Chapter 2 from: C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, H. Zheng "Heterogeneous Concurrent Modeling and Design in Java (Volume 2: Ptolemy II Software Architecture) ," Technical Memorandum UCB/ERL M04/16, University of California, Berkeley, CA USA 94720, June 24, 2004.

2 Actor Package

Author: *Edward A. Lee*
Contributors: *Mudit Goel*
Christopher Hylands
Jie Liu
Lukito Muliadi
Steve Neuendorffer
Neil Smyth
Yuhong Xiong
Haiyang Zheng

2.1 Concurrent Computation

In the kernel package, entities have no semantics. They are syntactic placeholders. In many of the uses of Ptolemy II, entities are executable. The actor package provides basic support for executable entities. It makes a minimal commitment to the semantics of these entities by avoiding specifying the order in which actors execute (or even whether they execute sequentially or concurrently), and by avoiding specifying the communication mechanism between actors. These properties are defined in the domains.

In most uses, these executable entities conceptually (if not actually) execute concurrently. The goal of the actor package is to provide a clean infrastructure for such concurrent execution that is neutral about the model of computation. It is intended to support dataflow, discrete-event, synchronous-reactive, continuous-time, communicating sequential processes, and process networks models of computation, at least. The detailed model of computation is then implemented in a set of derived classes called a *domain*. Each domain is a separate package.

Ptolemy II is an object-oriented application framework. *Actors* [1] extend the concept of objects to concurrent computation. Actors encapsulate a thread of control and have interfaces for interacting with other actors. They provide a framework for “open distributed object-oriented systems.” An actor can create other actors, send messages, and modify its own local state.

Inspired by this model, we group a certain set of classes that support computation within entities in the actor package. Our use of the term “actors,” however, is somewhat broader, in that it does not require an entity to be associated with a single thread of control, nor does it require the execution of threads associated with entities to be fair. Some subclasses, in other packages, impose such requirements, as we will see, but not all.

Agha’s actors [1] can only send messages to *acquaintances* — actors whose addresses it was given at creation time, or whose addresses it has received in a message, or actors it has created. Our equivalent constraint is that an actor can only send a message to an actor if it has (or can obtain) a reference to a receiver belonging to an input port of that actor. The usual mechanism for obtaining a reference to a receiver uses the topology, probing for a port that it is connected to. Our relations, therefore, provide explicit management of acquaintance associations. Derived classes may provide additional implicit mechanisms. We define *actor* more loosely to refer to an entity that processes data that it receives through its ports, or that creates and sends data to other entities through its ports.

The actor package provides templates for two key support functions. These templates support message passing and the execution sequence (flow of control). They are *templates* in that no mechanism is actually provided for message passing or flow of control, but rather base classes are defined so that domains only need to override a few methods, and so that domains can interoperate.

2.2 Message Passing

The actor package provides templates for executable entities called *actors* that communicate with one another via message passing. Messages are encapsulated in *tokens* (see the Data Package chapter). Messages are sent and received via ports. IOPort is the key class supporting message transport, and is shown in figure 2.2. An IOPort can only be connected to other IOPort instances, and only via IORelations. The IORelation class is also shown in figure 2.2. TypedIOPort and TypedIORelation are subclasses that manage type resolution. These subclasses are used much more often, in order to benefit from the type system. This is described in detail in the Type System chapter.

An instance of IOPort can be an input, an output, or both. An *input port* (one that is capable of receiving messages) contains one or more instances of objects that implement the Receiver interface. Each of these receivers is capable of receiving messages from a distinct *channel*.

The type of receiver used depends on the communication protocol, which depends on the model of computation. The actor package includes two receivers, Mailbox and QueueReceiver. These are generic enough to be useful in several domains. The QueueReceiver class contains a FIFOQueue, the capacity of which can be controlled. It also provides a mechanism for tracking the history of tokens that are received by the receiver. The Mailbox class implements a FIFO (first in, first out) queue with capacity equal to one.

2.2.1 Data Transport

Data transport is depicted in figure 2.1. The originating actor E1 has an output port P1, indicated in the figure with an arrow in the direction of token flow. The destination actor E2 has an input port P2, indicated in the figure with another arrow. E1 calls the send() method of P1 to send a token *t* to a remote actor. The port obtains a reference to a remote receiver (via the IORelation) and calls the put() method of the receiver, passing it the token. The destination actor retrieves the token by calling the get() method of its input port, which in turn calls the get() method of the designated receiver.

Domains typically provide specialized receivers. These receivers override get() and put() to implement the communication protocol pertinent to that domain. A domain that uses asynchronous message passing, for example, can usually use the QueueReceiver shown in figure 2.2. A domain that uses synchronous message passing (rendezvous) has to provide a new receiver class.

