Process Networks in Ptolemy II

by

Mudit Goel

mudit@eecs.berkeley.edu

Technical Memorandum UCB/ERL M98/69

Electronics Research Laboratory, Berkeley, CA 94720 December 16, 1998

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Master of Science

in

Electrical Engineering

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

1998

Abstract

Process Networks in Ptolemy II

by

Mudit Goel

Master of Science in Electrical Engineering

University of California at Berkeley

Professor Edward Lee, Chair

To model hardware and embedded applications, a highly concurrent model of computation is required. We present a mechanism to model concurrency using the Kahn process networks model of computation. The process networks model of computation has a dataflow flavor to it. This makes it well suited for modeling embedded dataflow applications and hardware architectures. Java provides a low level mechanism for constructing concurrent systems using threads and synchronizing monitors. We provide an implementation of process networks that is based on Java threads and is part of a heterogeneous modeling and design environment called Ptolemy II. The process networks model of computation has been extended to enable mutations of networks in a non-deterministic way. This can be used to model applications with migrating code, agents, and arrivals and departures of customers and services.

Contents

List of Figures

To my parents.

Acknowledgements

I would like to thank my advisor Edward A. Lee for introducing me to the wonderful world of research in this field. I thank him for providing me with valuable support and encouragement throughout the duration of my project.

I would also like to thank John Davis, John Reekie, Christopher Hylands and other members of the Ptolemy research group for their immense help and the valuable, thoughtful discussions that I had with them.

I especially thank my parents for their unflagging support for my education. Without their support, this report would have been a distant dream.

Chapter 1

Introduction

Embedded applications often have a high level of concurrency. These applications normally run on separate hardware components which run in parallel and cooperate with each other to achieve the functionality intended by the system as a whole. To design these embedded systems, a model of computation that can model the concurrency in a system is required. Process networks is a model of computation that has a high degree of concurrency and is well suited for embedded system software and for hardware implementations.

We provide the user with an implementation of process networks in a heterogeneous modeling and design framework called Ptolemy II [1]. This is implemented in Java. Java provides a low level mechanism to model concurrent behavior using threads and synchronizing monitors [2]. Getting the synchronization correct and using the threads safely is quite difficult. Our implementation of process networks uses this low level mechanism and provides an additional layer of abstraction over it. This makes it easier for the user to model and design concurrent systems correctly.

This chapter gives an overview of the theory behind the process networks model of computation. The second chapter provides a description of the software environment for process networks in Ptolemy II. The third chapter uses a few examples to demonstrate the way of modeling applications using the process networks model of computation in Ptolemy II. The last chapter has some discussion on possible extensions to the process network model of computation and their applications.

1.1 Kahn Process Networks - The Semantics

1.1.1 Denotational Semantics

A Kahn process network [3] is a network of processes that communicate only through unidirectional, single input single output FIFO channels with unbounded capacities. Each FIFO channel carries a possibly infinite sequence of data values called *tokens*. Any token in a channel can be read only once from the channel.

A description of the denotational semantics can be found in [3] and [4]. A brief summary is provided here. We start by defining some notation.

Prefix Ordering

Consider a set of tokens **S**. Let $\mathcal{S}^1$ be the set of all possible sequences of tokens from **S**, and $\mathcal{S}^M$ be the set of all sequences of M-tuples of tokens from **S**. $\mathcal{S}^1$ includes the empty sequence λ and $\mathcal{S}^M$ includes the empty sequence Λ.

For $X, Y \in \mathcal{S}^1$, the sequence X *precedes* the sequence Y (written as $X \sqsubseteq Y$), if X is a prefix (initial segment) of or is equal to Y. For example, $x_1x_2 \sqsubseteq x_1x_2x_3$. Now consider $\mathbf{X}, \mathbf{Y} \in \mathcal{S}^M$ where $\mathbf{X} = \{X_1, X_2, \ldots, X_M\}$ and $\mathbf{Y} = \{Y_1, Y_2, \ldots, Y_M\}$. We say $\mathbf{X} \sqsubseteq \mathbf{Y}$ if $X_i \sqsubseteq Y_i, \forall i \in 1, \ldots, M$. (Note that $X_i, Y_i \in \mathcal{S}^1, \forall i \in 1, \ldots, M$.) This defines a *prefix ordering* of sequences from $\mathcal{S}^M$. Also, $\Lambda \sqsubseteq \mathbf{X}, \forall \mathbf{X} \in \mathcal{S}^M$.

Partial Order

We next define a *partial order* [5] on a set $\mathcal{R}$. A binary relation $\leq$ is a partial ordering relation on $\mathcal{R}$ if $\forall x, y, z \in \mathcal{R}$, we have:

1. Reflexivity: $x \leq x$

2. Antisymmetry: $x \leq y$ and $y \leq x$ imply $x = y$

3. Transitivity: $x \leq y$ and $y \leq z$ imply $x \leq z$

As can be shown easily, the prefix ordering operator, $\sqsubseteq$, defines a partial ordering relation on $\mathcal{S}^M$.

Complete Partial Order and Continuous Mapping

Consider a set $\mathcal{R}$ with a partial ordering relation, $\leq$. $\perp \in \mathcal{R}$ is said to be a *bottom element* of $\mathcal{R}$ if $\perp \leq x$ for every $x \in \mathcal{R}$.

Now, let D be a non-empty subset of $\mathcal{R}$. It is said to be *directed* if, for every finite subset F of D, there exists a $z \in D$, such that $x \leq z, \forall x \in F$.

A partially ordered set $\mathcal{R}$ is a set with a partial ordering relation. It is a *complete partial order* (cpo) if

1. $\mathcal{R}$ has a bottom element, $\perp$.

2. For each directed subset D of $\mathcal{R}$, there exists a lowest upper bound $\sqcup D \in \mathcal{R}$.

In the case of the partial order defined above for $\mathcal{S}^M$, Λ is an element of $\mathcal{S}^M$. This is the bottom element in $\mathcal{S}^M$. Thus $\mathcal{S}^M$ satisfies the first of the two conditions for being a cpo. In $\mathcal{S}^M$, if for any increasing chain of sequences, $\chi : \mathbf{X_1} \sqsubseteq \mathbf{X_2} \sqsubseteq \cdots \sqsubseteq \mathbf{X_n} \sqsubseteq \cdots$, the limit with $n \to \infty$ exists, then the chain has a least upper bound (possibly an infinite sequence) denoted by $\sqcup\chi$, where $\sqcup\chi \in \mathcal{S}^M$. Now the elements of any directed subset of $\mathcal{S}^M$ will form an increasing chain as above and thus have a lowest upper bound in $\mathcal{S}^M$. This satisfies the second of the two conditions for a cpo, making $\mathcal{S}^M$ a cpo with the binary relation $\sqsubseteq$.

