Chapter 8 from: C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, H. Zheng "Heterogeneous Concurrent Modeling and Design in Java (Volume 3: Ptolemy II Domains)," Technical Memorandum UCB/ERL M04/17, University of California, Berkeley, CA USA 94720, June 24, 2004.

8

PN Domain

Author: *Mudit Goel*
Contributor: *Steve Neuendorffer*

8.1 Introduction

The process networks (PN) domain in Ptolemy II models a system as a network of processes that communicate with each other by passing messages through unidirectional first-in-first-out (FIFO) channels. A process blocks when trying to read from an empty channel until a message becomes available on it. This model of computation is deterministic in the sense that the sequence of values communicated on the channels is completely determined by the model. Consequently, a process network can be evaluated using a complete parallel or sequential schedule and every schedule in between, always yielding the same output results for a given input sequence.

PN is a natural model for describing signal processing systems where infinite streams of data samples are incrementally transformed by a collection of processes executing in parallel. Embedded signal processing systems are good examples of such systems. They are typically designed to operate indefinitely with limited resources. This behavior is naturally described as a process network that runs forever but with bounded buffering on the communication channels whenever possible.[1]

PN can also be used to model concurrency in the various hardware components of an embedded system. The original process networks model of computation can model the functional behavior of these systems and test them for their functional correctness, but it cannot directly model their real-time behavior. To address the involvement of time, we have extended the PN model such that it can include the notion of time.

Some systems might display adaptive behavior like migrating code, agents, and arrivals and departures of processes. To support this adaptive behavior, we provide a mutation mechanism that supports addition, deletion, and changing of processes and channels. With untimed PN, this might display non-

1. In general, bounded buffers cannot be ensured for an arbitrary process network. An important part of the design of a process network concerns showing that the buffers are, in fact, bounded. Synchronous dataflow models are an important type of process network which always have bounded buffers.

determinism, while with timed-PN, it becomes deterministic.

The PN model of computation is a superset of the synchronous dataflow model of computation (see the SDF Domain chapter). Consequently, any SDF actor can be used within the PN domain. Similarly any domain-polymorphic actor can be used in the PN domain. However, the execution of the model is very different from SDF, since a separate process is created for each actor. These processes are implemented as Java threads [113].

8.2 Using PN

There are two issues to be dealt with in the PN domain:

- Deadlock in feedback loops
- Designing actors

8.2.1 Deadlock in Feedback Loops

Feedback loops must be handled in much the same way as in the SDF domain. One of the actors in the feedback loop must create a number of tokens in its feedback loop in order to break the data dependency. Just like in the SDF domain, the SampleDelay actor can be used for this purpose. Remember, however, that the PN domain does not (and cannot) statically analyze the model to determine the size of the delay necessary in the feedback loop. It is up to the designer of the model to specify the correct amount of delay.

8.2.2 Designing Actors

Because of the way the PN domain is implemented, it is not possible for an actor to check if data is present at an input port. The hasToken() method always returns true indicating that a token is present, and if a token is not actually present, then the get() method will block until one becomes available. This allows models to execute deterministically. However, actors that take inputs from more than one input can often be difficult to write. The common way of creating such an actor is similar to the way the Select actor works. A control input is read first, and the data from that port determines which input port to read from.

8.3 Properties of the PN domain

Two important properties of the PN domain implemented in Ptolemy II are that processes communicate asynchronously (by ordered queues) and that the memory used in the communication is bounded whenever possible. The PN domain in Ptolemy II can be used with or without a notion of time.

8.3.1 Asynchronous Communication

Kahn and MacQueen [63][64] describe a model of computation where processes are connected by communication channels to form a network. Processes produce data elements or *tokens* and send them along a unidirectional communication channel where they are stored in a FIFO queue until the destination process consumes them. This is a form of asynchronous communication between processes. Communication channels are the *only* method processes may use to exchange information. A set of processes that communicate through a network of FIFO queues defines a *program*.

Kahn and MacQueen require that execution of a process be suspended when it attempts to get data from an empty input channel (*blocking reads*). Hence, a process may not poll a channel for presence or absence of data. At any given point, a process is either doing some computation (enabled) or it is blocked waiting for data (*read blocked*) on exactly one of its input channels; it cannot wait for data from more than one channel simultaneously. Systems that obey this model are determinate; the history of tokens produced on the communication channels does not depend on the execution order. Therefore, the results produced by executing a program are not affected by the scheduling of the various processes.

In case all the processes in a model are blocked while trying to read from some channel, then we have a *real deadlock*; none of the processes can proceed. Real deadlock is a program state that happens irrespective of the schedule chosen for the processes in a model. This characteristic is guaranteed by the determinacy property of process networks.