In figure 2.1 there is only a single channel, indexed 0. The "0" argument of the send() and get() methods refer to this channel. A port can support more than one channel, however, as shown in figure 2.3. This can be represented by linking more than one relation to the port, or by linking a relation that has a width greater than one. A port that supports this is called a *multiport*. The channels are indexed $0, \ldots, N-1$, where N is the number of channels. An actor distinguishes between channels using this index in its send() and get() methods. By default, an IOPort is not a multiport, and thus supports only one channel (or zero, if it is left unconnected). It is converted into a multiport by calling its setMultiport() method with a *true* argument. After conversion, it can support any number of channels.

Multiports are typically used by actors that communicate via an indeterminate number of channels. For example, a "distributor" or "demultiplexor" actor might divide an input stream into a number of output streams, where the number of output streams depends on the connections made to the actor. A *stream* is a sequence of tokens sent over a channel.

An IORelation, by default, represents a single channel. By calling its setWidth() method, however, it can be converted to a *bus*. A multiport may use a bus instead of multiple relations to distribute its data, as shown in figure 2.4. The *width of a relation* is the number of channels supported by the relation. If the relation is not a bus, then its width is one.

The *width of a port* is the sum of the widths of the relations linked to it. In figure 2.4, both the sending and receiving ports are multiports with width two. This is indicated by the "2" adjacent to each port. Note that the width of a port could be zero, if there are no relations linked to a port (such a port is said to be *disconnected*). Thus, a port may have width zero, even though a relation cannot. By convention, in Ptolemy II, if a token is sent from such a port, the token goes nowhere. Similarly, if a token is

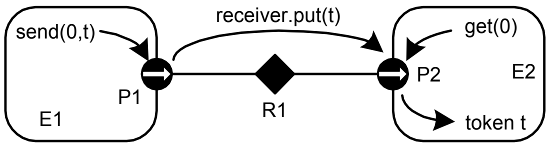

FIGURE 2.1. Message passing is mediated by the IOPort class. Its send() method obtains a reference to a remote receiver, and calls the put() method of the receiver, passing it the token *t*. The destination actor retrieves the token by calling the get() method of its input port.

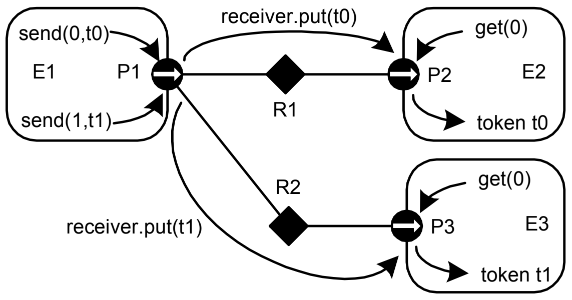

FIGURE 2.3. A port can support more than one channel, permitting an entity to send distinct data to distinct destinations via the same port. This feature is typically used when the number of destinations varies in different instances of the source actor.