A mapping from a cpo $\mathcal{A}$ into a cpo $\mathcal{B}$ is said to be *continuous* if, for any increasing chain a of $\mathcal{A}$,

$$f(\lim_{\mathcal{A}} a) = \lim_{\mathcal{B}} f(a)$$

Kahn Process Networks

We interpret Kahn process networks as follows. For every process P in the network, we associate a set $\mathbf{S}_P$, which contains all the different types of tokens that the input channels may carry. Similarly, we associate a set $\mathbf{T}_P$ with the output channels of P, which contains all the different types of tokens that the output channels may carry. Also denote the number of input channels to the process by N, and the number of output channels by M.

The history of all the input channels to the process P forms a cpo $\mathcal{S}_P^N$ with the partial ordering relation $\sqsubseteq$, and the history of all the output channels from the process forms a cpo $\mathcal{T}_P^M$ with the partial ordering relation $\sqsubseteq$. Note that $\mathcal{S}_P^N$ and $\mathcal{T}_P^M$ are sequences of N-tuples and M-tuples of tokens from $\mathbf{S}_P$ and $\mathbf{T}_P$ respectively. A Kahn process P is a continuous mapping f_P such that

$$f_P : \mathcal{S}_P^N \rightarrow \mathcal{T}_P^M$$

Denotational semantics are good for describing a model of computation, but one requires the operational semantics to arrive at an implementation.

1.1.2 Operational Semantics

The most common operational semantics used is the Kahn MacQueen [6] semantics. According to this, each process executes a sequential program. Processes communicate with each other only through unbounded, unidirectional FIFO channels. A process can read from an input channel or write to an output channel at any time, with an additional constraint that it cannot poll a channel for the presence of data. If a process tries to read from an input channel that has no data, the read blocks and the process waits until a token is available on that channel. The above constraints (sequential processes and blocking reads) make the processes monotonic. Monotonic processes can be shown to make a network of processes deterministic. This implies that the sequence of tokens passing through the channels depends only on the topology and not on its implementation.

1.2 Bounded Memory Execution and Termination

The high level of concurrency in process networks makes it an ideal match for embedded system software and for hardware implementations. But the Kahn MacQueen semantics do not guarantee bounded memory execution of process networks even if it is possible for the application to execute in bounded memory. Most real-time embedded applications and hardware processes are intended to run indefinitely with a limited amount of memory. Thus bounded memory execution of process networks becomes crucial for its usefulness for hardware and embedded software.

Parks [7] addresses this aspect of process networks and provides an algorithm to make a process networks application execute in bounded memory whenever possible. He provides an implementation of the Kahn MacQueen semantics that assigns a fixed capacity to each FIFO channel. In addition to blocking on a read from a channel, a process can block on a write to a channel if the FIFO channel has reached its capacity. Deadlocks can now occur when all processes are blocked either on a read or on a write to a channel. On detection of a deadlock, if there are some processes blocked on a write to a channel, Parks chooses the channel with the smallest capacity among the channels on which processes are blocked on a write and increases its capacity to break the deadlock. If all the processes are blocked on a read from a channel, then the network cannot execute further and can be terminated.

Chapter 2

Implementation

Ptolemy 0.x [8] is a tool for heterogeneous modeling and design of concurrent systems. It provides a highly flexible, extensible, object oriented foundation for specifying, simulating and synthesizing systems. Its main strength is in modeling complex heterogeneous systems at various levels of abstraction. It is divided into domains, each of which implements a model of computation. A system can be designed using an appropriate domain or combination of domains. Some of the models of computation in Ptolemy are untimed dataflow like the Synchronous Dataflow (SDF), and the timed Discrete Event (DE) models of computation.

Ptolemy II [1] is a complete redesign of Ptolemy 0.x in Java. The underlying structure of Ptolemy 0.x is tailored to the dataflow models of computation, and is not very natural for implementing other models of computation. Also the mutations of networks (dynamically changing graphs) that it permits are extremely limited. These are some of the issues being addressed in the redesign. In addition, the use of Java makes the Ptolemy II software appletable and platform independent.

The Kahn process networks model of computation is implemented in both Ptolemy 0.x and Ptolemy II as the Process Networks (PN) domain. The implementation is based on the bounded memory execution algorithm proposed by Parks [7]. In Ptolemy II, a mechanism of detecting deadlocks proposed by Laramie, et. al. [9] is used to determine the termination condition in the PN domain. Partial non-determinism in the form of dynamically changing or mutating graphs is also supported in the PN domain in Ptolemy II.

Ptolemy II is modular and is divided into packages, each of which provide separate functionalities. The abstract syntax is separated from the mechanisms that attach semantics. In PN, the package that attaches the process networks semantics is `ptolemy.domains.pn.kernel`. A UML static structure diagram [10] of the classes and methods directly related to the PN domain in Ptolemy II is shown in fig.2.1. The following sections discuss the implementation in detail. Some examples of applications written in the PN domain in Ptolemy II are presented in the next chapter.

2.1 Construction of a Graph

A graph or a topology in Ptolemy II consists of *entities* and *relations*. Entities have *ports* which are normally used for transfer of tokens between entities. Relations connect the different ports.

Ptolemy II supports topological hierarchy using clustered graphs. Thus an entity in a graph can contain a subgraph (entities and relations). This feature not only facilitates representation of large graphs in a more concise and understandable way but also makes

