Chapter 10. PN domain

Authors: *Thomas M. Parks*

Other Contributors: *Brian Evans*

10.1 Introduction

The Process Network (PN) domain is an implementation of the theory presented in Thomas M. Parks' thesis [Par95]. The PN domain includes the Synchronous Dataflow (SDF), Boolean Dataflow (BDF), and Dynamic Dataflow (DDF) domains as subdomains. This hierarchical relationship among the domains is shown in the *User's Manual* in Figure 1-2. The model of computation for each domain is a strict subset of the model for the domain that contains it.

The nodes of a program graph, which correspond to processes or dataflow actors, are implemented in Ptolemy by objects derived from the class `Star`. The firing function of a dataflow actor is implemented by the `run` method of `Star`. The edges of the program graph, which correspond to communication channels, are implemented by the class `Geodesic`. A `Geodesic` is a first-in first-out (FIFO) queue that is accessed by the `put` and `get` methods. The connections between stars and geodesics are implemented by the class `PortHole`. Each `PortHole` has an internal buffer. The methods `sendData` and `receiveData` transfer data between this buffer and a `Geodesic` using the `put` and `get` methods.

Several existing domains in Ptolemy, such as SDF and BDF, implement dataflow process networks by scheduling the firings of dataflow actors. The firing of a dataflow actor is implemented as a function call to the `run` method of a `Star` object. A scheduler executes the system as a sequence of function calls. Thus, the repeated actor firings that make up a dataflow process are interleaved with the actor firings of other dataflow processes. Before invoking the `run` method of a `Star`, the scheduler must ensure that enough data is available to satisfy the actor's firing rules. This makes it necessary for a `Star` object to inform the scheduler of the number of tokens it requires from its inputs. With this information, a scheduler can guarantee that an actor will not attempt to read from an empty channel.

By contrast, the PN domain creates a separate thread of execution for each node in the program graph. Threads are sometimes called *lightweight processes*. Modern operating systems, such as Unix, support the simultaneous execution of multiple processes. There need not be any actual parallelism. The operating system can interleave the execution of the processes. Within a single process, there can be multiple lightweight processes or threads, so there are two levels of multi-threading. Threads share a single address space, that of the parent process, allowing them to communicate through simple variables. There is no need for more complex, heavyweight inter-process communication mechanisms such as pipes.

Synchronization mechanisms are available to ensure that threads have exclusive access to shared data and cannot interfere with one another to corrupt shared data structures. Monitors and condition variables are available to synchronize the execution of threads. A monitor is an object that can be locked and unlocked. Only one thread may hold the lock on a monitor. If a thread attempts to lock a monitor that is already locked by another thread, it is suspended until the monitor is unlocked. At that point it wakes up and tries again to lock the monitor. Condition variables allow threads to send signals to each other. Condition variables must be used in conjunction with a monitor; a thread must lock the associated monitor before using a condition variable.

The scheduler in the PN domain creates a thread for each node in the program graph. Each thread implements a dataflow process by repeatedly invoking the `run` method of a `Star` object. The scheduler itself does very little work, leaving the operating system to interleave the execution of threads. The `put` and `get` methods of the class `Geodesic` have been re-implemented using monitors and condition variables so that a thread attempting to read from an empty channel is automatically suspended, and threads automatically wake up when data becomes available.

The classes `PtThread`, `PtGate`, and `PtCondition` define the interfaces for threads, monitors, and condition variables in Ptolemy. Different implementations can be used as long as they conform to the interfaces defined in these base classes. At different points in the development of the PN domain, we experimented with implementations based on Sun's Lightweight Process library, AWESIME (A Widely Extensible Simulation Environment) by Dirk Grunwald [Gru91}, and Solaris threads [Pow91,Eyk92,Kha92,Kle92a,Kle92b,Ste92,Sun94]. The current implementation is based on a POSIX thread library by Frank Mueller [Mue92,Mue93,Gie93,Mue95]. This library, which runs on several platforms, is based on Draft 6 of the POSIX standard. Parts of our implementation will need to be updated to be compliant with the final POSIX thread standard.