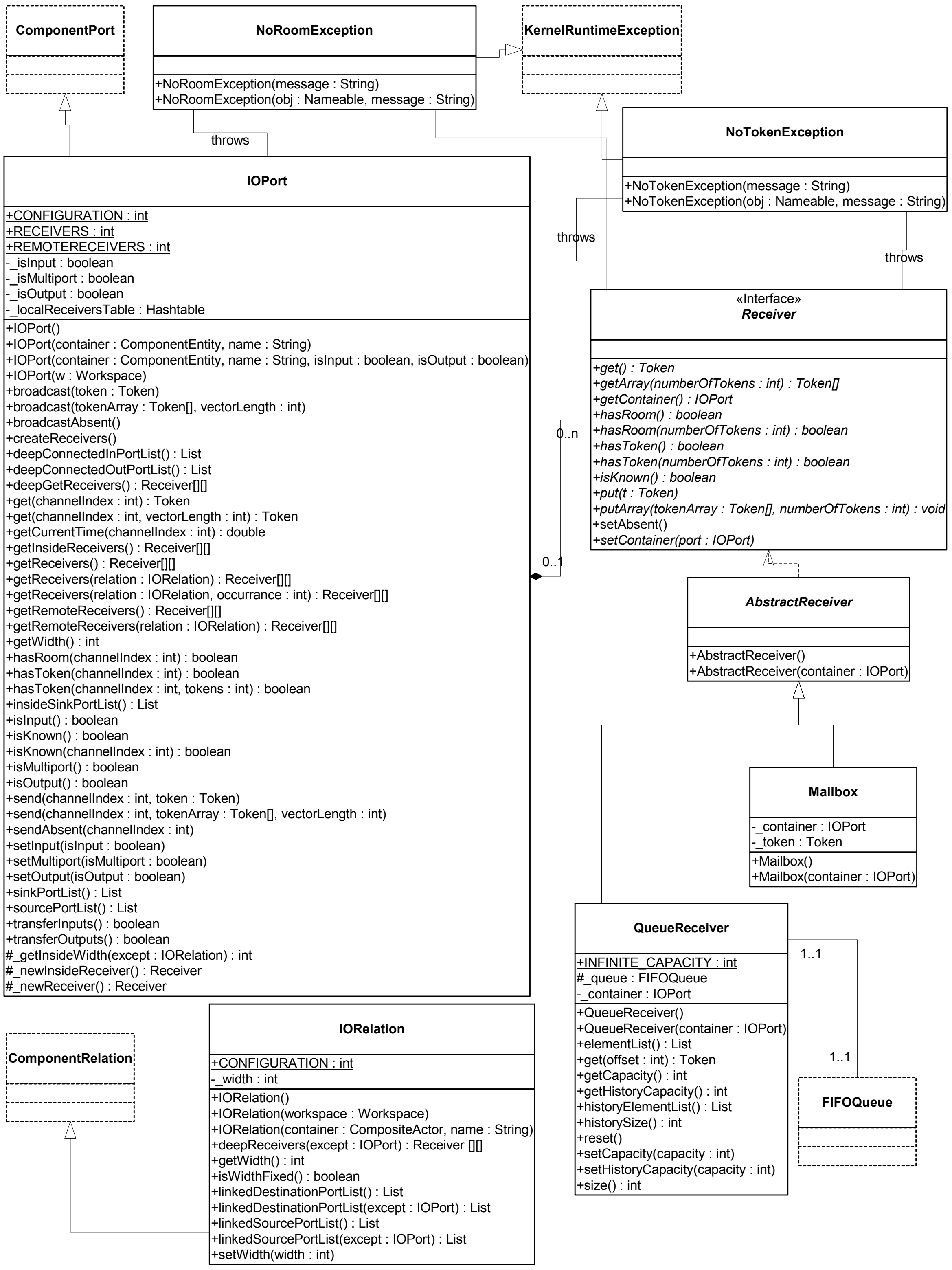
ComponentPort
NoRoomException
+NoRoomException(message : String)
+NoRoomException(obj : Nameable, message : String)
KernelRuntimeException
throws
NoTokenException
+NoTokenException(message : String)
+NoTokenException(obj : Nameable, message : String)
IOPort
+CONFIGURATION : int
+RECEIVERS : int
+REMOTERECEIVERS : int
-_isInput : boolean
-_isMultiport : boolean
-_isOutput : boolean
-_localReceiversTable : Hashtable
+IOPort()
+IOPort(container : ComponentEntity, name : String)
+IOPort(container : ComponentEntity, name : String, isInput : boolean, isOutput : boolean)
+IOPort(w : Workspace)
+broadcast(token : Token)
+broadcast(tokenArray : Token[], vectorLength : int)
+broadcastAbsent()
+createReceivers()
+deepConnectedInPortList() : List
+deepConnectedOutPortList() : List
+deepGetReceivers() : Receiver[][]
+get(channelIndex : int) : Token
+get(channelIndex : int, vectorLength : int) : Token
+getCurrentTime(channelIndex : int) : double
+getInsideReceivers() : Receiver[][]
+getReceivers() : Receiver[][]
+getReceivers(relation : IORelation) : Receiver[][]
+getReceivers(relation : IORelation, occurrance : int) : Receiver[][]
+getRemoteReceivers() : Receiver[][]
+getRemoteReceivers(relation : IORelation) : Receiver[][]
+getWidth() : int
+hasRoom(channelIndex : int) : boolean
+hasToken(channelIndex : int) : boolean
+hasToken(channelIndex : int, tokens : int) : boolean
+insideSinkPortList() : List
+isInput() : boolean
+isKnown() : boolean
+isKnown(channelIndex : int) : boolean
+isMultiport() : boolean
+isOutput() : boolean
+send(channelIndex : int, token : Token)
+send(channelIndex : int, tokenArray : Token[], vectorLength : int)
+sendAbsent(channelIndex : int)
+setInput(isInput : boolean)
+setMultiport(isMultiport : boolean)
+setOutput(isOutput : boolean)
+sinkPortList() : List
+sourcePortList() : List
+transferInputs() : boolean
+transferOutputs() : boolean
#_getInsideWidth(except : IORelation) : int
#_newInsideReceiver() : Receiver
#_newReceiver() : Receiver
throws
throws
«Interface»
Receiver
+get() : Token
+getArray(numberOfTokens : int) : Token[]
+getContainer() : IOPort
+hasRoom() : boolean
+hasRoom(numberOfTokens : int) : boolean
+hasToken() : boolean
+hasToken(numberOfTokens : int) : boolean
+isKnown() : boolean
+put(t : Token)
+putArray(tokenArray : Token[], numberOfTokens : int) : void
+setAbsent()
+setContainer(port : IOPort)
0..n
0..1
AbstractReceiver
+AbstractReceiver()
+AbstractReceiver(container : IOPort)
Mailbox
-_container : IOPort
-_token : Token
+Mailbox()
+Mailbox(container : IOPort)
QueueReceiver
+INFINITE_CAPACITY : int
#_queue : FIFOQueue
-_container : IOPort
+QueueReceiver()
+QueueReceiver(container : IOPort)
+elementList() : List
+get(offset : int) : Token
+getCapacity() : int
+getHistoryCapacity() : int
+historyElementList() : List
+historySize() : int
+reset()
+setCapacity(capacity : int)
+setHistoryCapacity(capacity : int)
+size() : int
1..1
1..1
FIFOQueue
ComponentRelation
IORelation
+CONFIGURATION : int
-_width : int
+IORelation()
+IORelation(workspace : Workspace)
+IORelation(container : CompositeActor, name : String)
+deepReceivers(except : IOPort) : Receiver [][]
+getWidth() : int
+isWidthFixed() : boolean
+linkedDestinationPortList() : List
+linkedDestinationPortList(except : IOPort) : List
+linkedSourcePortList() : List
+linkedSourcePortList(except : IOPort) : List
+setWidth(width : int)

