4

Process Networks and Rendezvous

Neil Smyth, John S. Davis II, Thomas Huining Feng, Mudit Goel, Edward A. Lee, Thomas M. Parks, Yang Zhao

Contents

4.1 Kahn Process Networks 132
Historical Notes: Process Networks 134
Sidebar: Process Networks and Dataflow 135
4.1.1 Concurrent Firings 136
Sidebar: Actors With Unbounded Firings 138
Sidebar: Useful Actors for PN Models 143
4.1.2 Stopping Execution of a PN Model 144
4.2 Rendezvous 145
Sidebar: Useful Actors for Rendezvous Models 146
4.2.1 Multiway Rendezvous 147
4.2.2 Conditional Rendezvous 148
4.2.3 Resource Management 150
4.3 Summary 151
Exercises 152

Dataflow models of computation are concurrent. The order of the firings is constrained only by data precedence, so the director has quite a bit of freedom to choose an ordering. There is no reason that two actors that do not depend on each other's data could not be fired simultaneously. Indeed, there is a long history of schedulers that do exactly that, as described in the sidebar on page 100. Nevertheless, the directors described in Chapter 3 fire actors one at a time. This chapter introduces two directors that fire actors concurrently. Both are similar to dataflow, semantically, but they are used for very different purposes.

The directors in this chapter execute each actor in its own thread. A **thread** is a sequential program that can potentially share variables with other threads that execute concurrently. On a multicore machine, two threads may execute in parallel on separate cores. On a single core, the instructions of each thread are arbitrarily interleaved. Understanding the interactions between threads is notoriously difficult (Lee, 2006) because of this arbitrary interleaving of instructions. The directors described in this chapter provide a way for threads to interact in more understandable and predictable ways.

One possible motivation for using the directors in this chapter is to exploit the parallelism offered by multiple cores. Every model described in the previous chapter, for example, could also be executed with a PN director, described below, and actors will fire simultaneously on multiple cores. One could expect models to execute faster than they would with an SDF director. However, this is not our experience. Our experience is that PN models almost always run slower than SDF models, regardless of the number of cores. The reason for this appears to be the cost of thread interactions associated with the mutual exclusion locks that enable Ptolemy II models to be edited while they run. In principle, this capability could be disabled, and one would then expect performance improvements. But as of this writing, no mechanism has been built in Ptolemy II to do this. Hence, improving performance is probably not an adequate reason to use the threaded directors of this chapter over SDF or DDF when SDF or DDF is suitable.

Instead, we focus in this chapter on models where concurrent firing of actors is required by the goals of the model. These examples show that these directors do in fact provide a fundamental increment in expressiveness, and not just a performance improvement.

4.1 Kahn Process Networks

A model of computation that is closely related to dataflow models is **Kahn process networks** (variously called **KPN**, **process networks** or **PN**), named after Gilles Kahn, who introduced them (Kahn, 1974). The relationship between dataflow and PN is studied in detail by Lee and Parks (1995) and Lee and Matsikoudis (2009), but the short story is quite simple. In PN, each actor executes concurrently in its own thread. That is, instead of being defined by its firing rules and firing functions, a PN actor is defined by a (typically non-terminating) program that reads data tokens from input ports and writes data tokens to output ports. All actors execute simultaneously (conceptually; whether they actually execute simultaneously or are interleaved is irrelevant to the semantics).

In the original paper, Kahn (1974) gave very elegant mathematical conditions on the actors that ensure that a network of such actors is determinate. In this, case "determinate" means that the sequence of tokens on every connection between actors is uniquely defined, and specifically is independent of how the processes are scheduled. Every legal thread scheduling yields exactly the same data streams. Thus, Kahn showed that concurrent execution was possible without nondeterminism.

Three years later, Kahn and MacQueen (1977) gave a simple, easily implemented mechanism for programs that guarantees that the mathematical conditions are met to ensure determinism. A key part of the mechanism is to perform **blocking reads** on input ports whenever a process is to read input data. Specifically, blocking reads mean that if the process chooses to access data through an input port, it issues a read request and blocks until the data become available. It cannot test the input port for the availability of data and then perform a conditional branch based on whether data are available, because such a branch would introduce schedule-dependent behavior.

Blocking reads are closely related to firing rules. Firing rules specify the tokens required to continue computing (with a new firing function). Similarly, a blocking read specifies a single token required to continue computing (by continuing execution of the process). Kahn and MacQueen showed that if every actor implements a mathematical *function* from input sequences to output sequences (meaning that for each input sequence, the output sequence is uniquely defined), then blocking reads are sufficient to ensure determinism.

When a process writes to an output port, it performs a **nonblocking write**, meaning that the write succeeds immediately and returns. The process does not block to wait for the receiving process to be ready to receive data.[1] This is exactly how writes to output ports work in dataflow MoCs as well. Thus, the only material difference between dataflow and

[1]The Rendezvous director, discussed later in this chapter, differs at exactly this point, in that with that director, a write to an output port does not succeed until the actor with the corresponding input port is ready to read.

PN is that with PN, the actor is not broken down into firing functions. It is designed as a continuously executing program. The firing of an actor does not need to be finite.

Historical Notes: Process Networks

Gilles Kahn (1946 – 2006)

The notion of concurrent processes interacting by sending messages is rooted in Conway's **coroutines** (Conway, 1963). Conway described software modules that interacted with one another as if they were performing I/O operations. In Conway's words, "When coroutines A and B are connected so that A sends items to B, B runs for a while until it encounters a read command, which means it needs something from A. The control is then transferred to A until it wants to write, whereupon control is returned to B *at the point where it left off*."

The least fixed-point semantics is due to Kahn (1974), who developed the model of processes as continuous functions on a **CPO** (complete partial order). Kahn and MacQueen (1977) defined process interactions using non-blocking writes and blocking reads as a special case of continuous functions, and developed a programming language for defining interacting processes. Their language included recursive constructs, an optional functional notation, and dynamic instantiation of processes. They gave a demand-driven execution semantics, related to the lazy evaluators of Lisp (Friedman and Wise, 1976; Morris and Henderson, 1976). Berry (1976) generalized these processes with stable functions.

The notion of unbounded lists as data structures first appeared in Landin (1965). This underlies the communication mechanism between processes in a process network. The UNIX operating system, due originally to Ritchie and Thompson (1974) includes the notion of **pipes**, which implement a limited form of process networks (pipelines only). Later, named pipes provided a more general form.

Kahn (1974) stated but did not prove what has come to be known as the **Kahn principle**, that a maximal and fair execution of process network yields the least fixed point. This was later proved by Faustini (1982) and Stark (1995).

Sidebar: Process Networks and Dataflow

Three major variants of dataflow have emerged in the literature: Dennis dataflow (Dennis, 1974), Kahn process networks (KPN) (Kahn, 1974), and dataflow synchronous languages (Benveniste et al., 1994). The first two are closely related, while the third is quite different. This chapter deals with Kahn process networks, Dennis dataflow is addressed in Chapter 3, and dataflow synchronous languages are addressed in Chapter 5.

In **Dennis dataflow**, the behavior of an actor is given by a sequence of atomic firings that are enabled by the availability of input data. KPN, by contrast, has no notion of an atomic firing. An actor is a process that executes asynchronously and concurrently with others. Dennis dataflow can be viewed as a special case of KPN (Lee and Parks, 1995) by defining a firing to be the computation that occurs between accesses to inputs. But the style in which actors and models are designed is quite different. Dennis' approach is based on an operational notion of atomic firings driven by the satisfaction of firing rules. Kahn's approach is based on a denotational notion of processes as continuous functions on infinite streams.