By choosing the POSIX standard, we improve the portability of our code. Sun and Hewlett Packard already include an implementation of POSIX threads in their operating systems, Solaris 2.5 and HPUX 10. Having threads built into the kernel of the operating system, as opposed to a user library implementation, offers the opportunity for automatic parallelization on multiprocessor workstations. Thus, the same program runs properly on uniprocessor workstations and multiprocessor workstations without needing to be recompiled. This is important because it would be impractical to maintain different binary executables of Ptolemy for each workstation configuration.

10.2 Processes

Figure 10-1 shows the class derivation hierarchy for the classes that implement the processes of Kahn process networks. The abstract base class `PtThread` defines the interface for threads in Ptolemy. The class `PosixThread` provides an implementation based on the POSIX thread standard. Other implementations using AWESIME [Gru91] or Solaris [Pow91] are possible. The class `PNThread` is a `typedef` that determines which implementation is used in the PN domain. Changing the underlying implementation simply requires changing this `typedef`. The class `DataFlowProcess`, which is derived from `PNThread`, implements a dataflow process. The `Star` object associated with an instance of `DataFlowProcess` is activated repeatedly, just as a dataflow actor is fired repeatedly to form a process.

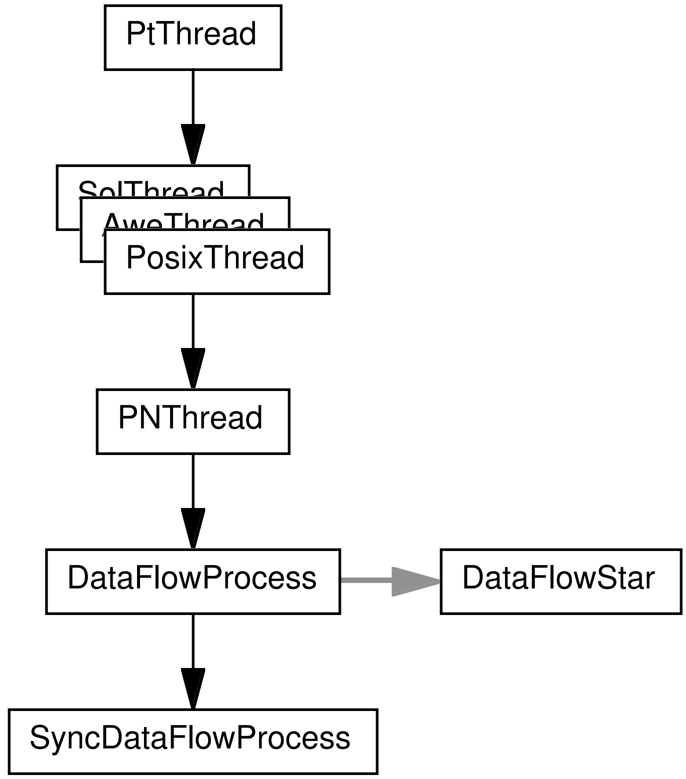

FIGURE 10-1: The class derivation hierarchy for threads. `PtThread` is an abstract base class with several possible implementations. Each `DataFlowProcess` refers to a `DataFlowStar`.

10.2.1 The PtThread Class

`PtThread` is an abstract base class that defines the interface for all thread objects in Ptolemy. Because it has pure virtual methods, it is not possible to create an instance of `PtThread`. All of the methods are virtual so that objects can be referred to as a generic `PtThread`, but with the correct implementation-specific functionality.

The class `PtThread` has three public methods.

```
virtual void initialize() = 0;
```

This method initializes the thread and causes it to begin execution.

`virtual void runAll();`

This method causes all threads to begin (or continue) execution.