8.3.2 Bounded Memory Execution

The high level of concurrency in process networks makes it an ideal match for embedded system software and for modeling hardware implementations. A characteristic of these embedded applications and hardware processes, is that they are intended to run indefinitely with a limited amount of memory. One problem with directly implementing the Kahn-MacQueen semantics is that bounded memory execution of a process network is not guaranteed, even if it is possible. Hence, bounded memory execution of process networks becomes crucial for its usefulness for hardware and embedded software.

Parks [117] addresses this aspect of process networks and provides an algorithm to make a process network application execute in bounded memory whenever possible. He provides an implementation of the Kahn-MacQueen semantics using *blocking writes* that assigns a fixed capacity to each FIFO channel and forces processes to block temporarily if a channel is full. Thus a process has now three states: *running* (*executing*), *read blocked*, or *write blocked* and a process may not poll a channel for either data or room.

In addition to the real deadlock described above, the introduction of a blocking write operation can cause an *artificial deadlock* of the process network. In this situation, all the processes in a model are blocked and at least one process is blocked on a write. However unlike after a real deadlock, a program can continue after artificial deadlock by increasing the capacity of the channels on which processes are write blocked. In particular, Parks chooses to increase only the capacity of the channel with the smallest capacity among the channels on which processes are write blocked. This algorithm minimizes overall required memory in the channels and is used in the PN domain to handled artificial deadlock.

8.3.3 Time

In real-time systems and embedded applications, the real time behavior of a system is as important as the functional correctness. Process networks can be used to describe the functional properties of a system, but cannot describe temporal properties since the basic model lacks the notion of time. One solution is to use some other timed model of computation, such as DE, for describing temporal properties. Another solution is to extend the process networks model of computation with a notion of time, as we have done in Ptolemy II. This extension is based on the Pamela model [44], which was originally developed for modeling the performance of parallel systems using Dykstra's semaphores.

In the timed PN domain, time is global. All processes in a model share the same time, which is referred to as the *current time* or *model time*. A process can explicitly wait for time to advance, by *delaying* itself for some fixed amount of time. After being suspended for the specified amount of time,

the process wakes up and continues to execute. If the process delays itself for zero time then the process simply continues to execute.

In the timed PN domain, time changes only at specific moments and never during the execution of a process. The time observed by a process can only advance when it is in one of the following two states:

1. The process is delayed and is explicitly waiting for time to advance (*delay block*).
2. The process is waiting for data to arrive on one of its input channels (*read block*).

When all the processes in a program are in one of these two states, then the program is in a state of *timed deadlock*. The fact that at least one process is delayed, distinguishes timed deadlock from other deadlocks. When timed deadlock is detected, the current time is advanced until at least one process can wake up from a delay block and the model continues executing.

8.3.4 Mutations

The PN domain tolerates mutations, which are run-time changes in the model structure. Normally, mutations are realized as *change requests* queued with the model. In PN there is no determinate point where mutations can occur other than a real deadlock. However, being able to perform mutations at this point is unlikely as a real deadlock might never occur. For example, a model with even one non-terminating source never experiences a real deadlock. Therefore mutations cannot be performed at determinate points since the processes in the network are not synchronized. Executing mutations at arbitrary times introduces non-determinism in PN, since the state of the processes is unknown.

In timed PN, however, the presence of timed deadlock provides a regular point at which the state of execution can be determined. This means that mutations in timed PN can be made deterministically. Implementation details are presented in the next section.

8.4 The PN Software Architecture

The PN domain kernel is realized in the package ptolemy.domains.pn.kernel. The structure diagram of the package is shown in figure 8.1.

8.4.1 PNDirector

This class extends the CompositeProcessDirector base class to add Kahn process networks (PN) semantics. This director does not support mutations or a notion of time. It provides only a mechanism to perform blocking reads and writes using bounded memory execution whenever possible.

This director is capable of handling both real and artificial deadlocks. Artificial deadlock is resolved as soon as it arises using Parks' algorithm as explained in section 8.3.2. Real deadlock, however, cannot be handled locally and must rely on the external environment to provide more data for execution to continue.

PNDirector has a parameter called *initialQueueCapacity* that sets the initial capacities of the queues in all the receivers created in the PN domain. Another parameter, *maximumQueueCapacity*, sets the upper bound on the queue capacities.

8.4.2 TimedPNDirector

TimedPNDirector extends the PNDirector to introduce a notion of global time to the model. It also

provides for deterministic execution of mutations. Mutations are performed at the earliest timed-deadlock that occurs after they are queued. Since occurrence of timed-deadlock is deterministic, performing mutations at this point makes mutations deterministic.

8.4.3 PNQueueReceiver

The PNQueueReceiver implements the ProcessReceiver interface and contains a FIFO queue to represent a process network communication channel. These receivers are also responsible for implementing the blocking reads and blocking writes through the get() and put() methods.

