Metropolis Polychrony Platform: User's Manual

Alessandro Pinto
University of California at Berkeley
545P Cory Hall, Berkeley, CA 94720
apinto@eecs.berkeley.edu

September 14, 2004

Copyright © 2001-2004 The Regents of the University of California.
All rights reserved.

Contents

1 Execution semantics

A more appropriate name for this platform would be *multi-rare synchronous*. The semantics is defined as a sequence of rounds. Let $\{P_i\}$ be a set of communicating processes. When a process P_i executes, it execute a sequence of three actions $write_i$, $read_i$ and $execute_i$. $write_i$ writes the content of the internal output buffer O_i to the output ports. $read_i$ read from the input channels and store the data into an internal buffer I_i. Also a process is characterized by two numbers: $P_i.r$ which is the execution rate and $P_i.p$ which is the process priority.

Each round has satisfy the following constraints:

- process P_i has to be executed $P_i.r$ times;
- if $P_i.p > P_j.p$ then P_i has to finish his execution in this round before P_i can execute.

Processes communicate through one-input one-output FIFOs. An variable can take the empty value $\perp$.

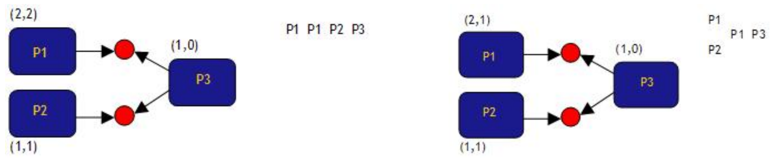

Figure 1: Execution examples of the multi-rate synchronous platform

Figure 1 shows two examples of possible executions. Processes are represented by rounded rectangles while medium are represented by circles. Each process is characterized by the pair of number $(P_i.r, P_i.p)$.

The left side of the picture shows a case where priorities are all different. A round execution is also shown in figure. Since P_1 has the highest priority it will be executed first as many times as its rate indicates. Then P_2 is executed once and finally P_3 is executed.

The right side of the picture shows a case P_1 and P_2 have the same priority. In this case the two processes can execute concurrently. However, $P_2.r$ is equal to one so it will finish after one execution. Then P_1 can finish its execution and finally P_3 which is the lowest priority process.

2 Motivation, platform structure and examples

Polychrony platform provides a set of predefined components to model synchronous multirate systems. Such systems are characterized by a centralized notion of clock round within which computation and communication are carried out in a specific order (see section 1).

There are many example of synchronous systems. Hardware designer are used to this kind of modeling framework where each component has *clock* input whose event is assumed to reach every component at the same time. In this kind of application, components can be considered as sequential machines communicating through registers (memory elements).

While the polychrony platform can be used in this way, each component is parametrized to allow the user to build more sophisticated multirate systems. Multirate systems are widely used in signal processing applications where up-sampling and down-sampling plays a fundamental role.

In polychrony, each process has a **rate** parameter indicating the number of requested execution in each clock round. Depending on the nature of the application and on the user's needs, the components' order execution can be randomly chosen by the platform or can be specified by an appropriate setting of the **priority** parameter. Using priority, the user can specify a specific schedule of the components within a round. This parameter allows the specification of data-flow models where an optimal schedule has been already computed.

Polychrnony platform provides also a formal system
de-synchronization. This feature is implemented as a refinement step.

Components provided by this platform are:

- `SynchTopNetlist`
- `SynchScheduledNetlist`
- `SynchSchedulingdNetlist`
- `SynchProcess`
- `SynchSignal`

There are many other components that are not exposed to the user but are internally used by the platform. Each component provides a set of services that will be explained in the following sections.

This document describes the usage of polychrony platform by going through a concrete modeling example.

2.1 The synchronous leader election algorithm

Consider a network of processes where each process has a unique identifier. Assume that the set of all the identifier is totally ordered. The leader election algorithm must eventually identify one process as leader of the network.

A simple algorithm for solving this problem is the *FloodMax* algorithm. The algorithm can be described as follows:

This algorithm is executed at every round. All processes have rate equal to one and all of them have the same priority. This means that all processes must be executed concurrently in each clock round.

2.2 Producer/consumer example

Producer/consumer example is a very simple system where a *producer* process P has one output $P.out$, and a *consumer* process C has one input $C.in$. $P.out$ is connected to $C.in$.