`virtual void terminate() = 0;`

This method causes execution of the thread to terminate.

The class `PtThread` has one protected method.

`virtual void run() = 0;`

This method defines the functionality of the thread. It is invoked when the thread begins execution.

10.2.2 The PosixThread Class

The class `PosixThread` provides an implementation for the interface defined by `PtThread`. It does not implement the pure virtual method `run`, so it is not possible to create an instance of `PosixThread`. This class adds one protected method, and one protected data member to those already defined in `PtThread`.

`static void* runThis(PosixThread*);`

This static method invokes the `run` method of the referenced thread. This provides a C interface that can be used by the POSIX thread library.

`pthread_t thread;`

A handle for the POSIX thread associated with the `PosixThread` object.

`pthread_attr_t attributes;`

A handle for the attributes associated with the POSIX thread.

`int detach;`

A flag to set the detached state of the POSIX thread.

The `initialize` method shown below initializes attributes, then creates a thread. The thread is created in a non-detached state, which makes it possible to later synchronize with the thread as it terminates. The controlling thread (usually the main thread) invokes the `terminate` method of a thread and waits for it to terminate. The priority and scheduling policy for the thread are inherited from the thread that creates it, usually the main thread. A function pointer to the `runThis` method and the `this` pointer, which points to the current `PosixThread` object, are passed as arguments to the `pthread_create` function. This creates a thread that executes `runThis`, and passes `this` as an argument to `runThis`. Thus, the `run` method of the `PosixThread` object is the main function of the thread that is created. The `runThis` method is required because it would not be good practice to pass a function pointer to the `run` method as an argument to `pthread_create`. Although the `run` method has an implicit `this` pointer argument by virtue of the fact that it is a class method, this is really an implementation detail of the C++ compiler. By using the `runThis` method, we make the pointer argument explicit and avoid any dependencies on a particular compiler implementation.

```
void PosixThread::initialize()
{
```

```
        // Initialize attributes.
        pthread_attr_init(&attributes);

        // Detached threads free up their resources as soon
        // as they exit; non-detached threads can be joined.
        detach = 0;
        pthread_attr_setdetachstate(&attributes, &detach);

        // New threads inherit their priority and scheduling policy
        // from the current thread.
        pthread_attr_setinheritsched(&attributes,
                                     PTHREAD_INHERIT_SCHED);

        // Set the stack size to something reasonably large. (32K)
        pthread_attr_setstacksize(&attributes, 0x8000);

        // Create a thread.
        pthread_create(&thread, &attributes,
                       (pthread_func_t)runThis, this);
        // Discard temporary attribute object.
        pthread_attr_destroy(&attributes);
}
```

The `runAll` method, which is shown below, allows all threads to run by lowering the priority of the main thread. If execution of the threads ever stops, control returns to the main thread and its priority is raised again to prevent other threads from continuing.

```
// Start or continue the running of all threads.
void PosixThread::runAll()
{
        // Lower the priority to let other threads run. When control
        // returns, restore the priority of this thread to prevent
        // others from running.

        pthread_attr_t attributes;
        pthread_attr_init(&attributes);
        pthread_getschedattr(mainThread, &attributes);

        pthread_attr_setprio(&attributes, minPriority);
        pthread_setschedattr(mainThread, attributes);

        pthread_attr_setprio(&attributes, maxPriority);
        pthread_setschedattr(mainThread, attributes);

        pthread_attr_destroy(&attributes);
}
```

The `terminate` method shown below causes the thread to terminate before deleting the `PosixThread` object. First it requests that the thread associated with the `PosixThread` object terminate, using the `pthread_cancel` function. Then the current thread is suspended by `pthread_join` to give the cancelled thread an opportunity to terminate. Once termination of that thread is complete, the current thread resumes and deallocates resources used by the terminated thread by calling `pthread_detach`. Thus one thread can cause another to terminate by invoking the `terminate` method of that thread.

```
void PosixThread::terminate()
{
      // Force the thread to terminate if it has not already done so.
      // Is it safe to do this to a thread that has already
      // terminated?
      pthread_cancel(thread);

      // Now wait.
      pthread_join(thread, NULL);
      pthread_detach(&thread);
}
```

10.2.3 The DataFlowProcess Class

The class `DataFlowProcess` is derived from `PosixThread`. It implements the *map* higher-order function (see the PN Domain chapter in the *User's Manua*l). A `DataFlowStar` is associated with each `DataFlowProcess` object.