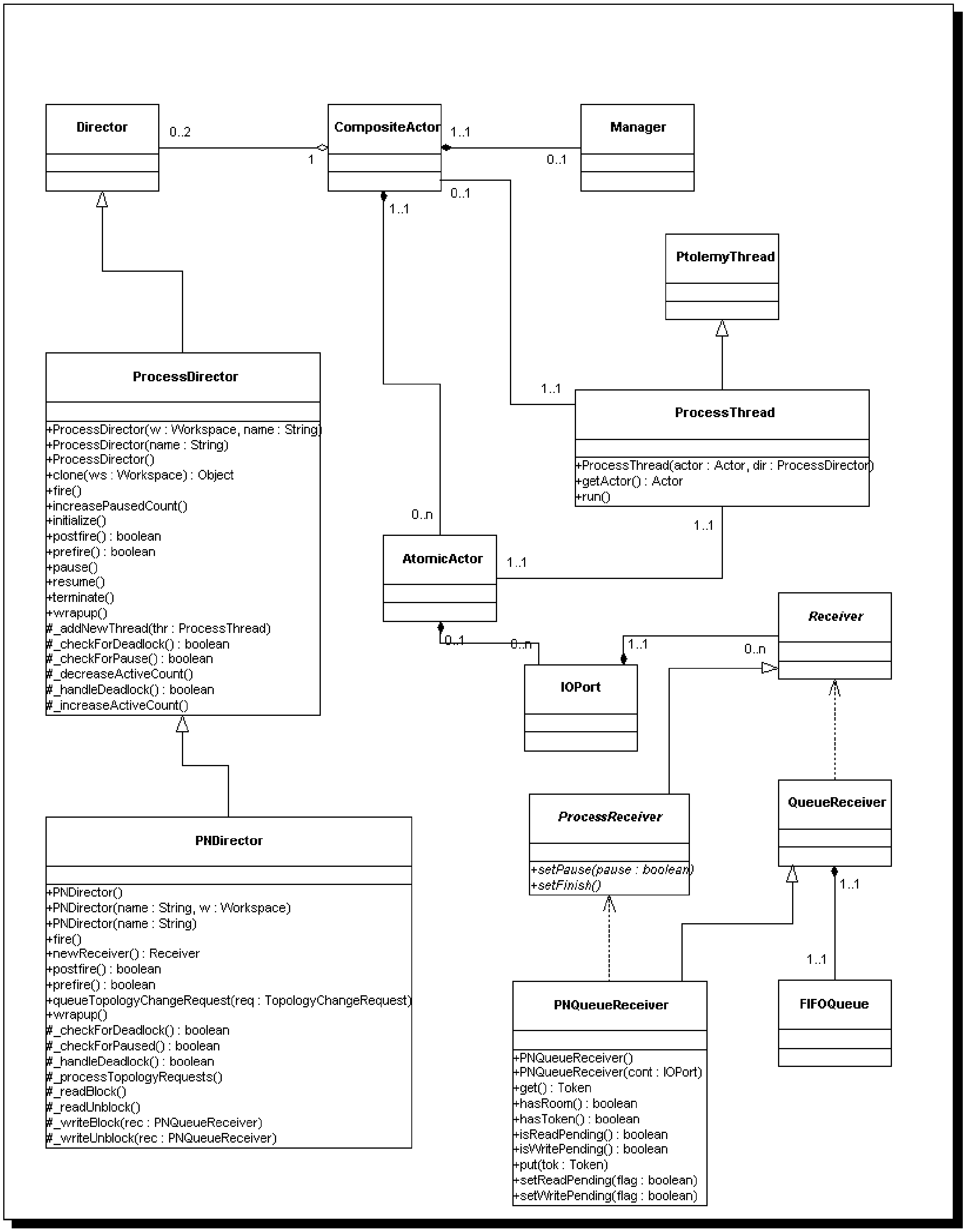
Director
0..2
1
CompositeActor
1..1
0..1
Manager
0..1
1..1
PtolemyThread
ProcessDirector
+ProcessDirector(w : Workspace, name : String)
+ProcessDirector(name : String)
+ProcessDirector()
+clone(ws : Workspace) : Object
+fire()
+increasePausedCount()
+initialize()
+postfire() : boolean
+prefire() : boolean
+pause()
+resume()
+terminate()
+wrapup()
#_addNewThread(thr : ProcessThread)
#_checkForDeadlock() : boolean
#_checkForPause() : boolean
#_decreaseActiveCount()
#_handleDeadlock() : boolean
#_increaseActiveCount()
1..1
ProcessThread
+ProcessThread(actor : Actor, dir : ProcessDirector)
+getActor() : Actor
+run()
0..n
1..1
AtomicActor
1..1
Receiver
0..1
0..n
1..1
0..n
IOPort
ProcessReceiver
+setPause(pause : boolean)
+setFinish()
QueueReceiver
1..1
1..1
PNDirector
+PNDirector()
+PNDirector(name : String, w : Workspace)
+PNDirector(name : String)
+fire()
+newReceiver() : Receiver
+postfire() : boolean
+prefire() : boolean
+queueTopologyChangeRequest(req : TopologyChangeRequest)
+wrapup()
#_checkForDeadlock() : boolean
#_checkForPaused() : boolean
#_handleDeadlock() : boolean
#_processTopologyRequests()
#_readBlock()
#_readUnblock()
#_writeBlock(rec : PNQueueReceiver)
#_writeUnblock(rec : PNQueueReceiver)
PNQueueReceiver
+PNQueueReceiver()
+PNQueueReceiver(cont : IOPort)
+get() : Token
+hasRoom() : boolean
+hasToken() : boolean
+isReadPending() : boolean
+isWritePending() : boolean
+put(tok : Token)
+setReadPending(flag : boolean)
+setWritePending(flag : boolean)
FIFOQueue

Figure 2.1: UML static structure diagram of classes involved directly in the implementation of the PN domain in Ptolemy II

the graph more modular, by letting each cohesive part of the graph be represented as a separate subgraph. Every such subgraph is represented as a single entity at the level of hierarchy above it.

An executable entity is called an *actor*. If it contains a graph, then it is a *composite actor* and is an instance of `ptolemy.actor.CompositeActor` or of a derived class. If it is atomic and does not contain a graph, then it is an *atomic actor* and is an instance of `ptolemy.actor.AtomicActor` or a derived class. The relations are instances of `ptolemy.actor.IORelation` and the ports are instances of `ptolemy.actor.IOPort`. An IOPort can be an input port, an output port or both. An input port has *receivers* embedded in it. These receivers are capable of receiving tokens from distinct channels. In the PN domain, each receiver contains a FIFO queue in it.

Thus in a process networks graph, the nodes (processes) correspond to actors. The FIFO channels are embedded in the receivers in the input ports. The entire graph is constructed in an instance of a composite actor, called the top-level composite actor. It contains actors (both atomic actors and composite actors) and relations. More details about constructing a topology in Ptolemy II can be found in the design document [11].

2.2 The Execution Sequence

In Ptolemy II, a *manager* governs the overall execution of a model. This interacts with the top-level composite actor and calls the execution methods on it. A *director* is responsible for the execution of a composite actor. It performs any scheduling or any other actions that might be necessary for the execution of the components in the composite actor.

The execution methods on the director are called from the methods in the corresponding composite actor.

2.2.1 Director

In process networks, each node of the graph is a separate process. In the PN domain in Ptolemy II, this is achieved by letting each actor (node) have its own separate thread of execution. These threads are based on the native Java threads [12] and are instances of `ptolemy.actors.ProcessThread`.

In the PN domain, the director is an instance of `ptolemy.domains.pn.kernel.PNDirector`. It starts a separate thread for every actor in the topology in the beginning of the simulation. Each thread acts as a dataflow process by calling the execution methods of the dataflow actor repeatedly.

The first step in the execution is the call to the `initialize()` method of the director. This method creates the receivers in the input ports of the actors for all the channels and creates a thread for each actor. It initializes all the actors in the graph. It also sets the count of active actors in the model, which is required for detection of deadlocks and termination, to the number of actors in the composite actor.

The next stage is the iterations. It starts with a call to the `prefire()` method of the director. This method starts all the threads that were created for the actors in the `initialize()` method of the director. In PN, this method always returns `true`.

The `fire()` method of the director is called next. In PN, the `fire()` method is responsible for handling deadlocks (both real and artificial), as explained later in sec. 2.4.1. It is also responsible for making all the mutations in PN as described in sec. 2.5.

The last stage of the iteration cycle of the director is the call to the `postfire()` method. This method returns `true` unless a real deadlock is detected. A real deadlock is when no actor in the composite actor associated with the director has any token that it can read and process. That implies that unless some new data is to be provided to the graph, the simulation can be terminated. The new data can come from the topology at a higher level in the hierarchy if the deadlocked composite actor is not the top-level composite actor.