Dennis' approach influenced computer architecture (Arvind et al., 1991; Srini, 1986), compiler design, and concurrent programming languages (Johnston et al., 2004). Kahn's approach influenced process algebras (e.g. Broy and Stefanescu (2001)) and concurrency semantics (e.g. Brock and Ackerman (1981); Matthews (1995)). It has had practical realizations in stream languages (Stephens, 1997) and operating systems (e.g. Unix pipes). Interest in these MoCs has grown with the resurgence of parallel computing, driven by multicore architectures (Creeger, 2005). Dataflow MoCs are being explored for programming parallel machines (Thies et al., 2002), distributed systems (Lzaro Cuadrado et al., 2007; Olson and Evans, 2005; Parks and Roberts, 2003), and embedded systems (Lin et al., 2006; Jantsch and Sander, 2005). There are improved execution policies (Thies et al., 2005; Geilen and Basten, 2003; Turjan et al., 2003; Lee and Parks, 1995) and standards (Object Management Group (OMG), 2007; Hsu et al., 2004).

Lee and Matsikoudis (2009) bridge the gap between Dennis and Kahn, showing that Kahn's methods extend naturally to Dennis dataflow, embracing the notion of firing. This is done by establishing the relationship between a firing function and the Kahn process implemented as a sequence of firings of that function. A consequence of this analysis is a formal characterization of firing rules and firing functions that preserve determinacy.

Kahn and MacQueen (1977) called the processes in a PN network coroutines for an interesting reason. A **routine** or **subroutine** is a program fragment that is "called" by another program. The subroutine executes to completion before the calling fragment can continue executing. The interactions between processes in a PN model are more symmetric, in that there is no caller and callee. When a process performs a blocking read, it is in a sense invoking a routine in the upstream process that provides the data. Similarly, when it performs a write, it is in a sense invoking a routine in the downstream process to process the data. But the relationship between the producer and consumer of the data is much more symmetric than with subroutines.

When using a conventional Ptolemy II actor with the PN director (vs. a custom-written actor), the actor is automatically wrapped in an infinite loop that fires it until either the model halts (see Section 4.1.2 below) or the actor terminates (by returning false from its postfire method). When the actor accesses an input, it blocks until input is available. When it sends an output, the output token goes into a **FIFO queue** (first-in, first-out) that is (conceptually) unbounded. The tokens will be eventually delivered in order to the destination actors.

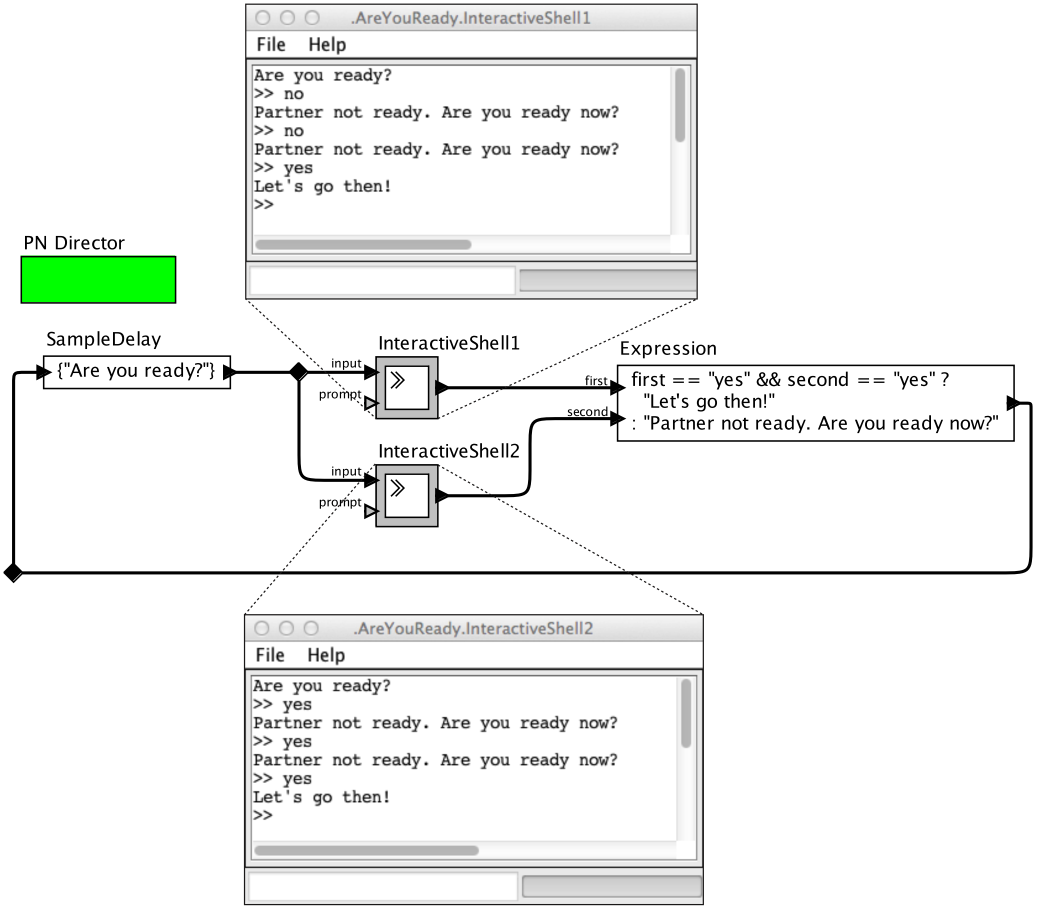
An interesting subtlety arises when using actors that behave differently depending on whether input tokens are available on the inputs. For example, the AddSubtract actor will add all available tokens on its *plus* port, and subtract all available tokens on its *minus* port. When executing under the PN director, this actor will not complete its operation until it has received one token from every input channel. This is because accesses to the inputs are blocking. When the actor asks whether a token is available on the input (using the `hasToken` method), the answer is always yes! When it then goes to read that token (using the `get` method of the input port), it will block until there actually is a token available.

Just like dataflow, PN poses challenging questions about boundedness of buffers and about deadlock. PN is expressive enough that these questions are undecidable. An elegant solution to the boundedness question is given by Parks (1995) and elaborated by Geilen and Basten (2003). The solution given by Parks is the one implemented in Ptolemy II.

4.1.1 Concurrent Firings

PN models are particularly useful when there are actors that do not return immediately when fired. The **InteractiveShell** actor, for example, found in `Sources→SequenceSources`, opens a window (like the top and bottom windows in Figure 4.1) into which a user can enter information. When the actor fires, it displays in the window whatever input it has received, followed by a prompt (which defaults to “>>”). It then waits for the user to enter a new value. When the user types a value and hits return, the actor outputs an event containing that value.