`DataFlowStar& star;`
: This protected data member refers to the dataflow star associated with the `DataFlowProcess` object.

The constructor, shown below, initializes the `star` member to establish the association between the thread and the star.

```
DataFlowProcess(DataFlowStar& s)
      : star(s) {}
```

The `run` method, shown below, is defined to repeatedly invoke the `run` method of the star associated with the thread, just as the *map* function forms a process from repeated firings of a dataflow actor. Some dataflow stars in the BDF domain can operate with static scheduling or dynamic, run-time scheduling. Under static scheduling, a BDF star assumes that tokens are available on control inputs and appropriate data inputs. This requires that the scheduler be aware of the values of control tokens and the data ports that depend on these values. Because our scheduler has no such special knowledge, these stars must be properly configured for dynamic, multi-threaded execution in the PN domain. Stars in the BDF domain that have been configured for dynamic execution, and stars in the DDF domain dynamically inform the scheduler of data-dependent firing rules by designating a particular input `PortHole` with the `waitPort` method. Data must be retrieved from the designated input before invoking the star's `run` method. The star's `run` method is invoked repeatedly, until it indicates an error by returning `FALSE`.

```
void DataFlowProcess::run()
{
      // Configure the star for dynamic execution.
      star.setDynamicExecution(TRUE);
```

```
        // Fire the Star ad infinitum.
        do
        {
              if (star.waitPort()) star.waitPort()->receiveData();
        } while(star.run());
}
```

10.3 Communication Channels

Figure 10-2 shows the class derivation hierarchy for the classes that implement the

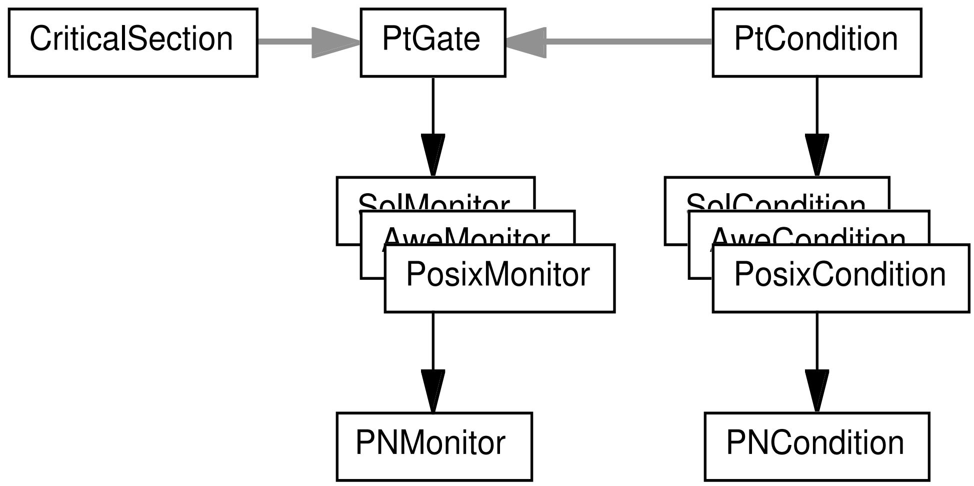

FIGURE 10-2: The class derivation hierarchy for monitors and condition variables. `PtGate` and `PtCondition` are abstract base classes, each with several possible implementations. Each `CriticalSection` and `PtCondition` refers to a `PtGate`.

communication channels of Kahn process networks. The classes that implement the communication channels provide the synchronization necessary to enforce the blocking read semantics of Kahn process networks. The classes `PtGate`, `PosixMonitor` and `CriticalSection` provide a mutual exclusion mechanism. The classes `PtCondition` and `PosixCondition` provide a synchronization mechanism. The class `PNGeodesic` uses these classes to implement a communication channel that enforces the blocking read operations of Kahn process networks and the blocking write operations required for bounded scheduling.