FIGURE 2.2. Port and receiver classes for message passing under various communication protocols.

sent via a relation that is not linked to any input ports, then the token goes nowhere. Such a relation is said to be *dangling*.

A given channel may reach multiple ports, as shown in figure 2.5. This is represented by a relation that is linked to multiple input ports. In the default implementation, in class IOPort, a reference to the token is sent to all destinations. Note that tokens are assumed to be immutable, so the recipients cannot modify the value. This is important because in most domains, it is not obvious in what order the recipients will see the token.

The send() method takes a channel number argument. If the channel does not exist, the send() method silently returns without sending the token anywhere. This makes it easier for model builders, since they can simply leave ports unconnected if they are not interested in the output data.

IOPort provides a broadcast() method for convenience. This method sends a specified token to all receivers linked to the port, regardless of the width of the port. If the width is zero, of course, the token will not be sent anywhere.

2.2.2 Example

An elaborate example showing all of the above features is shown in figure 2.6. In that example, we assume that links are constructed in top-to-bottom order. The arrows in the ports indicate the direction of the flow of tokens, and thus specify whether the port is an input, an output, or both. Multiports are indicated by adjacent numbers larger than one.

The top relation is a bus with width two, and the rest are not busses. The width of port *P1* is four. Its first two outputs (channels zero and one) go to *P4* and to the first two inputs of *P5*. The third output of *P1* goes nowhere. The fourth becomes the third input of *P5,* the first input of *P6*, and the only input of *P8*, which is both an input and an output port. Ports *P2* and *P8* send their outputs to the same set of

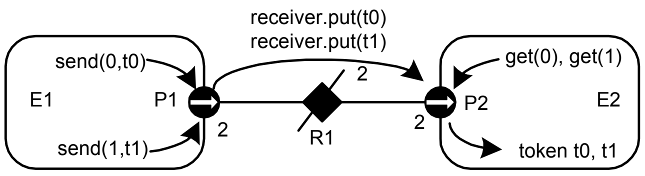

FIGURE 2.4. A bus is an IORelation that represents multiple channels. It is indicated by a relation with a slash through it, and the number adjacent to the bus is the width of the bus.

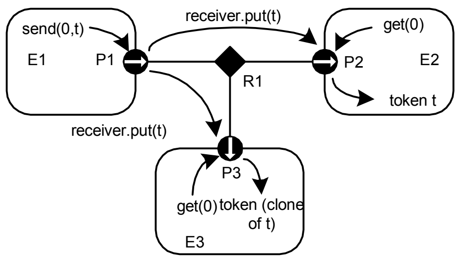

FIGURE 2.5. Channels may reach multiple destinations. This is represented by relations linking multiple input ports to an output port.

destinations, except that *P8* does not send to itself. Port *P3* has width zero, so its send() method returns without sending the token anywhere. Port *P6* has width two, but its second input channel has no output ports connected to it, so calling get(1) will trigger an exception that indicates that there is no data. Port *P7* has width zero so calling get() with any argument will trigger an exception.

2.2.3 Transparent Ports

Recall that a port is transparent if its container is transparent (isOpaque() returns *false*). A CompositeActor is transparent unless it has a local director. Figure 2.7 shows an elaborate example where busses, input, and output ports are combined with transparent ports. The transparent ports are filled in white, and again arrows indicate the direction of token flow. The Jacl code to construct this example is

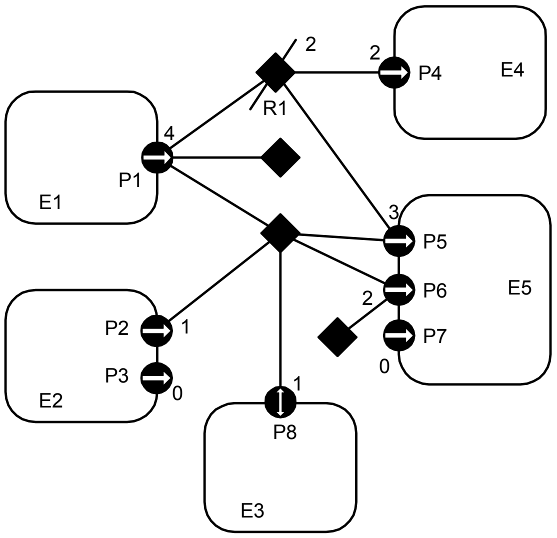

FIGURE 2.6. An elaborate example showing several features of the data transport mechanism.

