPort.release := true;
  mutexPort.request := false;
  // Acknowledge flags from neighbors
  if pre(right.flagIn) then
    right.acknowledgeOut := true;
  end if;
  if pre(left.flagIn) then
    left.acknowledgeOut := true;
  end if;
end when;
// When done eating lay down the forks and
// set a new time to get hungry
when pre(doneEating) then
  state := thinking;
  left.flagOut := true;
  left.pickedUp := false;
  (T,randomSeed) := Random.normalvariate(mu, sigma, pre(randomSeed));
  timeOfNextChange := time + abs(T);
end when;
when pre(left.acknowledgeIn) then
  left.flagOut := false;
  right.flagOut := true;
  right.pickedUp := false;
end when;
when pre(right.acknowledgeIn) then
  right.flagOut := false;
end when;
end Philosopher;
```

// set then cancel the acknowledgements.
when not pre(right.flagIn) then
  right.acknowledgeOut := false;
end when;
when not pre(left.flagIn) then
  left.acknowledgeOut := false;
end when;
// Got the mutex
when pre(mutexPort.ok) then
  if not pre(right.busy) then
    right.pickedUp := true;
  end if;
  // If the forks are still available
  // then grab them and decide for how long to eat
  if pre(canEat) then
    left.pickedUp := true;
    right.pickedUp := true;
    (T,randomSeed) := Random.normalvariate(mu, sigma, pre(randomSeed));
    timeOfNextChange := time + abs(T);
  state := eating;
  end if;
  // Release the mutex
  mutexPort.release := true;
  mutexPort.request := false;
  // Acknowledge flags from neighbors
  if pre(right.flagIn) then
    right.acknowledgeOut := true;
  end if;
  if pre(left.flagIn) then
    left.acknowledgeOut := true;
  end if;
end when;
// When done eating lay down the forks and
// set a new time to get hungry
when pre(doneEating) then
  state := thinking;
  left.flagOut := true;
  left.pickedUp := false;
  (T,randomSeed) := Random.normalvariate(mu, sigma, pre(randomSeed));
  timeOfNextChange := time + abs(T);
end when;
when pre(left.acknowledgeIn) then
  left.flagOut := false;
  right.flagOut := true;
  right.pickedUp := false;
end when;
when pre(right.acknowledgeIn) then
  right.flagOut := false;
end when;
end Philosopher;
```
The Mutex class operates in the following way. Three of the local variables are always equal to the corresponding port variables through the equations in the Mutex model. The first when-statement is activated when one of the philosophers signals that he wants the mutex.

If the mutex is not occupied then it is reserved and the philosopher receives the ok signal, i.e. occupied=true, and the port[i].ok is set to true. If it is occupied the philosopher is set waiting. When the mutex is released and there are waiting philosophers then one of the waiting philosophers receives the ok signal and is removed from the waiting list. Finally, when the philosopher sets release[i] to false, the mutex is freed.

In Figure 2 below the result of simulating the dining table with random eating and thinking times are shown.

![Figure 2](image)

**Figure 2.** The result of simulating the dining table. Each philosopher alternates between Thinking and eating with the hungry state in the middle.

**Conclusion & Future work**

In this paper we have presented a solution to the dining philosophers problem with the use of a mutex class. We have shown that Modelica is powerful enough to model resource allocation in concurrent processes and that it can be done using familiar constructs such as the mutex. In the future one might continue to create Modelica implementations of other familiar constructs such as semaphores and monitors.

**References**


http://www.Modelica.org