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 [1,2,4,6,10] 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 guaranties freeness 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 [5,9]. Another commonly used language for describing concurrence and resource sharing is Esterel[8].

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 becomes hungry and tries to pick up the two forks closest to him. If one of the forks is already in the hand of a 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 autonomously, 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 subsequently 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 oneself 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 but 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 up both forks simultaneously. Instead, when a philosopher is hungry he always picks up his right fork if it is available.

Proof of freeness from deadlock and starvation:
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 picks up the fork must just have become hungry and so must everyone because if there is at least one philosopher who is not hungry and at least one who 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 turn is waiting, then somewhere around the table there must be an eating philosopher who will loose up the chain, otherwise all philosophers would be waiting for their left fork in a deadlock, which we just proved could not happen.

Implementation

In this section the representation of the dining philosophers problem and the mutex as a 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.

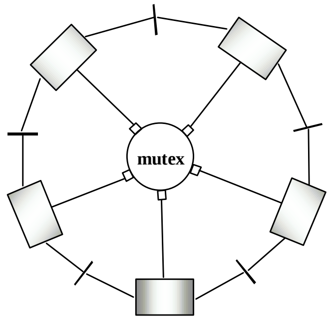

Figure 1. Connection structure of the dining philosophers model, alternating philosophers and forks, together with a central mutex.

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.

```
model DiningTable
  parameter Integer n = 5
   "Number of philosophers and forks";
  parameter Real sigma = 5
   "Standard deviation of delay times";

  // Give each a different random start seed
  Philosopher phil[n](startSeed=[1:n,1:n,1:n],
                      sigma=fill(sigma,n));
  Mutex       mutex(n=n);
  Fork        fork[n];
equation
  for i in 1:n loop
    connect(phil[i].mutexPort, mutex.port[i]);
    connect(phil[i].right, fork[i].left);
    connect(fork[i].right,
            phil[mod(i, n) + 1].left);
  end for;
end DiningTable;
```

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.

```
connector ForkPhilosopherConnection
  Boolean pickedUp(start=false);
  Boolean busy;
  Boolean flagIn(start=false);
  Boolean acknowledgeIn(start=false);
  Boolean flagOut;
  Boolean acknowledgeOut;
end ForkPhilosopherConnection;
```

The **Fork** class transfers the information between the left and right ports.

```
model Fork
  ForkPhilosopherConnection left
     "Connection to the philosopher to the " +
     "left of the fork";
  ForkPhilosopherConnection right
     "Connection to the philosopher to the"+
     "right of the fork";
equation
  // If one philosopher picks up the fork then
  // tell the other it is busy
  right.busy = left.pickedUp;
  left.busy  = right.pickedUp;
  left.flagOut = right.flagIn;
  left.flagIn = right.flagOut;
  left.acknowledgeOut = right.acknowledgeIn;
  left.acknowledgeIn = right.acknowledgeOut;
end Fork;
```

The **Philosopher** model implements the algorithm described earlier. The time intervals used to determine for how long each philosopher is thinking and eating comes from the `Random.normalvariate` function, which gives a normally distributed pseudo random number.
The first equations define some boolean variables that triggers events that occur within each philosopher. The `timeToChangeState` variable is set to true each time the simulated time exceed the randomly selected time of the next state change, causing the state to change from thinking to hungry or from eating to thinking. The variables `timeToGetHungry` and `doneEating` specializes the previously mentioned variable to the cases where the philosophers are in the tinking and eating states, respectively.
In the algorithm section all the events are handled. First, the built-in `initial()` event is used the set all the initial states and set the time for the first `timeToGetHungry`-event. The elsewhen part handles the `timeToGetHungry`-event. Note that pre() operator must be used around the timeToGetHungry-variable, otherwise there would be an algebraic loop since the code inside the when-clause affects the value of the variable that triggers the event.
If either neighbor signals the philosopher about a fork being laid down or there is an opportunity to eat or even just grab the right fork, then the philosopher asks for the mutex by setting the request flag on the mutex port.
When the philosopher receives the mutex the `mutexPort.ok` event fires the philosopher can carry out its intentions whether it was just to grab a free right fork or check if both forks are free and start eating or simply just acknowledge a tip from a neighbor.
When a philosopher is done eating the `doneEating`-event fires the state is changed to

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.

```
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              timeToChangeState;
  Boolean              timeToGetHungry;
  Boolean              doneEating;
equation
  timeToChangeState = timeOfNextChange <= time;
  canEat = (state == hungry) and
           not (left.busy or right.busy);
  timeToGetHungry   = (state == thinking) and
                      timeToChangeStateand 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);
  elsewhen 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;
  // If the neighbors no longer has the flags
  // 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.