In case there is a real deadlock at the top-level composite actor and the simulation cannot proceed, the manager calls the `wrapup()` method of the top-level composite actor. This in turn calls the `wrapup()` method of the director. The director then terminates the simulation. Details of termination are discussed in sec. 2.4.1.

2.2.2 Execution of Actors

As mentioned earlier, a separate thread is responsible for the execution of each actor in PN. This thread is started in the `prefire()` method of the director. After starting, this thread repeatedly calls the `prefire()`, `fire()`, and `postfire()` methods of the actor. This sequence continues until the `postfire()` or the `prefire()` method returns `false`. The only way for an actor to terminate gracefully in PN is by returning from the `fire()` method and returning `false` in the `postfire()` or `prefire()` method of the actor. If an actor finishes execution as above, then the thread calls the `wrapup()` method of the actor. Once this method returns, the thread informs the director about the termination of this actor and finishes its own execution. This actor is not fired again unless the director creates and starts a new thread for the actor. Also, if an actor returns `false` in its `prefire()` method the first time it is called, then it is never fired in PN.

2.3 Message Passing

In Ptolemy II, data transfer between entities is achieved using ports and the receivers embedded in the input ports. Each receiver in an input port is capable of receiving messages from a distinct channel.

An actor calls the `send()` or `broadcast()` method on its output port to transmit a token to a remote actor. The port obtains a reference to a remote receiver (via the relation connecting them) and calls the `put()` method of the receiver, passing it the token. The destination actor retrieves the token by calling the `get()` method of its input port, which in turn calls the `get()` method of the designated receiver.

Both the `get()` and `send()` methods of the port take an integer index as an argument, which the actor uses to distinguish between the different channels that its port is connected to. This index specifies the channel to which the data is being sent or from which the data is being received. If the ports are connected to a single channel, then the index is 0. But if the port is connected to more than one channel (a multiport), say N channels, then the index ranges from 0 to $N-1$. The `broadcast()` method of the port does not require an index as it transmits the token to all the channels to which it is connected.

In the PN domain, these receivers are instances of
`ptolemy.domains.pn.kernel.PNQueueReceiver`. These receivers have a FIFO queue in them to provide the functionality of a FIFO channel in a process networks graph. In addition to this, these receivers are also responsible for implementing blocking reads and blocking writes. They handle this using the `get()` and the `put()` methods. These methods are as shown in figs. 2.2 and 2.3. The `get()` method checks whether the FIFO queue has

```
public Token get() {
    IOPort port = getContainer();
    Workspace workspace = port.workspace();
    Actor actor = (Actor)port.getContainer();
    PNDirector director = (PNDirector)actor.getDirector();
    Token result = null;
    synchronized (this) {
        while (!_terminate && !super.hasToken()) {
            director._readBlock();
            _readpending = true;
            while (_readpending && !_terminate) {
                workspace.wait(this);
            }
        }
        if (_terminate) {
            throw new TerminateProcessException("");
        } else {
            result = super.get();
            if (_writepending) {
                director._writeUnblock(this);
                _writepending = false;
                notifyAll(); //Wake up threads waiting on a write;
            }
        }
        while (_pause) {
            director.increasePausedCount();
            workspace.wait(this);
        }
        return result;
    }
}
```

Figure 2.2: `get()` method of the PNQueueReceiver

any tokens. If not, then it increases the count tracking the number of actors blocked on a read in the director and sets its `_readpending` flag to true. Then it suspends the calling thread until some actor puts a token in the FIFO queue and sets the `_readpending` flag of this receiver to false. (This is done in the `put()` method as described later.) When the calling thread is resumed, the `get()` method reads the first token from the FIFO queue. In case some actor is blocked on a write to this receiver (the FIFO queue is full to capacity), it unblocks that actor, notifies all the threads that are blocked, and returns. This method also handles the termination of the simulation as is explained later in sec. 2.4.2.

The `put()` method of the receiver is responsible for implementing blocking writes. This method checks whether the FIFO queue is full to capacity. If it is, then it sets its `_writepending` flag to true and informs the director that an actor is blocked on a write. Then it suspends its calling thread until another thread wakes it up after setting the `_writepending` flag to false. After this, it puts the token into the FIFO queue and checks whether some actor is blocked on a read from this receiver. If an actor is blocked on a read, it unblocks it and notifies all blocked threads. Then it returns.

2.4 Detection of Deadlocks and Termination

2.4.1 Deadlocks

The mechanism for detecting deadlocks in the Ptolemy II implementation of PN is based on the mechanism suggested in [9]. This mechanism requires keeping count of the number of threads currently active, paused, and blocked in the simulation. The number of threads that are currently active in the graph is set by a call to the `_increaseActiveCount()`

```
public void put(Token token) {
    IOPort port = getContainer();
    Workspace workspace = port.workspace();
    Actor actor = (Actor)port.getContainer();
    PNDirector director = (PNDirector)actor.getDirector();
    synchronized(this) {
        if (!super.hasRoom()) {
            _writepending = true;
            director._writeBlock(this);
            while(_writepending) {
                workspace.wait(this);
            }
        }
        super.put(token);
        if (_readpending) {
            director._readUnblock();
            _readpending = false;
            notifyAll();
        }
        while (_pause) {
            director.increasePausedCount();
            workspace.wait(this);
        }
    }
}
```

Figure 2.3: `put()` method of the PNQueueReceiver

method of the director. This method is called whenever a new thread corresponding to an actor is created in the simulation. The corresponding method for decreasing the count of active actors (on termination of a process) is `_decreaseActiveCount()` in the director.

Whenever an actor blocks on a read from a channel, the count of actors blocked on a read is incremented by calling the `_readBlock()` method in `PNDirector`. Similarly, the number of actors blocked on a write is incremented by a call to the `_writeBlock()` method of the director. The corresponding methods for decreasing the count of the actors blocked on a read or a write are `_readUnblock()` and `_writeUnblock()`, respectively. These methods are called from the instances of the `PNQueueReceiver` class when an actor tries to read from or write to a channel.

Every time an actor blocks on a read or a write, the director checks for a deadlock. If the total number of actors blocked (on a read or a write) equals the total number of actors active in the simulation, a deadlock is detected. On detection of a deadlock, if one or more actors are blocked on a write, then this is an artificial deadlock. The channel with the smallest capacity among all the channels with actors blocked on a write is chosen and its capacity is incremented. This implements the bounded memory execution as suggested by Parks [7].