The abstract base class `PtGate` defines the interface for mutual exclusion objects in Ptolemy. The class `PosixMonitor` provides an implementation of `PtGate` based on the POSIX thread standard. Other implementations are possible. The class `PNMonitor` is a `typedef` that determines which implementation is used in the PN domain. Changing the underlying implementation simply requires changing this `typedef`.

The abstract base class `PtCondition` defines the interface for condition variables in Ptolemy. The class `PosixCondition` provides an implementation based on the POSIX thread standard. Other implementations are possible. The class `PNCondition` is a `typedef` that determines which implementation is used in the PN domain. Changing the underlying implementation simply requires changing this `typedef`.

The class `CriticalSection` provides a convenient method for manipulating

PtGate objects, preventing some common programming errors. The class PNGeodesic uses all of these classes to implement a communication channel.

10.3.1 PtGate

A PtGate can be locked and unlocked, but only one thread can hold the lock. Thus if a thread attempts to lock a PtGate that is already locked by another thread, it is suspended until the lock is released.

`virtual void lock() = 0;`
: This protected method locks the PtGate object for exclusive use by one thread.

`virtual void unlock() = 0;`
: This protected method releases the lock on the PtGate object.

10.3.2 PosixMonitor

The class PosixMonitor provides an implementation for the interface defined by PtGate. It has a single protected data member.

`pthread_mutex_t thread;`
: A handle for the POSIX monitor associated with the PosixMonitor object.

The implementations of the lock and unlock methods are shown below.

```
void PosixMonitor::lock()
{
      pthread_mutex_lock(&mutex);
}

void PosixMonitor::unlock()
{
      pthread_mutex_unlock(&mutex);
}
```

10.3.3 CriticalSection

The class CriticalSection provides a convenient mechanism for locking and unlocking PtGate objects. Its constructor, shown below, locks the gate. Its destructor, also shown below, unlocks the gate. To protect a section of code, simply create a new scope and declare an instance of CriticalSection. The PtGate is locked as soon as the CriticalSection is constructed. When execution of the code exits scope, the CriticalSection destructor is automatically invoked, unlocking the PtGate and preventing errors caused by forgetting to unlock it. Examples of this usage are shown in Section 10.3.6. Because only one thread can hold the lock on a PtGate, only one section of code guarded in this way can be active at a given time.

```
CriticalSection(PtGate* g) : mutex(g)
{
      if (mutex) mutex->lock();
```

```
}

~CriticalSection()
{
      if (mutex) mutex->unlock();
}
```

10.3.4 PtCondition

The class `PtCondition` defines the interface for condition variables in Ptolemy. A `PtCondition` provides synchronization through the `wait` and `notify` methods. A condition variable can be used only when executing code within a critical section (i.e., when a `PtGate` is locked).

`PtGate& mon;`
: This data member refers to the gate associated with the `PtCondition` object.

`virtual void wait() = 0;`
: This method suspends execution of the current thread until notification is received. The associated gate is unlocked before execution is suspended. Once notification is received, the lock on the gate is automatically reacquired before execution resumes.

`virtual void notify() = 0;`
: This method sends notification to one waiting thread. If multiple threads are waiting for notification, only one is activated.