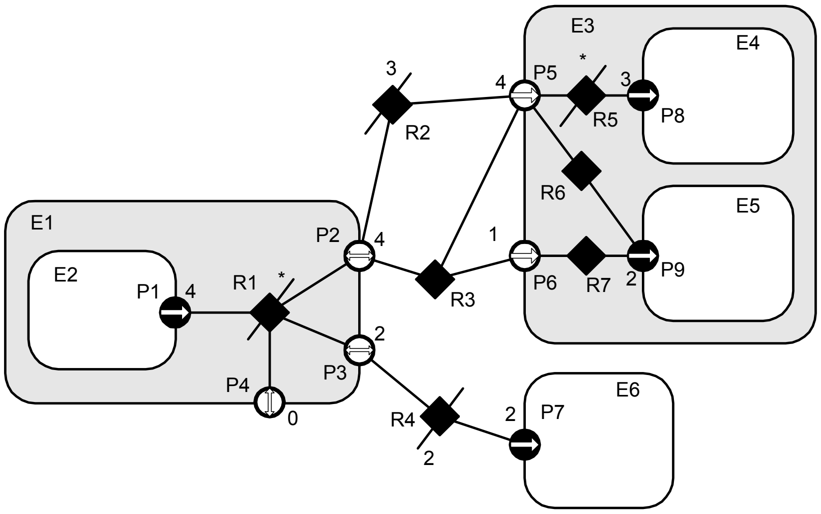

FIGURE 2.7. An example showing busses combined with input, output, and transparent ports.

shown in figure 2.8.

By definition, a transparent port is an input if either

- it is connected on the inside to the outside of an input port, or
- it is connected on the inside to the inside of an output port.

That is, a transparent port is an input port if it can accept data (which it may then just pass through to a transparent output port). Correspondingly, a transparent port is an output port if either

- it is connected on the inside to the outside of an output port, or
- it is connected on the inside to the inside of an input port.

Thus, assuming P1 is an output port and P7, P8, and P9 are input ports, then P2, P3, and P4 are both input and output ports, while P5 and P6 are input ports only.

Two of the relations that are inside composite entities (R1 and R5) are labeled as busses with a star (*) instead of a number. These are busses with unspecified width. The width is inferred from the topology. This is done by checking the ports that this relation is linked to from the inside and setting the width to the maximum of those port widths, minus the widths of other relations linked to those ports on the inside. Each such port is allowed to have at most one inside relation with an unspecified width, or an exception is thrown. If this inference yields a width of zero, then the width is defined to be one.

```
set e0 [java::new ptolemy.actor.CompositeActor]
$e0 setDirector $director
$e0 setManager $manager

set e1 [java::new ptolemy.actor.CompositeActor $e0 E1]
set e2 [java::new ptolemy.actor.AtomicActor $e1 E2]
set e3 [java::new ptolemy.actor.CompositeActor $e0 E3]
set e4 [java::new ptolemy.actor.AtomicActor $e3 E4]
set e5 [java::new ptolemy.actor.AtomicActor $e3 E5]
set e6 [java::new ptolemy.actor.AtomicActor $e0 E6]

set p1 [java::new ptolemy.actor.IOPort $e2 P1 false true]
set p2 [java::new ptolemy.actor.IOPort $e1 P2]
set p3 [java::new ptolemy.actor.IOPort $e1 P3]
set p4 [java::new ptolemy.actor.IOPort $e1 P4]
set p5 [java::new ptolemy.actor.IOPort $e3 P5]
set p6 [java::new ptolemy.actor.IOPort $e3 P6]
set p7 [java::new ptolemy.actor.IOPort $e6 P7 true false]
set p8 [java::new ptolemy.actor.IOPort $e4 P8 true false]
set p9 [java::new ptolemy.actor.IOPort $e5 P9 true false]

set r1 [java::new ptolemy.actor.IORelation $e1 R1]
set r2 [java::new ptolemy.actor.IORelation $e0 R2]
set r3 [java::new ptolemy.actor.IORelation $e0 R3]
set r4 [java::new ptolemy.actor.IORelation $e0 R4]
set r5 [java::new ptolemy.actor.IORelation $e3 R5]
set r6 [java::new ptolemy.actor.IORelation $e3 R6]
set r7 [java::new ptolemy.actor.IORelation $e3 R7]

$p1 setMultiport true
$p2 setMultiport true
$p3 setMultiport true
$p4 setMultiport true
$p5 setMultiport true
$p7 setMultiport true
$p8 setMultiport true
$p9 setMultiport true
```

```
$r1 setWidth 0
$r2 setWidth 3
$r4 setWidth 2
$r5 setWidth 0

$p1 link $r1
$p2 link $r1
$p3 link $r1
$p4 link $r1
$p2 link $r2
$p5 link $r2
$p2 link $r3
$p5 link $r3
$p6 link $r3
$p3 link $r4
$p7 link $r4
$p5 link $r5
$p8 link $r5
$p5 link $r6
$p9 link $r6
$p6 link $r7
$p9 link $r7
```

FIGURE 2.8. Tcl Blend code to construct the example in figure 2.7.