When this actor fires in response to an input, the firing will not complete until a user enters a value in the window that is opened. If a dataflow director is used (or any other director that fires actors one at a time), then the entire model will block waiting for user



```
Are you ready?
>> no
Partner not ready. Are you ready now?
>> no
Partner not ready. Are you ready now?
>> yes
Let's go then!
>>
```

```
Are you ready?
>> yes
Partner not ready. Are you ready now?
>> yes
Partner not ready. Are you ready now?
>> yes
Let's go then!
>>
```

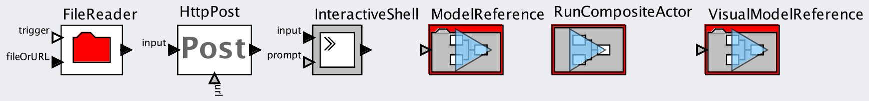
Figure 4.1: A model with two instances of InteractiveShell, each of which blocks waiting for user input. [online]

input from just one InteractiveShell. No other actor can fire. For some models, this will be a problem.

Example 4.1: Consider the model in Figure 4.1. This model emulates a dialog with two users where it is seeking concurrence from both users before proceeding with some other action. In this case, it simply asks each user "Are you ready?" It waits for one response from each user, and if both users have respond "yes," then it responds "Let's go then!" In this PN model, each instance of InteractiveShell executes in its own thread. If this model were executed instead using the SDF director, then the SDF director would fire one of these first, and would not even ask the second user the question until the first one has responded.

Sidebar: Actors With Unbounded Firings

Below are a few actors that benefit particularly from use of the PN director because they do not (necessarily) return immediately when fired:



- FileReader reads a file on the local computer or a URL (uniform resource locator from the Internet. In the latter case, the actor may block for an indeterminate amount of time while the server responds.
- HttpPost posts a record to a URL on the Internet, emulating what a browser typically does when a user fills in an online form. The actor may block for an indeterminate amount of time while the server responds.
- InteractiveShell opens a window and waits for user input.
- ModelReference executes a referenced model to completion (or indefinitely, if the model does not terminate).
- RunCompositeActor executes a contained model to completion (or indefinitely, if the model does not terminate).
- VisualModelReference opens a Vergil window to display a referenced model and executes the model to completion (or indefinitely, if the model does not terminate).

In the previous example, the InteractiveShell actor interacts with the world outside the PN model. It performs I/O, and in particular, it blocks while performing the I/O, waiting for actions from the outside world. Another example that has this property is considered in Exercise 5, which considers actors that collect data from sensors.

As explained above, a key property of Kahn process networks is that models are deterministic. It may seem odd to assert that the model in Figure 4.1 is deterministic, however. Clearly, the sequence of values generated by the Expression actor, for example, differs from run to run, because it depends on what users type into the dialog windows that open. Indeed, determinism requires that every actor implement a mathematical *function* from input sequences to output sequences. Strictly speaking, InteractiveShell does not implement such a function. In two different runs, the same sequence of inputs will not generate the same sequence of outputs, because the outputs depend on what a user types. Nevertheless, the Kahn property is valuable, because it asserts that if you fix the user behavior, then the behavior of the model is unique and well defined. If on two successive runs, two users type exactly the same sequence of responses, then the model will behave exactly the same way. The behavior of the model depends *only* on the behavior of the users, and not on the behavior of the thread scheduler.

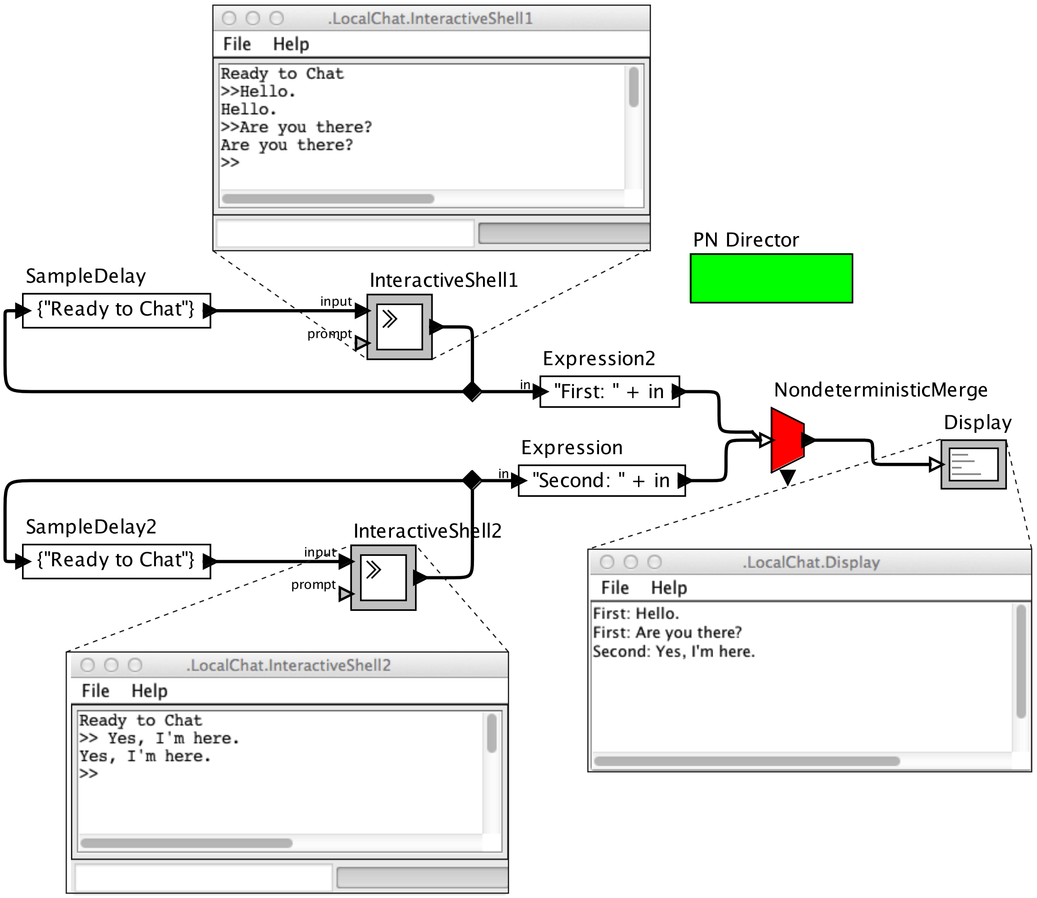
In Ptolemy II, it is possible to build process networks that are nondeterminate using a special actor called a NondeterministicMerge (see box on page 143). Such models can be quite useful.

Example 4.2: Consider the example in Figure 4.2. This model emulates a chat interaction between two users, where users can provide input in any order. In this model, users type their inputs into the windows opened by the InteractiveShell actors, and their inputs are merged and displayed by the Display actor.

In this model, it is *not* true that if on two successive runs, users type the same thing, that the results displayed will be the same. Indeed, if two users type things nearly simultaneously, then the order in which their text appears at the output of the NondeterministicMerge is undefined. Either order is correct. The order that results will depend on the thread scheduler, not on what the users have typed. This contrasts notably with the model in Figure 4.1.

The nondeterminism introduced in the previous model can be quite useful. But it comes at a cost. Models like this are much harder to test, for example. There is no single correct result of execution. For this reason, NondeterministicMerge should be used with caution, and only when it is really needed. If a model can be built using deterministic mechanisms, then it should be built using deterministic mechanisms.