Algorithm 1 Synchronous Leader Election

Require: u this process unique identifier
Require: $diam \geq 0$ to be the diameter of the network
Require: $status \leftarrow unknown$, $rounds \leftarrow 0$, $maxid \leftarrow 0$

```
$rounds \leftarrow rounds + 1$
let $U$ be the set of all IDs form in neighbors
$maxid = \max(\{maxid\} \cup U)$
**if** $rounds \geq diam$ **then**
  **if** $maxid = u$ **then**
    $status \leftarrow leader$
  **else**
    $status \leftarrow nonleader$
  **end if**
**else**
  send $maxid$ to all out neighbors
**end if**
```

In this simple example, P produces an integer and C consumes the an integer at every reaction. This example is a test to try different values for priority and rate.

3 How to describe a synchronous process

Polychrony platform provides a base class `SynchProcess` to describe a synchronous process. This is the computation part of this model of computation.

`SynchProcess` constructor takes the following parameters:

- `String n`: a name for this class;
- `int ninput`: the number of input ports;
- `int noutput`: the number of output ports;
- `int rate`: the number of executions per clock round;
- `int pr`: this process priority.

It provides three methods that are of interest for the user:

- `public Object Value(int pn)`: this method returns the value of the signal connected to the input port whose number is pn. Each port has a number and is user's responsibility to associate a meaning to a port number. Since connections of processes is done by using port numbers, the meaning of a port depends on numbers are used to connect them (see section 4).
- `public void Post (int pn,Object data)`: this method writes the content of $data$ into the output buffer whose number is pn. From the user point of view, it is like writing the data on the output port whose number is pn.
- `public boolean isPresent(int pn)`: this method returns true if the input whose number is pn is present. Since this is a synchronous platform, a signal can be present or absent. Absence of signals is encoded by the special value `null`.

Also there are some variables that are accessible to the user and that are very useful in the description of processes:

- `int _rounds`: it counts the number of times a process executes;
- `int _numberofinput`: is the number of input ports;
- `int _numberofoutput`: is the number of output ports;

We want to describe a node of a network that behaves like the algorithm in section 2.1. First of all we must import the polychrony package. Assuming that the path `$(METRO)/lib` is in `METRO_CLASSPATH`, we can import the package as follows:

```
import metamodel.plt.polychronyscaled.*;
```

Then we define a process called `netnode` implementing our algorithm:

```
process netnode extends SynchProcess {
  int _uid;
  int _diam;
  int _maxuid;
  boolean _isleader;
  public netnode(String _n,
                 int nin, int nout,
                 int clock,int p,
 9                   int uid,int diam){
10      super(_n,nin,nout,clock,p);
11      _uid = uid;
12      _diam = diam;
13      _maxuid = uid;
14      _isleader = false;
15    }
16
17    public void execute(){
18      int i;
19      if (_rounds < _diam){
20        for(i=0;i<_numberofoutput;i++){
21          Post(i,new Integer(_maxuid));
22        };
23      };
24      for (i=0;i<_numberofinput;i++){
25        if (isPresent(i)){
26          if (((Integer)Value(i)).intValue() > _maxuid){
27            _maxuid = ((Integer)Value(i)).intValue();
28          };
29        };
30      };
31      if (_rounds == _diam){
32        if(_maxuid == _uid){
33          _isleader = true;
34          blackbox(SystemCSim)%%
35           cout <<
36           "I am the leader with UID = " <<
37           _uid << endl;
38           sc_stop();
39          %%
40        };
41      };
42    };
43  }
```

Line 1 declares a process `netnode` which extends the base class `SynchProcess` provided with the platform. Lines $2-5$ defines some internal variables like the unique user identifier `_uid`, the network diameter `_diam`, the maximum identifier `_maxuid` and a flag `_isleader` which indicates whether the process is the network elected leader or not. Lines $6 - 15$ are the process constructor. It takes the same parameters of a SynchProcess and also the unique identifier and the network diameter. Line 10 calls the SynchProcess constructor with parameters like name, number of inputs, number of outputs, rate and priority. Constructor of the base class will create the necessary ports and all the internal data structure which is totally transparent to the user. Then internal fields are initialized.

To specify a behavior, users have to overload the execute function (this is basically a standard procedure common to almost all the platform in metropolis). This function gets executed as many times in a clock rounds as specified by the rate parameter. The algorithm description is pretty evident and I will go through the most important part only.

Code in lines $19 - 23$ writes the maximum identifier found until now to all output ports. In this snippet of code it is possible to see the use of the internal fields `_rounds` and `_numberofinput`. Also `Post` is used to write on the outputs.

Code in line $24 - 30$ computes the maximum identifier. It first checks is the input is present (line 25) and, if it is present, reads the input value and compare it with the maximum id found until now.

Finally, lines $31 - 41$ check if the number or round reached the network diameter in which case if the maximum id is equal to the process id then the process is the network leader. When the number of rounds is equal to the network diameter we know that a leader has been elected, meaning that the algorithm has successfully terminated. I have introduced a blackbox (line $34 - 38$) to stop the simulation.