If there are no actors blocked on a write, then the deadlock detected is a real deadlock. If a real deadlock is detected at the top-level composite actor, then the manager terminates the execution.

```
public synchronized void setFinish() {
    _terminate = true;
    notifyAll();
}
```

Figure 2.4: `setFinish()` method of the PNQueueReceiver

2.4.2 Termination of a simulation

A simulation can be ended (on detection of a real deadlock) by calling the `wrapup()` method on either the toplevel composite actor or the corresponding director. This method is normally called by the manager on the top-level composite actor. In PN, this method traverses the topology of the graph and calls the `setFinish()` method of the receivers in the input ports of all the actors. Since this method is called only when a real deadlock is detected, one can be sure that all the active actors in the simulation are currently blocked on a read from a channel and are waiting in the call to the `get()` method of a receiver. This fact is used to wrap up the simulation. The `setFinish()` method of the receiver sets the termination flag to true, and wakes up all the threads currently waiting in the `get()` method of the receiver (fig. 2.4). This is implemented using the `wait()-notifyAll()` mechanism of Java [2],[12]. Once these threads wake up, they see that the termination flag is set. This results in the `get()` method of the receivers throwing a `TerminateProcessException` (a runtime exception in Ptolemy II). This exception is never caught in any of the actor methods and is eventually caught by the process thread. The thread catches this runtime exception, calls the `wrapup()` method of the actor and finishes its execution. Eventually after all threads catch this exception and finish executing, the simulation ends.

2.5 Mutations of a graph

The PN domain in Ptolemy II allows graphs to mutate during execution. This implies that old processes or channels can disappear from the graph and new processes and channels can be created during the execution. This is demonstrated with an example in sec. 3.2.

Though other domains, like SDF, also support mutations in their graphs, there is a big difference between the two. In domains like SDF, mutations can occur only between iterations. This keeps the simulation determinate as changes to the topology occur only at a fixed point in the execution cycle. In PN, the execution of a graph is not centralized, and hence, the notion of an iteration is quite difficult to define. Thus, in PN, we let mutations happen as soon as they are requested. This form of mutations is generally non-deterministic as they can occur at any point during a simulation. The point in the execution where mutations occur would normally depend on the schedule of the underlying Java threads. Under certain conditions where the application can guarantee a fixed point in the execution cycle for mutations, or where the mutations are localized, mutations can still be determinate.

An actor can request a mutation by creating an instance of a class derived from `ptolemy.kernel.event.TopologyChangeRequest`. It should override the method `constructEventQueue()` and include the commands that it wants to use to perform mutations in this method. It should also list all the commands needed to inform the topology listeners about the changes made to the topology because of the mutations that the actor wishes to perform. This is demonstrated using an example in sec. 3.2. Further details can

```
    public void queueTopologyChangeRequest(TopologyChangeRequest req) {
        super.queueTopologyChangeRequest(req);
        synchronized(this) {
            _urgentMutations = true;
            notifyAll();
        }
    }
```

Figure 2.5: The `queueTopologyChangeRequest()` method of the director in PN

be found in the Ptolemy II design document [11].

After creating the above class, the actor calls the method, `queueTopologyChangeRequest(TopologyChangeRequest req)` on the director. This method (fig. 2.5) queues the requested mutations, and notifies the director, currently waiting in its `fire()` method, and asks it to take over the control. On detecting the request for mutations, the director calls its `_processTopologyRequests()` method to perform the mutations.

The `_processTopologyRequests()` method pauses the simulation and performs all the mutations. Then, it creates receivers and threads for all the new actors, if any, and starts the threads. It then resumes all the actors and returns. This method (called from the `fire()` method of the director) is shown in fig. 2.6.

The above sections summarize most of the implementation details of PN. But a developer should be cautious about a few issues while attempting to extend the PN domain in Ptolemy II. If not handled properly, these issues could result in errors in simulation. They are discussed in the next section.

```
protected void _processTopologyRequests()
        throws IllegalActionException, TopologyChangeFailedException {
    Workspace worksp = workspace();
    pause();
    super._processTopologyRequests();
    LinkedList threadlist = new LinkedList();
    Enumeration newactors = _newActors();
    while (newactors.hasMoreElements()) {
        Actor actor = (Actor)newactors.nextElement();
        actor.createReceivers();
        actor.initialize();
        ProcessThread pnt = new ProcessThread(actor, this);
        threadlist.insertFirst(pnt);
        _addNewThread(pnt);
    }
    resume();
    Enumeration threads = threadlist.elements();
    while (threads.hasMoreElements()) {
        ProcessThread pnt = (ProcessThread)threads.nextElement();
        pnt.start();
    }
}
```

Figure 2.6: The `_processTopologyRequests()` method of the director in PN that handles mutations

2.6 Important Issues in the Implementation

There are two main issues that a developer should be aware of while extending PN. The first one is to get the mutual exclusion right and the second is to avoid undetected deadlocks.

2.6.1 Mutual Exclusion using Monitors

In PN, threads interact in various ways for message passing, deadlock detection, etc. This requires various threads to access the same data structures. Concurrency can easily lead to inconsistent states as threads could access a data structure while it is being modified by some other thread. This can result in race conditions and undesired deadlocks [13]. For this, Java provides a low-level mechanism called a *monitor* to enforce mutual exclusion. Monitors are invoked in Java using the *synchronized* keyword. A block of code can be *synchronized* on a monitor lock as follows:

```
synchronized (obj) {
    ... //Part of code that requires exclusive lock on obj.
}
```

This implies that if a thread wants to access the synchronized part of the code, then it has to grab an exclusive lock on the monitor object, `obj`. Also while this thread has a lock on the monitor, no thread can access *any* code that is synchronized on the same monitor.

There are many actions (like mutations) that could affect the consistency of more than one object, such as the director and receivers. Java does not provide a mechanism to acquire multiple locks simultaneously. Acquiring locks sequentially is not good enough as this could lead to deadlocks. For example, consider a thread trying to acquire locks on objects a and b in that order. Another thread might try to obtain locks on the same objects in the opposite order. The first thread acquires a lock on a and stalls to acquire a lock on b, while the second thread acquires a lock on b and waits to grab a lock on a. Both threads stall indefinitely and the application is deadlocked.

The main problem in the above example is that different threads try to acquire locks in conflicting orders. One possible solution to this is to define an order or hierarchy of locks and require all threads to grab the locks in the same top-down order [2]. In the above example, we could force all the threads to acquire locks in a strict order, say a followed by b. If all the code that requires synchronization respects this order, then this strategy can work with some additional constraints, like making the order on locks immutable. Although this strategy can work, this might not be very efficient and can make the code a lot less readable. Also Java does not permit an easy and straightforward way of implementing this.

We follow a similar but easier strategy in the PN domain of Ptolemy II. We define a three level strict hierarchy of locks with the lowest level being the director, the middle level being the various receivers and the highest level being the *workspace*. The rule that all threads have to respect after acquiring their first lock is to never try acquiring a lock at a higher or at the same level as their first lock. Specifically, a block of code synchronized on the director should not try to access a block of code that requires a lock on either the workspace or any of the receivers. Also, a block of code synchronized on a receiver should not try to call a block of code synchronized on either the workspace or any other receiver.

Some discussion about these locks in PN is presented in the following section.