Although the previous example emulates a chat session between two clients, it is not a realistic implementation of a chat application. The two interactive shell windows are opened on the same screen and run in the same process, so two distinct users cannot prac-

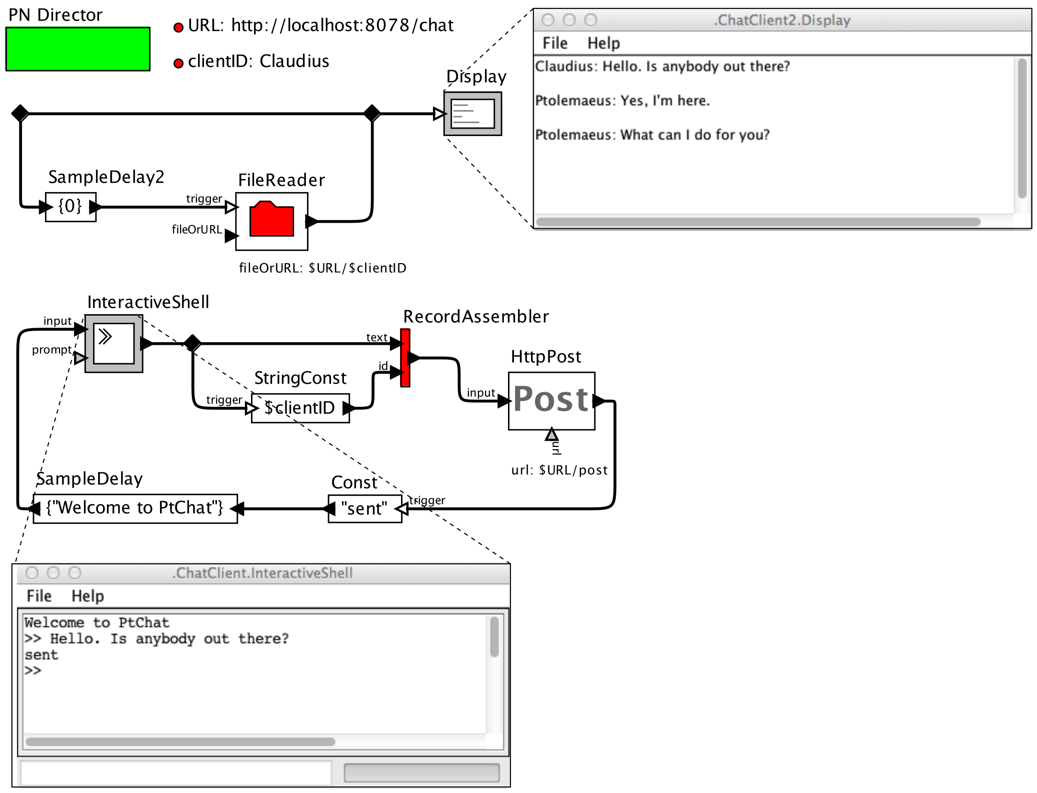


```
Ready to Chat
>>Hello.
Hello.
>>Are you there?
Are you there?
>>
```

```
First: Hello.
First: Are you there?
Second: Yes, I'm here.
```

```
Ready to Chat
>> Yes, I'm here.
Yes, I'm here.
>>
```

Figure 4.2: A model using a NondeterministicMerge to create a nondeterministic process network. [online]



```
Claudius: Hello. Is anybody out there?

Ptolemaeus: Yes, I'm here.

Ptolemaeus: What can I do for you?
```

```
Welcome to PtChat
>> Hello. Is anybody out there?
sent
>>
```

Figure 4.3: A model that implements a chat client. [online]

tically chat. PN can be used to build to build an actual chat client using only deterministic mechanisms.

Example 4.3: The model in Figure 4.3 implements a chat client. This model assumes that a server has been deployed at the URL specified by the *URL* parameter of the model. An implementation of such a server is developed in Exercise 1 of Chapter 16.

This client assumes that the server provides two interfaces. An HTTP Get request is used to retrieve chat text that will include text entered by the user of this model interleaved with text provided by any other participant in the chat session. The upper feedback loop in Figure 4.3 issues this HTTP Get using a FileReader actor. Normally, the server will not respond immediately, but rather will wait until it has chat data to provide. Hence, the FileReader actor will block, waiting for a response. Upon receiving a response, the model will display the response using the Display actor, and then issue another HTTP Get request.

This technique, where Get requests block until there are data to provide, is called **long polling**. In effect, it provides a simple form of push technology, where the server pushes data to a client when there are data to provide. Of course, it only works if the client has issued a Get request, and if the client can patiently wait for a response, rather than, say, timing out.

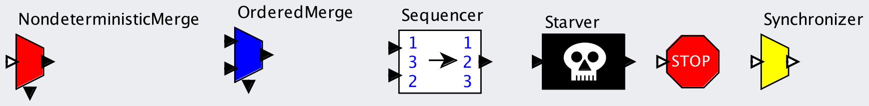
The lower feedback loop in Figure 4.3 is used for this client to supply chat text to the chat session. It uses an InteractiveShell to provide a window into which a user can enter text. When the user provides text, it assembles a record containing the text and user ID and posts the record to the server using an HttpPost actor. When the server replies, the model provides a confirmation to the user, "sent," and waits for the user to enter new text.

In the meantime, the server will normally respond to the HTTP Post by issuing a reply to all pending instances of HTTP Get. In the example execution shown in Figure 4.3, the local user, Claudius, first types "Hello. Is anybody there?" Another user, Ptolemaeus, somewhere else on the Internet, replies "Yes, I'm here." The remote user then further types "What can I do for you?" That text is "pushed" to this chat client using the long polling technique, and hence appears on the display spontaneously.

The previous example shows a use of PN to create two simultaneously executing tasks, each of which can block. This model is deterministic, in that for any sequence of outputs produced by the FileReader and InteractiveShell, all signals in the model are uniquely defined. The behavior of the model, therefore, depends *only* on user input, and not on thread scheduling, unlike the example in Figure 4.2. This form of determinism is very valuable. Among the many benefits is that models can be tested by defining input sequences and checking the responses.

Sidebar: Useful Actors for PN Models

A few actors that are particularly useful in PN models are shown below:



All of these can be found in `DomainSpecific→ProcessNetworks`, but some of them also appear in `Actors→FlowControl`.

- **NondeterministicMerge**. Merge sequences of tokens by arbitrarily interleaving their tokens onto a single stream.
- **OrderedMerge**. Merge two monotonically increasing sequences of tokens into one monotonically increasing sequence of tokens.
- **Sequencer**. Take a stream of input tokens and stream of sequence numbers and re-order the input tokens according to the sequence numbers. On each iteration, this actor reads an input token and a sequence number. The sequence numbers are integers starting with zero. If the sequence number is the next one in the sequence, then the token read from the *input* port is produced on the *output* port. Otherwise, it is saved until its sequence number is the next one in the sequence.
- **Starver**. Pass input tokens unchanged to the output until a specified number of tokens have been passed. At that point, consume and discard all further input tokens. This can be used in a feedback loop to limit the execution to a finite data set.
- **Stop**. Stop execution of a model when a true token is received on any input channel.
- **Synchronizer**. Synchronize multiple streams so that they produce tokens at the same rate. That is, when at least one new token exists on every input channel, exactly one token is consumed from each input channel, and the tokens are output on the corresponding output channels.

