Metropolis YAPI Platform: User's Manual

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

September 14, 2004

Contents

1 Semantics **2**

2 The Yapi Package **2**
2.1 Yapi Channels 3
2.2 Yapi Processes 4

3 Example **6**

---

1 Semantics

This package provides a library for modeling *non-deterministic Kahn Process Networks*. This model has been developed by Philips and its formal semantics is described in [1].

In this model, processes communicate through of one input, one output unidirectional unbounded FIFOs. A process can have ports to read/write n tokens form/to a channel. Writing is non-blocking while the reading blocks the process until there are enough data to complete the read operation.

Non determinism is achieved by letting the process check the number of tokens available on its input channels. A process can non-deterministically choose a port among a set of ports where a read operation can eventually complete.

2 The Yapi Package

The yapi package is implemented using templates. A model is then parameterized with respect to the type of data that it deals with.

In order to use the package, the user must import it explicitly with the following import statement at the beginning of each mmm file:

```
import metamodel.plt.yapitemplate.*
```

This assumes that the METRO_CLASSPATH variable is set to $METRO lib.

The package provides two basic components: $yapiprocess$ to specify computation and $yapichannel$ that implements an unbounded FIFO.

2.1 Yapi Channels

The package provides the definition of the interface to communicate data. A basic interface is defined to check the number of tokens in a FIFO:

```
public interface yapiinterface extends Port {
    eval boolean checkfifo(int n,int dir);
}
```

The $checkfifo$ function takes two parameters: n is a number of tokens and dir is a direction associated with the port (0 for input and 1 for output). If we are checking an input port, this function returns true if there are at least n tokens in the FIFO. If we are checking an output port then it returns true if there are at least n available spaces in the FIFO. Since FIFOs are unbounded, this function always return true for output ports.

Two interfaces define the services that processes can use to exchange data:

```
//input port interface
template(T)
public interface yapioutinterface extends yapiinterface {
    update void write(T data);
    update void write(T[] data,int n);
    update void write(T[][] data, int n, int m);
}

//output port interface
template(T)
public interface yapiininterface extends yapiinterface {
    update T read();
    update void read(T[] data,int n);
    update void read(T[][] data,int n,int m);
}
```

$yapioutinterface$ is used as type for output ports. It defines services for writing into a channel. Three services are provided: for writing matrices, arrays and single tokens. $yapiininterface$ is used as type for input ports and defines services for reading from a channel.

Write is non-blocking since the fifo is unbounded. Unboundedness is achieved by resizing the FIFO each time that a write would overflow the current size.

$yapichannel$ is a template medium that implements these interfaces:

```
template(T)
public medium yapichannel implements yapiininterface-<T>-,
                                     yapioutinterface-<T>-,
```

```
                                        rdi,wri,cki {
  ...
  //Constructor
  public yapichannel(String n,int isize){
    super(n);
    ...
  }
  ...
}
```

where $isize$ is the FIFO initial size that has to be greater than zero.

2.2 Yapi Processes

The package provides the super-process *yapiprocess*. The user can extend this process and write his/her own *execute* function to modify the behavior of the process (see section **??**).

A process has four predefined input ports $inport0 - 3$ and four predefined output ports $output0 - 3$ that can be used when a non deterministic selection is needed. The user can define his/her own ports in addition to the predefined ports but select function is implemented on the predefined ports only. The Metal-Model language doesn't impose this limitation since it supports array of ports.

Yapiprocess implements the `select` function whose signature is the following:

```
int select(int nin0,int nin1,int nin2,int nin3,
             int nout0, int nout1, int nout2,int out3);
```

Here `nin<n>` indicates the number of tokens required at the n-th predefined input port and `nout<n>` indicates the number of places in the FIFO required at the n-th predefined output port. The ports that are considered in the select operation are only the ones that have a non negative number of tokens/spaces in the argument list. The select function returns a number that indicates which port was selected:

- 0-3 are the inputs ports inport[0-3]
- 4-7 are the output ports outport[0-3]

An input/output port is a candidate for selection if the number of tokens/spaces in the FIFO is greater or equal to the number of tokens/spaces required.

$yapiprocess$ is a template process whose definition is as follows:

```
template(T)
public process yapiprocess{
  port yapiininterface-<T>- inport0;
  port yapiininterface-<T>- inport1;
  port yapiininterface-<T>- inport2;
  port yapiininterface-<T>- inport3;

  port yapioutinterface-<T>- outport0;
  port yapioutinterface-<T>- outport1;
  port yapioutinterface-<T>- outport2;
  port yapioutinterface-<T>- outport3;

  public yapiprocess(String n){
    super(n);
  }

  public int select(int nin0,int nin1,int nin2,int nin3,
                    int nout0, int nout1, int nout2,int nout3){
    await{
      (inport0.checkfifo(nin0,0);;){
        return 0;
      }
      (inport1.checkfifo(nin1,0);;){
        return 1;
      }
      (inport2.checkfifo(nin2,0);;){
        return 2;
      }
      (inport3.checkfifo(nin3,0);;){
        return 3;
      }
      (outport0.checkfifo(nout0,1);;){
        return 4;
      }
      (outport1.checkfifo(nout1,1);;){
        return 5;
      }
      (outport2.checkfifo(nout2,1);;){
        return 6;
      }
      (outport3.checkfifo(nout3,1);;){
        return 7;
      }
    }
  }

  void thread() {
    execute();
  }
```

```
  public void execute(){}
}
```

The user extends this process and overrides the $execute$ method.

3 Example

This is a simple producer/consumer example that doesn't use the select function . The producer has only one output and generates random data. The consumer reads a number of tokens and sum them up. The number of tokens to sum is a parameter.

```
import metamodel.plt.yapitemplate.*;
process Producer extends yapiprocess-<yapiint>- {
  port yapioutinterface-<yapiint>- outport;
  Producer( String n , int numberofwrites ){
    super( n );
    _numberofwrites = numberofwrites;
  }
  public void execute( ) {
    while( _numberofwrites > 0 ){
      yapiint a = new yapiint( nondeterminism( int ) );
      outport.write(a);
      _numberofwrites--;
    }
  }
  int _numberofwrites;
}
```

The first line imports the $yapitemplate$ package. The $Producer$ process extends the $yapiprocess$. Since the base class is a template we have to specify the type that in this case is a class that contains and integer ($yapiint$). The process produces $numberofwrites$ integers whose values are non deterministic.

The top level netlist is described as follows:

```
import metamodel.plt.yapitemplate.*;
public netlist ProdCons {
  public ProdCons( String n ){
    super( n );
    Producer pr = new Producer( "TheProducer" , 50 );
    Consumer cs = new Consumer( "TheConsumer" , 50 );
    yapichannel-<yapiint>- ch = new yapichannel-<yapiint>-( "TheChannel" , 10 );
    addcomponent( pr , this , "TheProducerInstance" );
    addcomponent( cs , this , "TheConsumerInstance" );
```

```
    addcomponent( ch , this , "TheChannelInstance" );
    connect( pr , outport , ch );
    connect( cs , inport , ch );
}
```

Each component is instantiated and the added to the current netlist. Note that the $yapichannel$ are templates and their type has to be specified.

References

[1] W. J. M. Smits P. van der Wolf J.-Y. Brunel W. M. Kruijtzer P. Lieverse K. A. Vissers E. A. de Kock, G. Essink. Yapi: Application modeling for signal processing systems. *Proceedings of the Design Automation Conference*, June 2000.