Hierarchy of locks

The highest level in the hierarchy of locks is the `Workspace`, which is a class defined specifically for this purpose. This level of synchronization though is quite different from the other two forms. This synchronization is modeled explicitly in Ptolemy II and is another layer of abstraction based on the Java synchronization mechanism. The principle behind this mechanism is that if a thread wants to read the topology, then it wants to read it only in a consistent state. Also if a thread is modifying the topology, then no other thread should try to read the topology as it might be in an inconsistent state. To enforce this, we use a reader-writer mechanism to access the workspace [11]. Any thread that wants to read the topology but does not modify it requests a read access on the workspace. If the thread already has a read or write access on the workspace, it gets another read access immediately. Otherwise if no thread is currently modifying the topology, and no thread has requested a write access on the workspace, the thread gets the read access on the workspace. If the thread cannot get the read access currently, it stalls until it gets it. Similarly, if a thread requests a write access on the workspace, it stalls until all other threads give up their read and write access on the workspace. Thus though a thread does not have an exclusive lock on the workspace, the above mechanism provides a mutual exclusion between the activities of reading the topology and modifying the topology. This way of synchronizing on the workspace is distinctly different from possessing an exclusive lock on the workspace.

Once a thread has a read or write access on the workspace, it can call methods or blocks of code that are synchronized on a single receiver or the director.

The receivers form the next level in the hierarchy of locks. These receivers are once again accessed by different threads (the reader and the writer to the queue) and need to be synchronized. For example, a writer thread might try to write to a receiver while another token is being read from it. This could leave the receiver in an inconsistent state. The state of a receiver might include information about the number of tokens in the queue, the information about any process blocked on a read or a write to the receiver and some other information. These methods or blocks of code accessing and modifying the state of the receivers are forced to get an exclusive lock on the receiver. These blocks might call methods that require a lock on the director, but do not call methods that require a lock on any other receiver.

The lower-most lock in the hierarchy is the `PNDirector` object. There are some internal state variables, such as the number of processes blocked on a read, that are accessed and modified by different threads. For this, the code that modifies any internal state variable should not let more than one thread access these variables at the same time. Since access to these variables is limited to the methods in director, the blocks of code modifying these state variables obtain an exclusive lock on the director itself. These blocks should not try to access any block of code that requires an exclusive lock on the receivers or requires a read or a write access on the workspace.

2.6.2 Undetected Deadlocks

Undetected deadlocks should be avoided while extending the PN domain in Ptolemy II. We discuss a significant but subtle issue that a developer should be aware of when trying to extend the PN domain. This concerns the release of locks from a suspended thread.

In Java, when a thread with an exclusive lock on multiple objects suspends by calling `wait()` on an object, it releases the lock only on that object and does not release other locks. For example, consider a thread that holds a lock on two objects, say a and b. It calls `wait()` on b and releases the lock on b alone. If another thread requires a lock on a to perform whatever action the first thread is waiting for, then deadlock will ensue. That thread cannot get a lock on a until the first thread releases its exclusive lock on a, and the first thread cannot continue until the second thread gets the lock on a from the first and performs whatever action it is waiting for.

This sort of scenario is currently avoided in PN by following some simple rules. The first of them being that a method or block synchronized on the director never calls `wait()` on any object. Thus once a thread grabs a lock on director, it is guaranteed to release it. The second is that a block of code with an exclusive lock on a receiver does not call the `wait()` method on the workspace. (Note that the code should never synchronize directly on the workspace object and should always use the read and write access mechanism.) The third rule is that a thread should give up all the read permissions on the workspace before calling the `wait()` method on the receiver object. Note that in case of workspace, we require this because of the explicit modeling of mutual exclusion between the read and write activities on the workspace. If a thread does not release the read permissions on the workspace and suspends, while a second thread requires a write access on the workspace to perform the action that the first thread is waiting for, a deadlock results. Also to be in a consistent state with respect to the number of read accesses on the workspace, the thread should regain those read accesses after returning from the call to the `wait()` method. For this a `wait(Object obj)` method is provided in the class `Workspace` that releases all the read accesses to the workspace, calls `wait()` on the argument `obj`, and regains all the read accesses on waking up.

The above rules guarantee that a deadlock does not occur in the PN domain because of contention for various locks.

Chapter 3

Examples

This chapter demonstrates examples of applications using the PN domain in Ptolemy II.

3.1 The basic structure

Let us consider the example shown in fig. 3.1, taken from Parks [7]. The two instances of the process *PNRedirect*, produce initial tokens 0 and 1 that allow the program graph to execute. The process *Commutator* reads tokens alternately from the two input channels it is connected to, X and Y, and redirects those tokens to the output channel Z. The process *Distributor* reads tokens from channel Z and writes tokens to channels S_0 and S_1 alternately. The instances of process *PNRedirect* read tokens from channels S_0 and S_1 and write them to channels X and Y respectively.

This simple network as can be seen does not do much. But this example is ideal to demonstrate the basics of modeling applications in PN. This example is modeled in PN

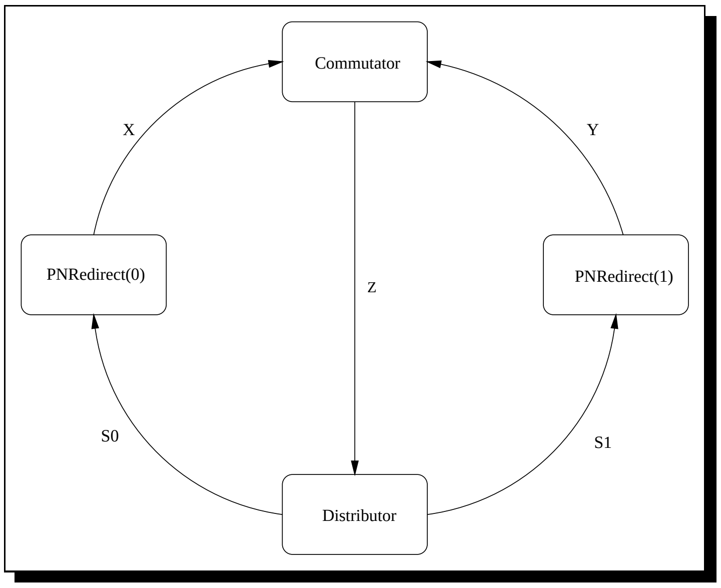

Figure 3.1: Graphical representation of a process network constructed with the processes defined in figs. 3.2, 3.3, 3.4.

using three different processes. The definition of these processes is shown in figs. 3.2, 3.3, and 3.4. These definitions are as coded in Java in the Ptolemy II framework. For clarity, the exception handling code has been removed.