4.1.2 Stopping Execution of a PN Model

A PN model in Ptolemy II executes until one of the following occurs:

- All actors have terminated. An actor terminates when its postfire method returns false. Many actors in the Ptolemy II library have a *firingCountLimit* parameter (e.g., the Const actor). Setting this parameter to a positive integer will cause the actor's process to terminate after the actor has fired the specified number of times.
- All processes are blocked on reads of input ports. This is a deadlock, similar to the ones that can occur in dataflow models. A deadlock may occur due to an error in the model, or it may be deliberate. For example, the *firingCountLimit* parameter mentioned above and the Starver actor (see box on page 143) can be used to create a **starvation** condition, where actors that have not terminated are blocked on reads that will never be satisfied. Despite the pejorative tone of the words "deadlock" and "starvation," these are perfectly reasonable and useful techniques for terminating the execution of a PN model.
- The Stop actor reads a true-valued input token on its one input port. Upon reading this input, the Stop actor will coordinate with the director to stop all executing threads. Specifically, any thread that is blocked on a read or write of a port will terminate immediately, and any thread that is not blocked will terminate on its next attempt to perform a read or write. The Stop actor can be found in the `Actors →FlowControl→ExecutionControl` library.
- A buffer overflows. This occurs when the number of unconsumed tokens on a communication channel exceeds the value of the *maximumQueueCapacity* parameter of the director. Note that if you set *maximumQueueCapacity* to 0 (zero), then this will not occur until the operating system denies the Ptolemy system additional memory, which typically occurs when you have run out system memory.
- An exception occurs in some actor process. Other threads are terminated in a manner similar to what the Stop actor does.
- A user presses the stop button in Vergil. All threads are terminated in a manner similar to what the Stop actor does.

These are the only mechanisms for stopping an execution. How to use them is explored in Exercise 6.

Even so, there are some limitations. The FileReader actor, for example, used in Figure 4.3, sadly, cannot be interrupted if it is blocked on a read. Stopping the model in Figure 4.3, therefore, is difficult. If you hit the stop button, it will eventually time out, but the wait may be considerable. This appears to be a limitation in the underlying java.net.URLConnection class that is used by this actor.

4.2 Rendezvous

In the **Rendezvous** domain in Ptolemy II, like PN, each actor executes in its own thread. Unlike PN, communication between actors is by **rendezvous** rather than message passing with unbounded FIFO queues. Specifically, when an actor is ready to send a message via an output port, it blocks until the receiving actor is ready to receive it. Similarly if an actor is ready to receive a message via an input port, it blocks until the sending actor is ready to send it. Thus, this domain realizes both **blocking writes** and blocking reads.

This domain supports both conditional and multiway rendezvous. In **conditional rendezvous**, an actor is willing to rendezvous with any one of several other actors.[2] In **multiway rendezvous**, an actor requires rendezvous with multiple other actors at the same time.[3] When using conditional rendezvous, the choice of which rendezvous occurs is nondeterministic, in general, since which rendezvous occurs will generally depend on the thread scheduler.

The Rendezvous domain is based on the communicating sequential processes (**CSP**) model first proposed by Hoare (1978) and the calculus of communicating systems (**CCS**), given by Milner (1980). It also forms the foundation for the **Occam** programming language (Galletly, 1996), which enjoyed some success for a period of time in the 1980s and 1990s for programming parallel computers.

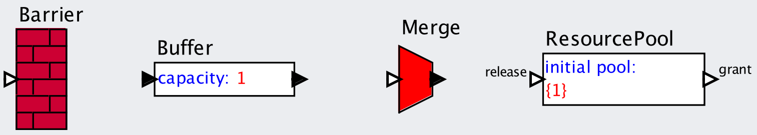
Rendezvous-based communication is also known as **synchronous message passing**, but we avoid this term to avoid confusion with the SR (synchronous-reactive) domain, described in Chapter 5, and the SDF domain, described in Chapter 3.

[2]For those fameliar with the Ada language, this is similar to the select statement.

[3]Multiway rendezvous is also called **barrier synchronization**, since it blocks each participating thread until all participating threads have reached a barrier point, indicated by a send or a get via a port.

Sidebar: Useful Actors for Rendezvous Models

A few particularly useful actors in Rendezvous models are shown below:



- **Barrier**. A barrier synchronization actor. This actor will accept inputs only when the sending actors on *all* input channels are ready to send.
- **Buffer**. A FIFO buffer. This actor buffers data provided at the input, sending it to the output when needed. The actor is willing to rendezvous with the output whenever the buffer is not empty. It is willing to rendezvous with the input as long as the buffer is not full. Inputs are delivered to the output in FIFO (first-in, first-out) order. You can specify a finite or infinite capacity for the buffer.
- **Merge**. A conditional rendezvous. This actor merges any number of input sequences onto one output sequence. It begins with a rendezvous with any input. When it receives an input, it will then rendezvous with the output. After successfully delivering the input token to the output, it returns again to being willing to rendezvous with any input.
- **ResourcePool**. A resource contention manager. This actor manages a pool of resources, where each resource is represented by a token with an arbitrary value. Resources are granted on the *grant* output port and released on the *release* input port. These ports are both multiports, so resources can be granted to multiple users of the resources, and released by multiple actors. The initial pool of resources is provided by the *initialPool* parameter, which is an array of arbitrary type. At all times during execution, this actor is ready to rendezvous with any other actor connected to its *release* input port. When such a rendezvous occurs, the token provided at that input is added to the resource pool. In addition, whenever the resource pool is non-empty, this actor is ready to rendezvous with any actor connected to its *grant* output port. If there are multiple such actors, this will be a conditional rendezvous, and hence may introduce nondeterminism into the model. When such an output rendezvous occurs, the actor sends the first token in the resource pool to that output port and removes that token from the resource pool.

4.2.1 Multiway Rendezvous

In multiway rendezvous, an actor performs a rendezvous with multiple other actors at the same time. Two mechanisms are provided by the Rendezvous director for this. First, if an actor sends an output to multiple other actors, the communication forms a multiway rendezvous. All destination actors must be ready to receive the token before any destination actor will receive it. And the sender will block until all destination actors are ready to receive the token.

Example 4.4: The model in Figure 4.4 illustrates multiway rendezvous. In this example, the Ramp is sending an increasing sequence of integers to the Display. However, the transfer is constrained to occur only when the Sleep actor reads inputs, because in the Rendezvous domain, sending data to multiple recipients via a relation is accomplished via a multiway rendezvous. The **Sleep** actor reads data, then sleeps an amount of time given by its bottom input before reading the next input data token. In this case, it will wait a random amount of time (given by the **Uniform** random-number generator) between input readings. This has the side effect of constraining the transfers from the Ramp to the Display to occur with the same random intervals.

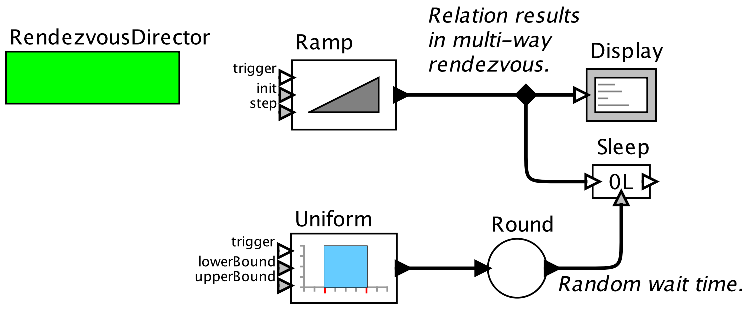


Figure 4.4: An illustration of multiway rendezvous, where the timing of the communication between the Ramp and the Display is controlled by the Sleep actor. [online]

The Barrier actor (see box on page 146) also performs multiway rendezvous.