`virtual void notifyAll() = 0;`
: This method sends notification to all waiting threads. If multiple threads are waiting for notification, all of them are activated. Once activated, all of the threads attempt to reacquire the lock on the gate, but only one of them succeeds. The others are suspended again until they can acquire the lock on the gate.

10.3.5 PosixCondition

The class `PosixCondition` provides an implementation for the interface defined by `PtCondition`. The implementations of the `wait`, `notify` and `notifyAll` methods are shown below.

```
void PosixCondition::wait()
{
      // Guarantee that the mutex will not remain locked
      // by a cancelled thread.
      pthread_cleanup_push((void(*)(void*))pthread_mutex_unlock,
            &mutex);

      pthread_cond_wait(&condition, &mutex);

      // Remove cleanup handler, but do not execute.
      pthread_cleanup_pop(FALSE);
```

```
}

void PosixCondition::notify()
{
      pthread_cond_signal(&condition);
}

void PosixCondition::notifyAll()
{
      pthread_cond_broadcast(&condition);
}
```

10.3.6 PNGeodesic

The class `PNGeodesic`, which is derived from the class `Geodesic` defined in the Ptolemy kernel, implements the communication channels for the PN domain. In conjunction with the `PtGate` member provided in the base class `Geodesic`, two condition variables provide the necessary synchronization for blocking read and blocking write operations.

`PtCondition* notEmpty;`
: This data member points to a condition variable used for blocking read operations when the channel is empty.

`PtCondition* notFull;`
: This data member points to a condition variable used for blocking write operations when the channel is full.

`int cap;`
: This data member represents the capacity of the communication channel and determines when it is full.

`static int numFull;`
: This static data member records the number of full geodesics in the system.

The `slowGet` method, shown in below, implements the get operation for communication channels. The entire method executes within a critical section to ensure consistency of the object's data members. If the buffer is empty, then the thread that invoked `slowGet` is suspended until notification is received on `notEmpty`. Data is retrieved from the buffer, and if it is not full notification is sent on `notFull` to any other thread that may have been waiting.

```
Particle* PNGeodesic::slowGet()
{
      // Avoid entering the gate more than once.
      CriticalSection region(gate);
      while (sz < 1 && notEmpty) notEmpty->wait();
      sz--;
      Particle* p = pstack.get();
      if (sz < cap && notFull) notFull->notifyAll();
      return p;
}
```

The `slowPut` method, shown below, implements the put operation for communication channels. The entire method executes within a critical section to ensure consistency of the object's data members. If the buffer is full, then the thread that invoked `slowPut` is suspended until notification is received on `notFull`. Data is placed in the buffer, and notification is sent on `notEmpty` to any other thread that may have been waiting.

```
// Block when full.
// Notify when not empty.
void PNGeodesic::slowPut(Particle* p)
{
      // Avoid entering the gate more than once.
      CriticalSection region(gate);
      if (sz >= cap && notFull)
      {
            {
                  CriticalSection region(fullGate);
                  numFull++;
            }
            while (sz >= cap && notFull) notFull->wait();
            {
                  CriticalSection region(fullGate);
                  numFull--;
            }
      }
      pstack.putTail(p); sz++;
      if (notEmpty) notEmpty->notifyAll();
}
```

The `setCapacity` method, shown below, is used to adjust the capacity limit of communication channels. If the capacity is increased so that a channel is no longer full, notification is sent on `notFull` to any thread that may have been waiting.

```
void PNGeodesic::setCapacity(int c)
{
      CriticalSection region(gate);
      cap = c;
      if (sz < cap && notFull) notFull->notifyAll();
}
```

10.4 Scheduling

Figure 10-3 shows the class derivation hierarchy for the classes that implement the