Thus, R1 will have width 4 and R5 will have width 3 in this example. The width of a transparent port is the sum of the widths of the relations it is linked to on the outside (just like an ordinary port). Thus, P4 has width 0, P3 has width 2, and P2 has width 4. Recall that a port can have width 0, but a relation cannot have width less than one.

When data is sent from P1, four distinct channels can be used. All four will go through P2 and P5, the first three will reach P8, two copies of the fourth will reach P9, the first two will go through P3 to P7, and none will go through P4.

By default, an IORelation is not a bus, so its width is one. To turn it into a bus with unspecified width, call setWidth() with a zero argument. Note that getWidth() will nonetheless never return zero (it returns at least one). To find out whether setWidth() has been called with a zero argument, call isWidthFixed() (see figure 2.2). If a bus with unspecified width is not linked on the inside to any transparent ports, then its width is one. It is not allowed for a transparent port to have more than one bus with unspecified width linked on the inside (an exception will be thrown on any attempt to construct such a topology). Note further that a bus with unspecified width is still a bus, and so can only be linked to multiports.

In general, bus widths inside and outside a transparent port need not agree. For example, if $M < N$ in figure 2.9, then first M channels from P1 reach P3, and the last $N - M$ channels are dangling. If $M > N$, then all N channels from P1 reach P3, but the last $M - N$ channels at P3 are dangling. Attempting to get a token from these channels will trigger an exception. Sending a token to these channels just results in loss of the token.

Note that data is not actually transported through the relations or transparent ports in Ptolemy II. Instead, each output port caches a list of the destination receivers (in the form of the two-dimensional array returned by getRemoteReceivers()), and sends data directly to them. The cache is invalidated whenever the topology changes, and only at that point will the topology be traversed again. This significantly improves the efficiency of data transport.

2.2.4 Data Transfer in Various Models of Computation

The receiver used by an input port determines the communication protocol. This is closely bound to the model of computation. The IOPort class creates a new receiver when necessary by calling its _newReceiver() protected method. That method delegates to the director returned by getDirector(), calling its newReceiver() method (the Director class will be discussed in section 2.3 below). Thus, the director controls the communication protocol, in addition to its primary function of determining the flow of control. Here we discuss the receivers that are made available in the actor package. This should not be viewed as an exhaustive set, but rather as a particularly useful set of receivers. These receivers are shown in figure 2.2.

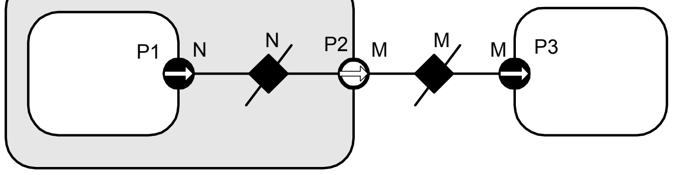

FIGURE 2.9. Bus widths inside and outside a transparent port need not agree..

Mailbox Communication. The Director base class by default returns a simple receiver called a Mailbox. A *mailbox* is a receiver that has capacity for a single token. It will throw an exception if it is empty and get() is called, or it is full and put() is called. Thus, a subclass of Director that uses this should schedule the calls to put() and get() so that these exceptions do not occur, or it should catch these exceptions.

Asynchronous Message Passing. This is supported by the QueueReceiver class. A QueueReceiver contains an instance of FIFOQueue, from the actor.util package, which implements a first-in, first-out queue. This is appropriate for all flavors of dataflow as well as Kahn process networks.

In the Kahn process networks model of computation [63], which is a generalization of dataflow [82], each actor has its own thread of execution. The thread calling get() will stall if the corresponding queue is empty. If the size of the queue is bounded, then the thread calling put() may stall if the queue is full. This mechanism supports implementation of a strategy that ensures bounded queues whenever possible [117].

In the process networks model of computation, the *history* of tokens that traverse any connection is determinate under certain simple conditions. With certain technical restrictions on the functionality of the actors (they must implement monotonic functions under prefix ordering of sequences), our implementation ensures determinacy in that the history does not depend on the order in which the actors carry out their computation. Thus, the history does not depend on the policies used by the thread scheduler.

FIFOQueue is a support class that implements a first-in, first-out queue. It is part of the actor.util package, shown in figure 2.10. This class has two specialized features that make it particularly useful in this context. First, its capacity can be constrained or unconstrained. Second, it can record a finite or infinite history, the sequence of objects previously removed from the queue. The history mechanism is useful both to support tracing and debugging and to provide access to a finite buffer of previously consumed tokens.