Example 4.5: The model in Figure 4.5 illustrates multiway rendezvous using the Barrier actor. In this example, the two Ramp actors are sending increasing sequences of integers to the Display actors. Again, the transfer is constrained to occur only when both the Barrier actor and the Sleep actor read inputs. Thus, a multiway rendezvous between the two Ramp actors, the two Display actors, the Barrier actor, and the Sleep actor constrains the two transfers to the Display actors to occur simultaneously.

4.2.2 Conditional Rendezvous

In conditional rendezvous, an actor is willing to rendezvous with any one of several other actors. Usually, this results in nondeterministic models.

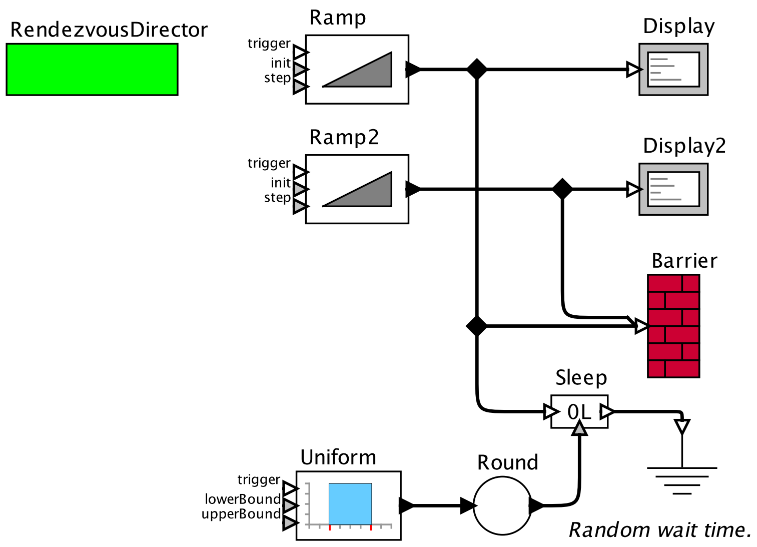


Figure 4.5: An illustration of multiway rendezvous using the Barrier actor. [online]

Example 4.6: The model in Figure 4.6 illustrates conditional rendezvous. This model uses the Merge actor (see box on page 146). The top Ramp actor will produce the sequence 0, 1, 2, 3, 4, 5, 6, 7 on its output. The bottom Ramp will produce the sequence -1, -2, -3, -4, -5, -6, -7, -8. The Display actor will display a nondeterministic merging of these two sequences.

The example in Figure 4.6 includes a parameter *SuppressDeadlockReporting* with value true. The Ramp actors starve the model by specifying a finite *firingCountLimit*, thus providing a stopping condition similar to the ones we would use with PN (see Section 4.1.2). By default, the director will report the deadlock, but with the *SuppressDeadlockReporting* parameter, it silently stops the execution of the model. This parameter indicates that the deadlock is a normal termination and not an error condition.

Suppose that, unlike the previous example, we wish to deterministically interleave the outputs of the two Ramp actors, alternating their outputs to get the sequence 0, -1, 1, -2, 2, ... One way to accomplish that, due to Arbab (2006), is shown in Figure 4.7. This model relies on the fact that the input to the Buffer actor (see box on page 146) participates in a multiway rendezvous with both instances of Ramp and the top channel of the Merge actor. Since it has capacity one, it forces this rendezvous to occur before it provides an input to

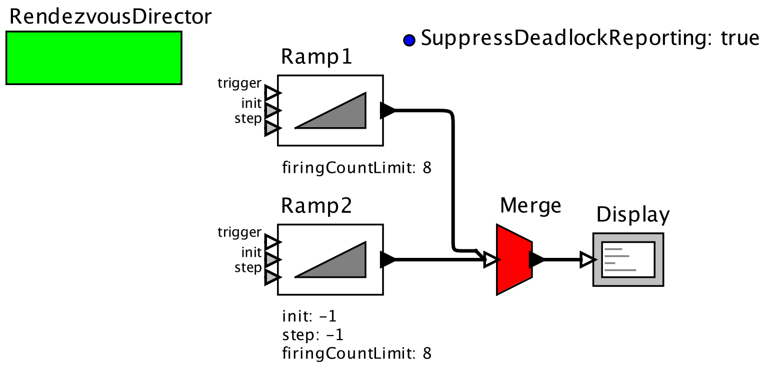


Figure 4.6: An illustration of conditional rendezvous for nondeterministic merge. [online]

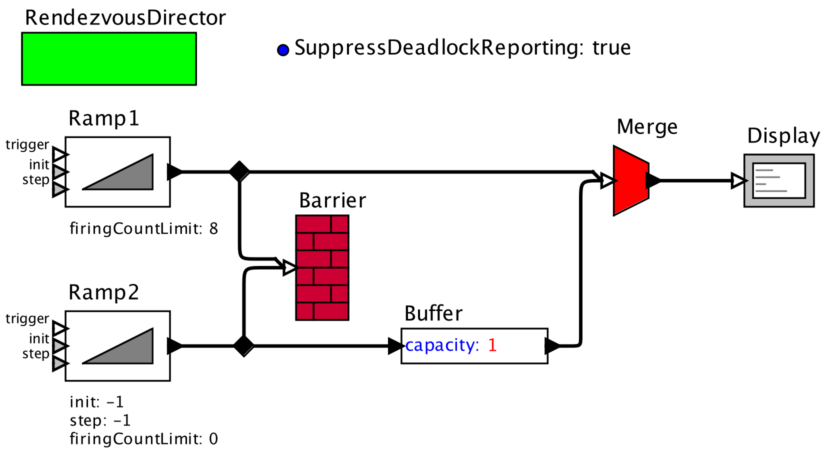


Figure 4.7: An illustration of conditional rendezvous used to create a deterministic merge. [online]

the bottom channel of the merge, and it blocks subsequent instances of this rendezvous until after it has provided the bottom input to the Merge.

Although this model is extremely clever, it is using nondeterministic mechanisms to accomplish deterministic aims. In fact, it is easy to construct a much simpler model that accomplishes the same goal without any nondeterministic mechanisms (see Exercise 7).

4.2.3 Resource Management

The conditional rendezvous mechanism provided by the Rendezvous director is ideally suited to resource management problems, where actors compete for shared resources. Generally, nondeterminism is acceptable and expected for such models. The Resource-Pool actor (see box on page 146) is ideal for such applications.

Example 4.7: The model in Figure 4.8 illustrates resource management where a pool (in this case containing only one resource) provides that resource nondeterministically to one of two Sleep actors. The resource is represented by an integer that initially has value 0. This value will be incremented each time the resource is

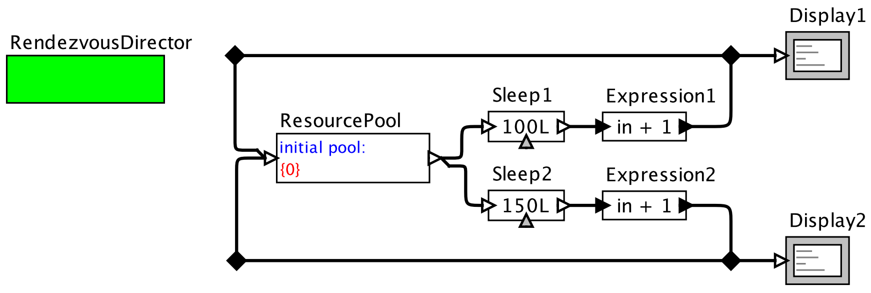


Figure 4.8: An illustration of conditional rendezvous for resource management. [online]

used. The Sleep actor that gets the resource holds it for a fixed amount of time (100 and 150 ms, respectively). After this time, it releases the resource, sending it to an Expression actor, which increments the value of the resource and then returns it to the resource pool.