The constructor defines the number of input and output ports that an actor or process has. For example the process *PNRedirect*, has a single input and a single output. They are defined as

```
_input = new IOPort(this, "input", true, false);
_output = new IOPort(this, "output", false, true);
```

An input port, by default, can be connected to only one channel. In case an actor wants to read tokens from more than one input channel and treat them similarly, independent of the number of input channels it is connected to, as in the case of *Commutator*, it can make a port multiport and connect it to multiple channels. The syntax for that is

```
public class Commutator extends AtomicActor {
    public Commutator(CompositeActor container, String name) {
        super(container, name);
        _input = new IOPort(this, "input", true, false);
        _input.makeMultiport(true);
        _output = new IOPort(this, "output", false, true);
    }

    public void fire() {
        for (int i=0; i<_input.getWidth(); i++) {
            _output.broadcast(_input.get(i));
        }
    }

    private IOPort _input;
    private IOPort _output;
}
```

Figure 3.2: A process - Commutator - that interleaves various input streams into one.

```
_input = new IOPort(this, "input", true, false);
_input.makeMultiport(true);
```

The default for an output port is the same as the input port and it can be connected to only one channel. In case an actor wants to provide different tokens on different channels connected to the same port, as in the case of *Distributor*, it can make an output port multiport with the same syntax as for an input port.

```
_output = new IOPort(this, "output", false, true);
_output.makeMultiport(true);
```

This implies that all the channels that are connected to this port will now receive different tokens in a cyclic order. The distributor reads tokens from its input and sends them in a cyclic order to all the different output channels that its output port is connected to. For this, it iterates through all the different channels connected to the port (the number of which is given by the `getWidth()` method of the port) and use the `send()` method to send

```
public class Distributor extends AtomicActor {
    public Distributor(CompositeActor container, String name) {
        super(container, name);
        _input = new IOPort(this, "input", true, false);
        _output = new IOPort(this, "output", false, true);
        _output.makeMultiport(true);
    }

    public void fire() {
        for (int i=0; i<_output.getWidth(); i++) {
            _output.send(i, _input.get(0));
        }
    }

    private IOPort _input;
    private IOPort _output;
}
```

Figure 3.3: A process - Distributor - that distributes elements of input streams to the output streams in a cyclic order.

a token to each of those channels, as done in the `Commutator` code.

```
for (int i=0; i<_output.getWidth(); i++) {
    _output.send(i, _input.get(0));
}
```

If an actor wants to send the same token to all the channels that a particular port is connected to, then it uses the `broadcast()` method as shown:

```
_output.broadcast(data);
```

The idea for the input port is similar with one difference. An equivalent to the `broadcast()` method does not exist on the input side. So if the actor knows that it is going to read from only one channel on the input, it uses the `get()` method of the input port, with the index 0 as an argument.

```
data = _input.get(0);
```

If it wants to read from all the channels on the input port, then it iterates through the list of channels and uses the index to read tokens from them one after the other as in:

```
public class PNRedirect extends AtomicActor{
    public PNRedirect(CompositeActor container, String name) {
        super(container, name);
        _input = new IOPort(this, "input", true, false);
        _output = new IOPort(this, "output", false, true);
        new Parameter(this, "Initial value", new IntToken(0));
    }

    public void fire() {
        Parameter param = (Parameter)getAttribute("Initial value"));
        IntToken init = (IntToken)param.getToken();
        _output.broadcast(init);
        while (true) {
            _output.broadcast(_input.get(0));
        }
    }

    private IOPort _input;
    private IOPort _output;
}
```

Figure 3.4: A process - PNRedirect - that inserts an element at the head of its output stream and redirects elements from its input to its output.

```
for (int i=0; i<_input.getWidth(); i++) {
    _output.broadcast(_input.get(i));
}
```

Each instance of the above process runs independently of the others and the only place where it interacts with them in any way is through the `get()`, `send()` or `broadcast()` methods of the ports. There is no sharing of information using shared variables or in any other way with other processes. This is a concrete example of the modularity in the implementation of Ptolemy II.

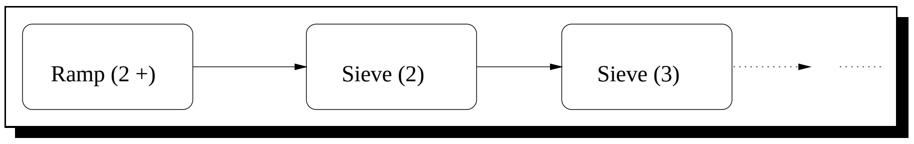

Figure 3.5: Graphical representation of a process network that models the Sieve of Eratosthenes algorithm for generating primes.

3.2 Mutating Graphs

Another slightly more complicated example is the Sieve of Eratosthenes, which detects prime numbers. The network in this example (fig. 3.5) is not a static graph as in the previous example. The graph here is a mutating network and grows over time.

The `Ramp`, with code shown in fig. 3.6 produces a stream of increasing integers. The first and the last token (if any) produced by the ramp are both parameters of the actor. For this example, we choose the parameter as 2, the smallest prime number.

Each sieve process corresponds to a distinct prime number. The first sieve (connected to the ramp) corresponds to the smallest prime, 2. Each sieve reads an integer token from the input channel and checks whether the integer in it is divisible by the prime number corresponding to the sieve. If it is, then it discards the number. If the number is not divisible and the current sieve corresponds to the largest known prime, then the sieve has discovered a new prime. It creates a new instance of sieve corresponding to the new prime and connects it to its output channel.

In case it is not the largest known prime and the new integer is not divisible by the current sieve's prime, it puts the new integer, encapsulated as a token, to the output channel for the next sieve to read. The code for the `Sieve` is as shown in figs.3.7, 3.8.

The new aspect of this actor is the mutations in the form of creation of a new

```
public class Ramp extends AtomicActor {
    public Ramp(CompositeActor container, String name) {
        super(container, name);
        _output = new IOPort(this, "output", false, true);
        new Parameter(this, "Initial value", new IntToken(0));
        new Parameter(this, "Run Forever?", new BooleanToken(true));
        new Parameter(this, "Maximum value", new IntToken(100));
    }

    public void fire() {
        Parameter param = (Parameter)getAttribute("Maximum value"));
        int max = ((IntToken)param.getToken()).intValue();
        param = (Parameter)getAttribute("Run Forever?"));
        boolean ever = ((BooleanToken)param.getToken()).booleanValue();
        if(ever || _seed <= max) {
            _output.broadcast(new IntToken(_seed));
            _seed++;
        } else {
            _notdone = false;
        }
    }

    public void initialize() throws IllegalActionException {
        Parameter param = (Parameter)getAttribute("Initial value"));
        _seed = ((IntToken)param.getToken()).intValue();
    }

    public boolean postfire() {
        return _notdone;
    }

    private boolean _notdone = true;
    private int _seed;
    private IOPort _output;
}
```

Figure 3.6: A process - Ramp - generating increasing integer tokens

```
public class Sieve extends AtomicActor {
    public Sieve(CompositeActor container, String name) {
        super(container, name);
        _input = new IOPort(this, "input", true, false);
        _output = new IOPort(this, "output", false, true);
        new Parameter(this, "Prime", new IntToken(2));
    }

    public void fire() {
        boolean islargestprime = true;
        while (true) {
            int value = ((IntToken)_input.get(0)).intValue();
            if (value%_prime != 0) {
                if (islargestprime) {
                    TopologyChangeRequest m = makeMutation(value);
                    PNDirector director = (PNDirector)getDirector();
                    director.queueTopologyChangeRequest(m);
                    islargestprime = false;
                } else {
                    _output.broadcast(data);
                }
            }
        }
    }

    public void initialize() throws IllegalActionException {
        Parameter param = (Parameter)getAttribute("Prime"));
        _prime = ((IntToken)param.getToken()).intValue();
    }

    private IOPort _input;
    private IOPort _output;
    private int _prime = 2;
}
```