An example of an actor definition is shown in figure 2.11. This actor has a multiport output. It reads successive input tokens from the input port and distributes them to the output channels. This actor is written in a domain-polymorphic way, and can operate in any of a number of domains. If it is

```
public class Distributor extends TypedAtomicActor {

   public TypedIOPort _input;
   public TypedIOPort _output;

   public Distributor(CompositeActor container, String name)
           throws NameDuplicationException, IllegalActionException {
      super(container, name);
      _input = new TypedIOPort(this, "input", true, false);
      _output = new TypedIOPort(this, "output", false, true);
      _output.setMultiport(true);
   }

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

FIGURE 2.11. An actor that distributes successive input tokens to a set of output channels.

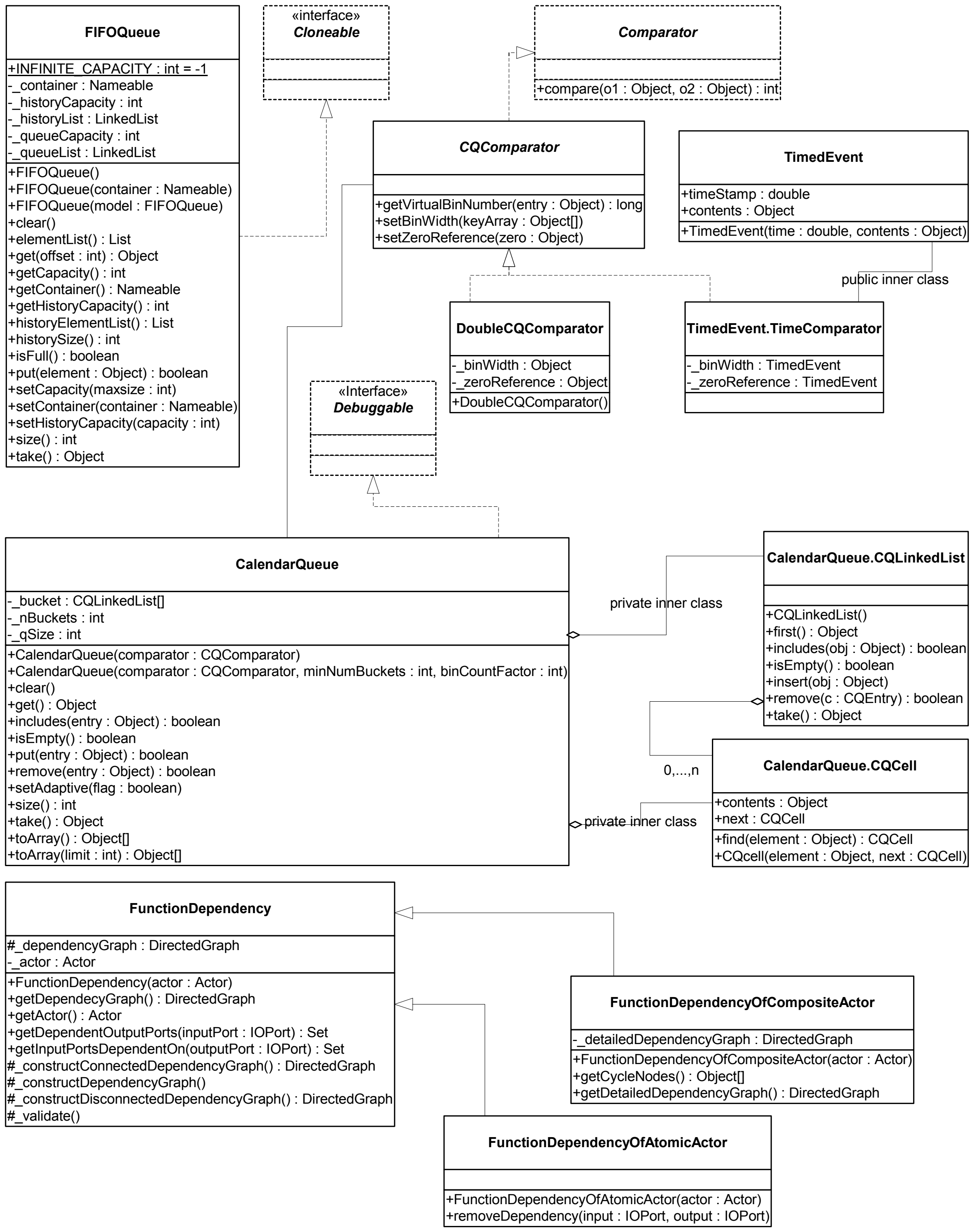
FIFOQueue
+INFINITE_CAPACITY : int = -1
-_container : Nameable
-_historyCapacity : int
-_historyList : LinkedList
-_queueCapacity : int
-_queueList : LinkedList
+FIFOQueue()
+FIFOQueue(container : Nameable)
+FIFOQueue(model : FIFOQueue)
+clear()
+elementList() : List
+get(offset : int) : Object
+getCapacity() : int
+getContainer() : Nameable
+getHistoryCapacity() : int
+historyElementList() : List
+historySize() : int
+isFull() : boolean
+put(element : Object) : boolean
+setCapacity(maxsize : int)
+setContainer(container : Nameable)
+setHistoryCapacity(capacity : int)
+size() : int
+take() : Object
«interface»
Cloneable
Comparator
+compare(o1 : Object, o2 : Object) : int
CQComparator
+getVirtualBinNumber(entry : Object) : long
+setBinWidth(keyArray : Object[])
+setZeroReference(zero : Object)
TimedEvent
+timeStamp : double
+contents : Object
+TimedEvent(time : double, contents : Object)
public inner class
DoubleCQComparator
-_binWidth : Object
-_zeroReference : Object
+DoubleCQComparator()
TimedEvent.TimeComparator
-_binWidth : TimedEvent
-_zeroReference : TimedEvent
«Interface»
Debuggable
CalendarQueue
-_bucket : CQLinkedList[]
-_nBuckets : int
-_qSize : int
+CalendarQueue(comparator : CQComparator)
+CalendarQueue(comparator : CQComparator, minNumBuckets : int, binCountFactor : int)
+clear()
+get() : Object
+includes(entry : Object) : boolean
+isEmpty() : boolean
+put(entry : Object) : boolean
+remove(entry : Object) : boolean
+setAdaptive(flag : boolean)
+size() : int
+take() : Object
+toArray() : Object[]
+toArray(limit : int) : Object[]
private inner class
CalendarQueue.CQLinkedList
+CQLinkedList()
+first() : Object
+includes(obj : Object) : boolean
+isEmpty() : boolean
+insert(obj : Object)
+remove(c : CQEntry) : boolean
+take() : Object
0,...,n
CalendarQueue.CQCell
+contents : Object
+next : CQCell
+find(element : Object) : CQCell
+CQcell(element : Object, next : CQCell)
private inner class
FunctionDependency
#_dependencyGraph : DirectedGraph
-_actor : Actor
+FunctionDependency(actor : Actor)
+getDependecyGraph() : DirectedGraph
+getActor() : Actor
+getDependentOutputPorts(inputPort : IOPort) : Set