The input and output ports of the ResourcePool actor both realize a conditional rendezvous. Hence, it is nondeterministic which Sleep actor will acquire the resource when both are ready for the resource. Note that there is no assurance of fairness in this system, and in fact it is possible for only one of the two Sleep actors to get access to the resource.

4.3 Summary

The two domains described in this chapter, PN and Rendezvous, both execute each actor in a model in its own thread. A PN model is deterministic as long as it does not include an instance of NondeterministicMerge. The Rendezvous domain is not generally deterministic. PN is particularly useful when models include actors that may block for indeterminate amounts of time, and where we don't wish to block the entire the model when this occurs. Rendezvous is particularly useful for resource management problems, where there is contention for limited resources, and for models where the timing of concurrent actions needs to be synchronized.

Exercises

The purpose of these exercises is to develop some intuition about the process networks model of computation and how programming with it differs from programming with an imperative model.[4] For all of the following exercises, you should use the PN Director and "simple" actors to accomplish the task. In particular, the following actors are sufficient:

- Ramp and Const (in `Sources`)
- Display and Discard (in `Sinks`)
- BooleanSwitch and BooleanSelect (in `FlowControl`→`BooleanFlowControl`)
- SampleDelay (in `FlowControl`→`SequenceControl`)
- Comparator, Equals, LogicalNot, or LogicGate (in `Logic`)

Feel free to use any other actor that you believe to be "simple." Also, feel free to use any other actors, simple or not, for testing your composite actors, but stick to simple ones for the implementation of the composite actors.

1. The SampleDelay actor produces initial tokens. In this exercise, you will create a composite actor that consumes initial tokens, and hence be thought of as a negative delay.

   (a) Create a PN model containing a composite actor with one input port and one output port, where the output sequence is the same as the input sequence except that the first token is missing. That is, the composite actor should discard the first token and then act like an identity function. Demonstrate by some suitable means that your model works as required.

   (b) Augment your model so that the number of initial tokens that are discarded is given by a parameter of the composite actor. **Hint:** It may be useful to know that the expression language[5] (see Chapter 13) includes a built-in function `repeat`, where, for example,

[4]You may want to run vergil with the -pn option, which gives you a subset of Ptolemy II that is more than adequate to do these exercises. To do this on the command line, simply type "`vergil -pn`". If you are running Ptolemy II from Eclipse, then in the toolbar of the Java perspective, select Run Configurations. In the Arguments tab, enter -pn.

[5]Note that you can easily explore the expression language by opening an ExpressionEvaluator window, available in the [`File`→`New`] menu. Also, clicking on Help in any parameter editor window will provide documentation for the expression language.

```
    repeat(5, 1) = {1, 1, 1, 1, 1}
```

2. This problem explores operations on a stream of data that depend on the data in the stream.
   (a) Create a PN model containing a composite actor with one input port and one output port, where the output sequence is the same as the input sequence except that any consecutive sequence of identical tokens is replaced by just one token with the same value. That is, redundant tokens are removed. Demonstrate by some suitable means that your model works as required.
   (b) Can your implementation run forever with bounded buffers? Give an argument that it can, or explain why there is no implementation that can run with bounded buffers.

3. Create an implementation of an OrderedMerge actor using only "simple" PN actors. (Note that there is a Java implementation of OrderedMerge, which you should not use, see box on page 143.) Your implementation should be a composite actor with two input ports and one output port. Given any two numerically increasing sequences of tokens on the input ports, your actor should merge these sequences into one numerically increasing sequence without losing any tokens. If the two sequences contain tokens that are identical, then the order in which they come out does not matter.

4. In Figure 4.9 is a model that generates a sequence of numbers known as the **Hamming numbers**. These have the form $2^n 3^m 5^k$, and they are generated in numerically increasing order, with no redundancies. This model can be found in the PN demos (labeled as OrderedMerge). Can this model run forever with bounded buffers? Why or why not?

   For this problem, assume that the data type being used is unbounded integers, rather than what is actually used, which is 32 bit integers. With 32 bit integers, the numbers will quickly overflow the representable range of the numbers, and wrap around to negative numbers.

5. A common scenario in embedded systems is that multiple sensors provide data at different rates, and the data must be combined to form a coherent view of the physical world. In general, this problem is called **sensor fusion**. The signal processing involved in forming a coherent view from noisy sensor data can be quite sophisticated, but in this exercise we will focus not on the signal processing, but rather on the concurrency and logical control flow. At a low level, sensors are connected to

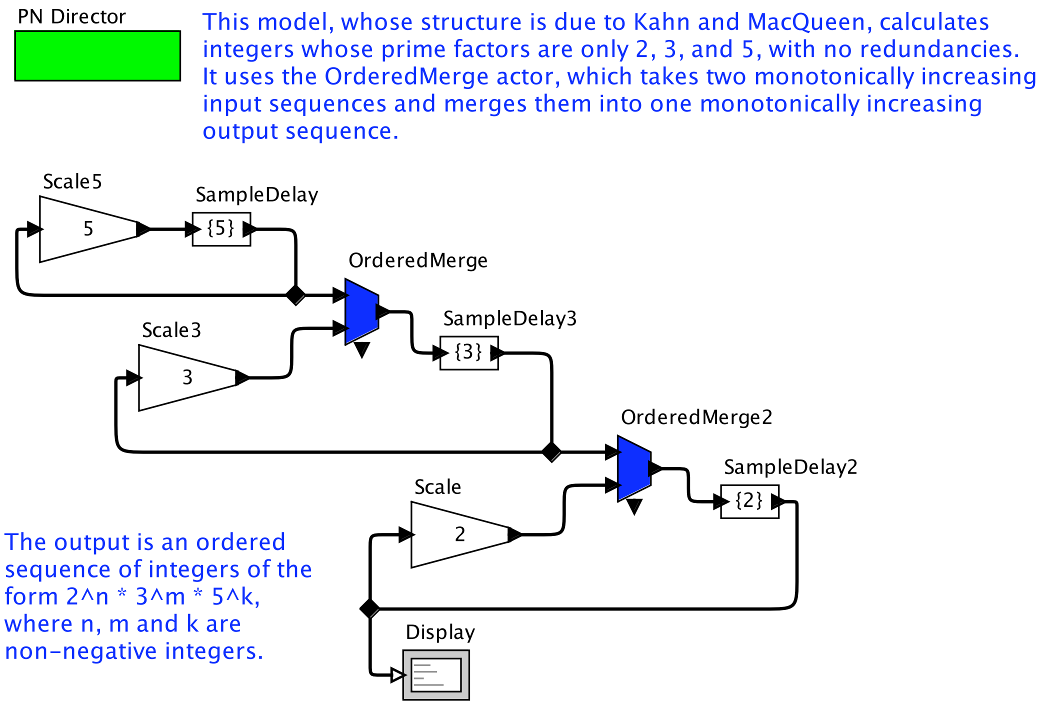

PN Director
This model, whose structure is due to Kahn and MacQueen, calculates integers whose prime factors are only 2, 3, and 5, with no redundancies. It uses the OrderedMerge actor, which takes two monotonically increasing input sequences and merges them into one monotonically increasing output sequence.
Scale5
5
SampleDelay
{5}
OrderedMerge
SampleDelay3
{3}
Scale3
3
OrderedMerge2
SampleDelay2
{2}
Scale
2
The output is an ordered sequence of integers of the form 2^n * 3^m * 5^k, where n, m and k are non-negative integers.
Display

Figure 4.9: Model that generates a sequence of Hamming numbers. [online]

embedded processors by hardware that will typically trigger processor interrupts, and interrupt service routines will read the sensor data and store it in buffers in memory. The difficulties arise when the rates at which the data are provided are different (they may not even be related by a rational multiple, or may vary over time, or may even be highly irregular).