Figure 3.7: A process - Sieve - that detects primes using the Sieve of Eratosthenes. The part of the code except for the `makeMutation()` method is shown.

```
private TopologyChangeRequest makeMutation(final int value) {
    TopologyChangeRequest request = new TopologyChangeRequest(this) {
        public void constructEventQueue() {
            Sieve newSieve = null;
            Relation newRelation = null;
            IOPort input = null;
            IOPort output = null;
            IOPort outport = null;
            CompositeActor container =  (CompositeActor)getContainer();
            newSieve = new Sieve(container, value + "_sieve");
            Parameter param = (Parameter)getAttribute("Prime"));
            param.setToken(new IntToken(value));
            Enumeration relations = _output.linkedRelations();
            LinkedList rellist = new LinkedList();
            if (relations.hasMoreElements()) {
                Relation relation = (Relation)relations.nextElement();
                _output.unlink(relation);
                outport = (IOPort)newSieve.getPort("output");
                outport.link(relation);
                rellist.insertLast(relation);
            }
            input = (IOPort)newSieve.getPort("input");
            newRelation = container.connect(input,_output,value+"_Q");
            queueEntityAddedEvent(container, newSieve);
            Enumeration temprel = rellist.elements();
            if (temprel.hasMoreElements()) {
                relation = (Relation)temprel.nextElement();
                queuePortUnlinkedEvent(relation, _output);
                queuePortLinkedEvent(relation, outport);
            }
            queueRelationAddedEvent(container, newRelation);
            queuePortLinkedEvent(newRelation, _output);
            queuePortLinkedEvent(newRelation, input);
        }
    };
    return request;
}
```

Figure 3.8: The part of code of `Sieve` responsible for mutations

instance of the sieve actor and the dynamically changing connections. For this, a new instance of an object derived from the class `topologyChangeRequest` is created. This object implements a method, `constructEventQueue()`. The method contains all the commands needed to make the mutations and the commands needed to inform any *topology listeners*, i.e, objects observing any changes to the graph (for visualization or some other purpose).

The `constructEventQueue()` method of the `Sieve` object is shown in fig. 3.8. For clarity, the code handling exceptions has been removed. The `constructEventQueue()` method creates a new instance of `Sieve`, and sets the parameter corresponding to the prime number for this instance. Then it disconnects all the channels from its output port, attaches them to the output port of the new sieve and attaches its output port to the input port of the new sieve. In the end, the method reports all its activities to the listeners that might be attached to the graph. To inform the topology listeners of the changes to the graph, it has calls to the corresponding commands defined in the base `TopologyChangeRequest` class. Further details about this can be found in the Ptolemy II design document [11].

Chapter 4

Future Work

This chapter talks about a few extensions to the Kahn process networks and discusses their usefulness for various applications.

4.1 Timed PN

The process networks model of computation lacks a notion of time. Thus, though it can effectively model the functional behavior of systems and can test them for functional correctness, it is unable to model their real-time behavior.

For example, because of its concurrency, PN is well suited for modeling hardware architectures. In some real-time systems and embedded applications, the real-time behavior of a system is as critical as the functional correctness. Due to this, developers often have to write their applications in PN to test them for functional correctness and use some other timed model of computation, such as DE, for testing their timing behavior. An example of this can be found in [14]. Introducing a notion of time to the process networks model of computation will thus be a natural extension of PN. Similar timed models of computation have been implemented in some software packages like Pamela [15].

Adding a notion of time would make it easier for PN to interact with other timed domains in Ptolemy II and to interact with hardware.

4.2 Nondeterminism in PN

Process networks as defined by Kahn is a deterministic model of computation. Though this is sufficient and desired for modeling applications in signal processing and similar fields, it might not be the best to model things like resource contention. In many applications, non-determinism in a limited and controlled way might be desirable. Lee and Parks [4] suggest five different ways of introducing non-determinism to Kahn process networks. They are (1) polling on channels, or allowing processes to check for presence of data on the channels, (2) allowing processes to be inherently non-determinate, (3) non-deterministic merge, or allowing more than one process to write to a channel, (4) non-deterministic split, or allowing more than one process to consume data from a channel, and (5) shared memory, or allowing processes to share variables.

Some of these mechanisms can be extremely useful for modeling certain applications. Let us consider hardware architectures. Non-deterministic split and merge are useful for this. Non-deterministic splits can be used to model resource pooling, while non-deterministic merge can be used to model resource contention and interrupts. Future work on process networks could involve studying these forms of non-determinism and evaluating their expressiveness.

Bibliography

[1] The Ptolemy II project, http://ptolemy.eecs.berkeley.edu/ptolemyII.

[2] D. Lea, *Concurrent Programming in Java.* Addison-Wesley, 1997.

[3] G. Kahn, "The semantics of a simple language for parallel programming," *Info. Proc. 74*, vol. 4, pp. 471–5, 1974.

[4] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proc. of IEEE*, vol. 83, pp. 773–801, May 1995.

[5] B. A. Davey and H. A. Priestley, *Introduction to Lattices and Order.* Cambridge University Press, 1990.

[6] G. Kahn and D. B. MacQueen, "Coroutines and networks of parallel processes," *Info. Proc. 77*, vol. 7, pp. 993–8, 1977.

[7] T. M. Parks, *Bounded Scheduling of Process Networks.* PhD thesis, Univ. of California at Berkeley, 1995.

[8] The Ptolemy project, http://ptolemy.eecs.berkeley.edu.

[9] P. Laramie, R. S. Stevens, and M. Wan, "Kahn process networks in Java," ee290n class project report, Univ. of California at Berkeley, 1996.

[10] M. Fowler and K. Scott, *UML Distilled.* Addison-Wesley, 1997.

[11] E. A. Lee, "Design document of Ptolemy II," tech. rep., Univ. of California at Berkeley, 1998.

[12] S. Oaks and H. Wong, *Java Threads.* O'Reilly, 1997.

[13] G. R. Andrews, *Concurrent Programming.* The Benjamin/Cummings Publishing co., inc., 1991.

[14] P. Lieverse, P. V. Wolf, E. Deprettere, and K. Vissers, "A methodology for architecture exploration of heterogeneous systems," *to be presented at DAC*, 1999.

[15] A. van Gemund, "Performance prediction of parallel processing systems: The Pamela methodology," in *Proc. 7th ACM Int. Conf. on Supercomputing*, (Tokyo), pp. 318–327, July 1993.

[16] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proc. of the IEEE*, vol. 75, pp. 1235–45, Sept. 1987.

[17] N. A. Lynch and E. W. Stark, "A proof of the Kahn principle for input/output automata," *Information and Computation 82*, pp. 81–92, 1989.