4 How to describe a synchronous/multirate system

An entire system can be described as interconnection of synchronous processes and signals. Polychrony platform provides a base netlist, `SynchScheduledNetlist`, that the user extends to describe her/his system.

`SynchScheduledNetlist` implements a set of elaboration method that can be called in the netlist constructor to build the sytem. The scheduled netlist constructor takes three parameters:

- `String n`: the netlist name;
- `int nproc`: the number of synchronous processes in the netlist.
- `int nssignal`: the number of synchronous signals in the netlist.

The scheduled netlist constructor builds the internal data structure to allow the implementation of elaboration methods described below. The internal data structure is also essential to implement the de-synchronization refinement algorithm.

The methods that are relevant are:

- `public elaborate void addSynchProcess(SynchProcess p)`: this method add process p to the netlist and stores a referent to p into an array for future purposes. Also it assigns a unique identifier to p. Finally it connects the process to a special component which is called clock assistant which is totally transparent to the user (to know more about the clock assistant please refer to the comments in the code).

- `public elaborate void addSynchSignal(SynchSignal s)`: this method adds a synchronous signal to the current netlist and stores a reference to s into an array.

- `public elaborate void synchConnect(SynchProcess p,int pn,boolean isinput,SynchSignal m)`: this method connects port pn of process p to the synchronous signal m. Here the user decided the binding between a port number and its use inside the process (this could be a bit tricky; in my opinion a process should define an enumeration variable to associate a port name to a port number). Boolean variable $isinput$ indicates is the port is an input port or an output port ($isinput = true$ means that p is an input port).

The following code is the description of a producer consumer example:

```
import metamodel.plt.polychronyscaled.*;
public netlist prodcons extends SynchScheduledNetlist {
  public prodcons(String n){
    super(n,2,2);
    producer p = new producer("Producer",1,1);
    consumer c = new consumer("Consumer",1,1);
    SynchSignal r = new SynchSignal("PtoC");
    SynchSignal a = new SynchSignal("CtoP");
    addSynchSignal(r);
    addSynchSignal(a);
    addSynchProcess(p);
    addSynchProcess(c);
    synchConnect(p,0,true,a);
    synchConnect(p,0,false,r);
15        synchConnect(c,0,true,r);
16        synchConnect(c,0,false,a);
17      }
18    }
```

Line 1 imports the required package. In line 2 the netlist declaration begins. The new netlist `prodcons` extends the base netlist `SynchScheduledNetlist`. Netlist constructor described the entire system and connects all the components. The first thing to do is to call the base class constructor with the appropriate parameters (line 4). This netlist contains two processes and two signals so the parent constructor is called with these two number as parameters. In line $5 - 8$ all components are instanced. Producer and consumer are two synchronous processes taking as parameters name, rate and priority.

After all components are created we have to add "register" them. We use the two methods provided by the base class to add synchronous signals and processes (lines $9 - 12$). Finally we establish synchronous connections between processes. Consider line 13: it says that output port number 0 of process p is connected to the synchronous signal a. Similarly, line 15 says that input port number 0 of process c is connected to the synchronous signal a. The two lines together connect output 0 of process p to input 0 of process c.

After the system is described in a scheduled netlist, we have to build a top level netlist and initialize the system.

5 How to write a top-netlist and simulate it

A scheduled netlsit is the description of how porcesses are interconnected to build the whole system. While computation and communication are described respectively by synchronous processes and synchronous signals, coordination is implemented by a scheduling netlist. The top level netlist contains both a scheduled and a scheduling netlist.

Polychrony platform provides a base class, `SynchTopNetlist`, that is meant to be a container for scheduled and scheduling netlsit. A top level netlist provides the following elaboration method that users can use:

- `public elaborate void initSynchTop(SynchScheduledNetlist synchscheduled)`: this method adds the scheduled netlist `synchscheduled` to the top netlist, creates an instance of a scheduling netlist, adds it to the current netlist and connects processes in the scheduled netslit to the scheduler in the scheduling netlist.

The scheduling netlist is then totally transparent to the user. In the case of producer consumer example, the top level netlist has the following structure:

```
1   import metamodel.plt.polychronyscaled.*;
2   public netlist topprodcons extends SynchTopNetlist{
3       public topprodcons(String n){
4           super(n);
5           prodcons prodconsnet =
6                  new prodcons("ProducerConsumerNetlist");
7           initSynchTop(prodconsnet);
8       }
9   }
```

As usual the first line imports the required package. Line 2 declares the top netlist to be an extension of the base class `SynchTopNetlist`. The netlist constructor instances the scheduled netlist (line 5) and then calls the `initSynchTop` method (line 6). At this point the netlist has been built and all connections have been made.