Assume we have two sensors, SensorA and SensorB, both making measurements of the same physical phenomenon that happens to be a sinusoidal function of time, as follows:

$$\forall\, t \in \mathbb{R}, \quad x(t) = sin(2\pi t/10)$$

Assuming time t is in seconds, this has a frequency of 0.1 Hertz. Assume further that the two sensors sample the signal with distinct sampling intervals to yield the

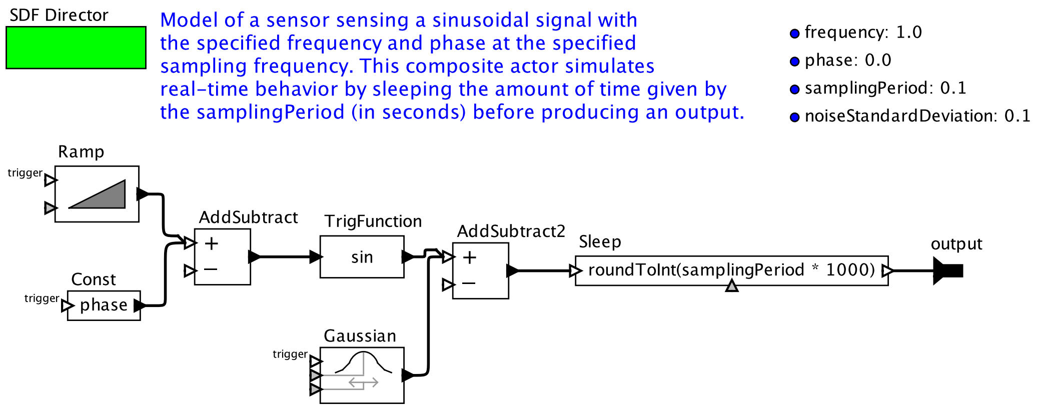

SDF Director
Model of a sensor sensing a sinusoidal signal with the specified frequency and phase at the specified sampling frequency. This composite actor simulates real-time behavior by sleeping the amount of time given by the samplingPeriod (in seconds) before producing an output.
● frequency: 1.0
● phase: 0.0
● samplingPeriod: 0.1
● noiseStandardDeviation: 0.1
Ramp
trigger
AddSubtract
TrigFunction
sin
AddSubtract2
Sleep
roundToInt(samplingPeriod * 1000)
output
Const
trigger
phase
Gaussian
trigger

Figure 4.10: Model of a real-time sensor. [online]

following measurements:

$$\forall\, n \in \mathbb{Z}, \quad x_A(n) = x(nT_A) = sin(2\pi n T_A/10),$$

where T_A is sampling interval of SensorA. A similar set of measurements is taken by SensorB, which samples with period T_B.

A model of such a sensor for use with the PN director of Ptolemy II is shown in figure 4.10. You can create an instance of that sensor in Vergil by invoking the [`Graph→Instantiate Entity`] menu command, and filling in the boxes as follows:

```
    class: SensorModel
    location (URL): http://embedded.eecs.berkeley.edu/
                    concurrency/models/SensorModel.xml
```

Create two instances of the sensor in a Ptolemy II model with a PN director.

The sensor has some parameters. The *frequency* you should set to 0.1 to match the equations above. The *samplingPeriod* you should set to 0.5 seconds for one of the sensor instances, and 0.75 seconds for the other. You are to perform the following experiments.[6]

[6] You may find the actors described in the sidebars on pages 106, 107, and 119 useful.

(a) Connect each sensor instance to its own instance of the SequencePlotter. Execute the model. You will likely want to change the parameters of the SequencePlotter so that *fillOnWrapup* is `false`, and you will want to set the *X Range* of the plot to, say, "0.0, 50.0" (do this by clicking on the second button from the right at the upper right of each plot). Describe what you see. Do the two sensors accurately reflect the sinusoidal signal? Why do they appear to have different frequencies?

(b) A simple technique for sensor fusion is to simply average sensor data. Construct a model that averages the data from the two sensors by simply adding the samples together and multiplying by 0.5. Plot the resulting signal. Is this signal an improved measurement of the signal? Why or why not? Will this model be able to run forever with bounded memory? Why or why not?

(c) The sensor fusion strategy of averaging the samples can be improved by normalizing the sample rates. For the sample periods given, 0.5 and 0.75, find a way to do this in PN. Comment about whether this technique would work effectively if the sample periods did not bear such a simple relationship to one another. For example, suppose that instead of 0.5 seconds, the period on the first sensor was 0.500001.

(d) When sensor data is collected at different rates without a simple relationship, one technique that can prove useful is to create time stamps for the data and to use those time stamps to improve the measurements. Construct a model that does this, with the objective simply of creating a plot that combines the data from the two sensors in a sensible way.

6. In this problem, we explore how to use the mechanisms of Section 4.1.2 to deterministically halt the execution of a PN model. Specifically, in each case, we consider a Source actor feeding a potentially infinite sequence of data tokens to a Display actor. We wish to make this sequence finite with a specific length, and we wish to ensure that the Display actor displays every element of the sequence.

   (a) Suppose that you have a Source actor with one output port and no parameters whose process iterates forever producing outputs. Suppose that its outputs are read by a Display actor, which has one input port and no output ports. Find a way to use the Stop actor to deterministically stop the execution, or argue that there is no way to do so. Specifically, the Source actor should produce a specified number of outputs, and every one of these outputs should be consumed and displayed by the Display actor before execution halts.

(b) Most Source actors in Ptolemy II have a *firingCountLimit* parameter that limits the number of outputs they produce. Show that this can be used to deterministically halt the execution without the help of a Stop actor.

(c) Many Source actors in Ptolemy II have *trigger* input ports. If these inputs are connected, then the actor process will read a value from that input port before producing each output. Show how to use this mechanism, with or without the Stop actor, to achieve our goal of deterministically halting execution, or argue that it is not possible to do so. Again, the Source should produce a prespecified amount of data, and the Display should consume and display all of that data. You may use Switch, Select, or any other reasonably simple actor. Be sure to explain each actor you use, unless you are sure it is exactly the actor provided in the Vergil library.

7. Figure 4.7 shows a model that deterministically interleaves the outputs of two Ramp actors. That model uses a nondeterministic mechanism (the conditional rendezvous of the Merge actor), and then carefully regulates the nondeterminism using a multiway rendezvous and a Buffer actor. The end result is deterministic. However, the same objective (deterministically interleaving two streams in an alternating, round-robin fashion) can be accomplished with purely deterministic mechanisms. Construct a Rendezvous model that does this. **Hint**: Your model will probably work unchanged with using the PN or SDF directors instead of Rendezvous.