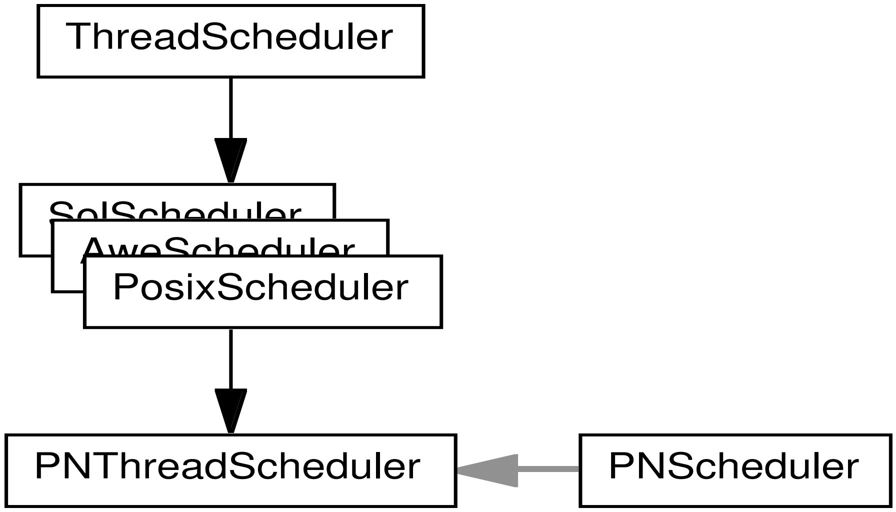

FIGURE 10-3: The class derivation hierarchy for schedulers. `ThreadList` is a container class for threads. Each `PNScheduler` uses a `ThreadList`.

dynamic scheduling of Kahn process networks. The class `ThreadList` provides mechanisms for terminating groups of threads. This class is used by `PNScheduler` to create threads for each node in the program graph. The class `SyncDataFlowProcess` implements the threads for the nodes.

10.4.1 ThreadList

The class `ThreadList` implements a container class for manipulating groups of threads. It has two public methods.

`virtual void add(PtThread*);`

This method adds a `PtThread` object to the list.

`virtual ~ThreadScheduler();`

This method terminates and deletes all threads in the list.

10.4.2 PNScheduler

The class `PNScheduler` controls the execution of a process network. Three data members support synchronization between the scheduler and the processes.

`ThreadList* threads;`

A container for the threads managed by the scheduler.

`PNMonitor* monitor;`

A monitor to guard the scheduler's condition variable.

`PNCondition* start;`

A condition variable for synchronizing with threads.

`int iteration;`
A counter for regulating the execution of the processes.

The `createThreads` method, shown below, creates one process for each node in the program graph. A `SyncDataFlowProcess` is created for each `DataFlowStar` and added to the `ThreadList` container.

```
// Create threads (dataflow processes).
void PNScheduler::createThreads()
{
	if (! galaxy()) return;
	GalStarIter nextStar(*galaxy());
	DataFlowStar* star;
	LOG_NEW; threads = new ThreadList;

	// Create Threads for all the Stars.
	while((star = (DataFlowStar*)nextStar++) != NULL)
	{
		LOG_NEW; SyncDataFlowProcess* p
			= new SyncDataFlowProcess(*star,*start,iteration);
		threads->add(p);
		p->initialize();
	}
}
```