```
connector MutexPortIn "Mutex port connector " +
                      "for receiveing requests"
  input  Boolean request "Set by application "+
                         "to request access";
  input  Boolean release "Set by application "+
                         "to release access";
  output Boolean ok  "Signal that ownership " +
                     "was granted";
end MutexPortIn;
```

```
connector MutexPortOut   "Application mutex " +
                  "port connector for access"
  output Boolean request "Set this to " +
              "request ownership of the mutex";
  output Boolean release "Set this to " +
              "release ownership of the mutex";
  input  Boolean ok "This signals that " +
                    "ownership was granted";
end MutexPortOut;
```

```
model Mutex "Mutual exclusion of shared " +
            "resource"
  parameter Integer n = 5  "The number of " +
                           "connected ports";
  MutexPortIn[n] port;
protected
  Boolean request[n];
  Boolean release[n];
  Boolean ok[n];
  Boolean waiting[n];
  Boolean occupied     "Mutex is locked if " +
                      "occupied is true";
equation
    ok      = port.ok;
    request = port.request;
    release = port.release;
algorithm
  for i in 1:n loop
    when request[i] then
      if not occupied then
        ok[i]      := true;
        waiting[i] := false;
      else
        ok[i]      := false;
        waiting[i] := true;
      end if;
      occupied := true;
    end when;
    when pre(waiting[i]) and not occupied then
      occupied   := true;
      ok[i]      := true;
      waiting[i] := false;
    end when;
    when pre(release[i]) then
      ok[i]    := false;
      occupied := false;
    end when;
  end for;
end Mutex;
```

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

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.

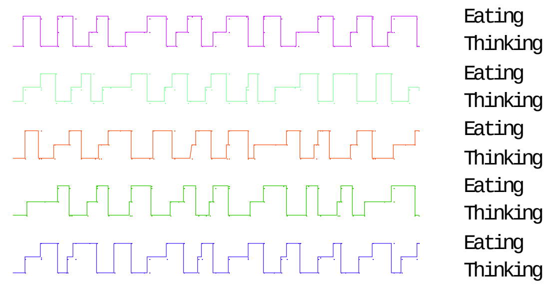

Figure 2. The result of simulating the dining table. Each philosopher alters between Thinking and eating with the hungry state in the middle.

References

[1] H.Elmqvist,S.E.Mattsson,and M.Otter. *A language for physical system modeling, visualization and interaction*. In Proceedings of the 1999 IEEE Symposium on Computer Aided Control System Design, Hawaii, Aug. 1999.

[2] H.Elmqvist and S.E.Mattsson.Modelica – *the next generation modeling language –an international design effort*. In Proceedings of the First World Congress on System Simulation, Singapore, Sept.1 –3 1997.

[3] Elmqvist H., Mattsson S. E., Otter M. (2000) *Object-Oriented and Hybrid Modeling in Modelica*. ADPM 2000, Dortmund, Germany

[4] P.Fritzson and V.Engelson. *Modelica —A unified object-oriented language for system modeling and simulation*. In Eric Jul, editor, ECOOP '98 —*Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 67 –90.Springer, 1998.

[5] Mosterman P.J., Otter M., Elmqvist H. (1998). *Modeling Petri Nets as Local Constraint Equations for Hybrid Systems Using Modelica*. The Proceedings of the Summer Computer Simulation Conference, July 19-22, S. 314-319, 1998 Reno, Nevada, U.S.A.

[6] Modelica Association http://www.Modelica.org

[7] Silberschantz A., Galvin P. B. *Operating System Concepts, Fourth Edition*. Addison-Wesley Publishing Company. ISBN 0-201-59292-4

[8] Halbwachs, Nicolas. *Synchronous Programming of Reactive Systems*. Kluwer Academic, 174 pages, ISBN 0-7923-9311-2, 1993.

[9] Peterson, James L. *Petri Net Theory and the Modeling of Systems*. Prentice Hall, 1981.

[10] P. Fritzson, *Principles of Object-Oriented Modeling and Simulation with Modelica*, Wiley-IEEE Press, Nov. 2003