When the get() method is called, the receiver first checks whether its FIFO queue has any token. If not, then it reports to the director that the reading thread is blocked waiting for data. It also sets an internal flag to indicate that a thread is read blocked. Then the reading thread is suspended until some other thread puts a token into the FIFO queue. At this point, the flag of the receiver is reset to false, the director is notified that a process has unblocked, the reading process retrieves the first token from the FIFO queue and execution continues.

The put() method of the receiver works similarly by first checking whether the FIFO queue is full to capacity. If so, it reports to the director that the writing thread is blocked waiting for space in the queue. It also sets an internal flag to indicate that a thread is write blocked. The writing thread blocks until some other thread gets a token from the FIFO queue, or the size of the queue is increased by the director because the model reached an artificial deadlock. In either case, the director is notified that a writing process unblocks and the internal flag is reset. The writing thread is waked up and its token is placed into the receiver.

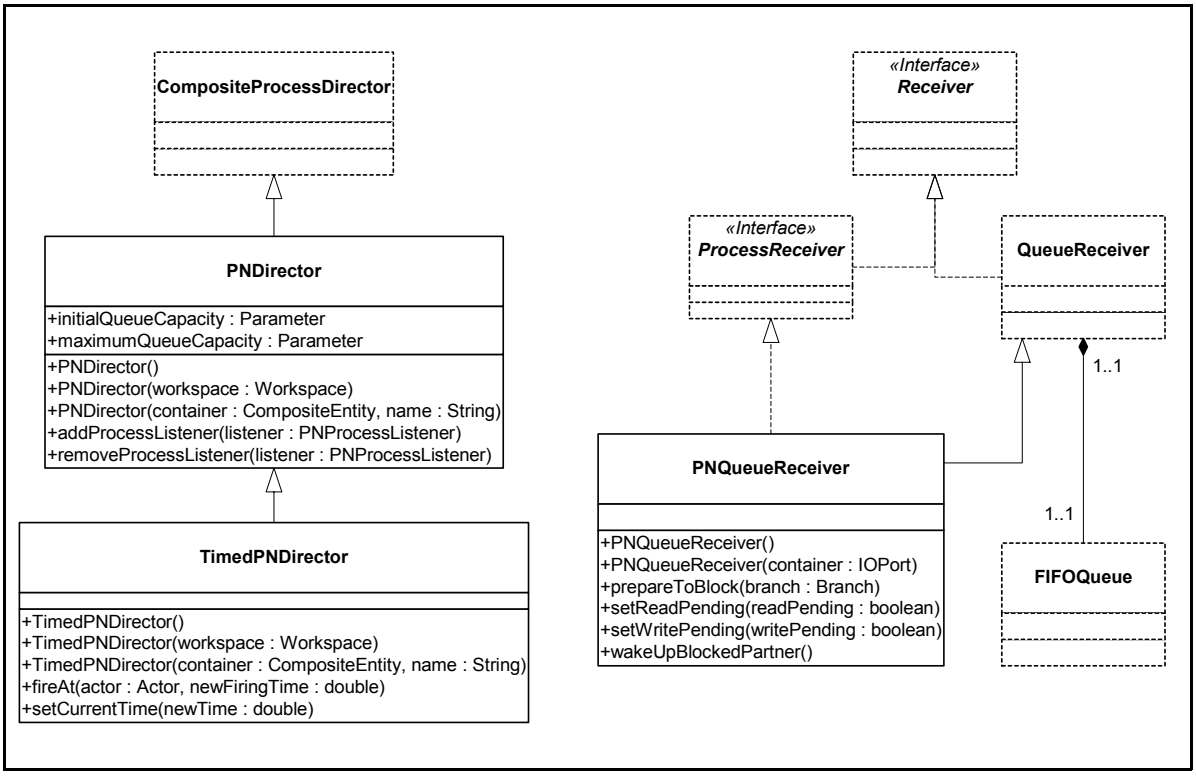

FIGURE 8.1. Static structure of the PN kernel.

8.4.4 Handling Deadlock

Every time an actor in PN blocks, the count of blocked actors is incremented. If the total number of actors blocked or paused by user request 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 doubled. This implements the bounded memory execution as suggested by [117]. If a real deadlock is detected, then the fire() method of the director returns, allowing a containing model to present more data to the inputs of the process network.

8.4.5 Finite Iterations

An important aspect of Ptolemy II is that the firing of an actor, or an entire model is guaranteed to complete. In the process domains the end of a firing occurs when deadlock is reached. The deadlock can be real or timed deadlock. However, in a process network real deadlock may never actually happen. In this case, in order to manually stop execution or to execute mutations there needs to be a way to halt all the executing threads in the network. This is handled by the stopFire() method of the executable interface. The process director implements this method to set a flag in each process which causes the process to pause. Note that as with most domains, it is not possible to simply call the wrapup() method of the process director, since the fire method has not yet returned.