It is often desirable to have a partial execution of a process network. The class `SyncDataFlowProcess`, which is derived from `DataFlowProcess`, supports this by synchronizing the execution of a thread with the `iteration` counter that belongs to the `PNScheduler`. The `run` methods of `PNScheduler` and `SyncDataFlowProcess` implement this synchronization. The `PNScheduler run` method, shown below, increments the `iteration` count to give every process an opportunity to run. The `SyncDataFlowProcess run` method, shown below, ensures that the number of invocations of the star's `run` method does not exceed the `iteration` count.

```
// Run (or continue) the simulation.
int PNScheduler::run()
{
	if (SimControl::haltRequested() || ! galaxy())
	{
		Error::abortRun("cannot continue");
		return FALSE;
	}

	while((currentTime < stopTime) && !SimControl::haltRequested())
	{
		// Notify all threads to continue.
		{
			CriticalSection region(start->monitor());
			iteration++;
			start->notifyAll();
		}
		PNThread::runAll();
```

```
            while (PNGeodesic::blockedOnFull() > 0
                  && !SimControl::haltRequested())
            {
                  increaseBuffers();
                  PNThread::runAll();
            }
            currentTime += schedulePeriod;
      }

      return !SimControl::haltRequested();
}

void SyncDataFlowProcess::run()
{
      int i = 0;
      // Configure the star for dynamic execution.
      star.setDynamicExecution(TRUE);

      // Fire the star ad infinitum.
      do
      {
            // Wait for notification to start.
            {
                  CriticalSection region(start.monitor());
                  while (iteration <= i) start.wait();
                  i = iteration;
            }
            if (star.waitPort()) star.waitPort()->receiveData();
      } while (star.run());
}
```

The `increaseBuffers` method is used during the course of execution to adjust the channel capacities according to the theory presented in [Par95, ch. 4]. Each time execution stops, the program graph is examined for full channels. If there are any full channels, then the capacity of the smallest one is increased.

```
// Increase buffer capacities.
// Return number of full buffers encountered.
int PNScheduler::increaseBuffers()
{
      int fullBuffers = 0;
      PNGeodesic* smallest = NULL;

      // Increase the capacity of the smallest full geodesic.
      GalStarIter nextStar(*galaxy());
      Star* star;
      while ((star = nextStar++) != NULL)
      {
            BlockPortIter nextPort(*star);
            PortHole* port;
            while ((port = nextPort++) != NULL)
            {
                  PNGeodesic* geo = NULL;
```

```
                if (port->isItOutput() &&
                    (geo = (PNGeodesic*)port->geo()) != NULL)
                {
                    if (geo->size() >= geo->capacity())
                    {
                        fullBuffers++;
                        if (smallest == NULL ||
                            geo->capacity() <
                              smallest->capacity())
                              smallest = geo;
                    }
                }
            }
        }
        if (smallest != NULL)
            smallest->setCapacity(smallest->capacity() + 1);

        return fullBuffers;
    }
```

10.5 Programming Stars in the PN Domain

Unlike portholes in the SDF domain, the number of tokens consumed by an input or produced by an output can be dynamic in the PN domain. This is indicated with the `P_DYNAMIC` porthhole attribute.

```
    input {
        name { a }
        type { int }
        attributes { P_DYNAMIC }
    }
```

For dynamic ports, it is necessary to invoke the `receiveData` and `sendData` methods explicitly. Note that the `receiveData` method must be used to initialize outputs. For static ports, the `receiveData` and `sendData` methods are invoked implicitly and should not be used in the go method.

Because a separate thread of execution is created for each star, the `go` method of a PN star is not required to terminate. As a programmer, you are free to use infinite loops, such as `while(TRUE) { ... }` within the `go` method of your PN stars. This may be necessary if you access a porthole (requiring a blocking read) before entering the main loop of the process. In the future, such code could be placed in the star's `begin` method, but currently (as of release 0.6) the `begin` method is executed before the star's thread is created.

```
    go {
        // Read both inputs the first time.
        a.receiveData();
        b.receiveData();
        while (TRUE) {
            output.receiveData();// Initialize the output.
            if (int(a%0) < int(b%0)) {
```

```
                    output%0 = a%0;
                    output.sendData();
                    a.receiveData();
                }
                else if (int(a%0) > int(b%0)) {
                    output%0 = b%0;
                    output.sendData();
                    b.receiveData();
                }
                else {     // Remove duplicates.
                    output%0 = a%0;
                    output.sendData();
                    a.receiveData();
                    b.receiveData();
                }
        }
}
```

Instead of using an infinite loop, most PN stars rely on the `run` method of `DataFlow-Process` to repeatedly invoke the star's `go` method.