+getInputPortsDependentOn(outputPort : IOPort) : Set
#_constructConnectedDependencyGraph() : DirectedGraph
#_constructDependencyGraph()
#_constructDisconnectedDependencyGraph() : DirectedGraph
#_validate()
FunctionDependencyOfCompositeActor
-_detailedDependencyGraph : DirectedGraph
+FunctionDependencyOfCompositeActor(actor : Actor)
+getCycleNodes() : Object[]
+getDetailedDependencyGraph() : DirectedGraph
FunctionDependencyOfAtomicActor
+FunctionDependencyOfAtomicActor(actor : Actor)
+removeDependency(input : IOPort, output : IOPort)

FIGURE 2.10. Static structure diagram for the actor.util package.

used in the PN domain, then its input will have a QueueReceiver and the output will be connected to ports with instances QueueReceiver.

Rendezvous Communications. Rendezvous, or synchronous communication, requires that the originator of a token and the recipient of a token both be simultaneously ready for the data transfer. As with process networks, the originator and the recipient are separate threads. The originating thread indicates a willingness to rendezvous by calling send(), which in turn calls the put() method of the appropriate receiver. The recipient indicates a willingness to rendezvous by calling get() on an input port, which in turn calls get() of the designated receiver. Whichever thread does this first must stall until the other thread is ready to complete the rendezvous.

This style of communication is implemented in the CSP domain. In the receiver in that domain, the put() method suspends the calling thread if the get() method has not been called. The get() method suspends the calling thread if the put() method has not been called. When the second of these two methods is called, it wakes up the suspended thread and completes the data transfer. The actor shown in figure 2.11 works unchanged in the CSP domain, although its behavior is different in that input and output actions involve rendezvous with another thread.

Nondeterministic transfers can be easily implemented using this mechanism. Suppose for example that a recipient is willing to rendezvous with any of several originating threads. It could spawn a thread for each. These threads should each call get(), which will suspend the thread until the originator is willing to rendezvous. When one of the originating threads is willing to rendezvous with it, it will call put(). The multiple recipient threads will all be awakened, but only one of them will detect that its rendezvous has been enabled. That one will complete the rendezvous, and others will die. Thus, the first originating thread to indicate willingness to rendezvous will be the one that will transfer data. Guarded communication [7] can also be implemented.

Discrete-Event Communication. In the discrete-event model of computation, tokens that are transferred between actors have a *time stamp*, which specifies the order in which tokens should be processed by the recipients. The order is chronological, by increasing time stamp. To implement this, a discrete-event system will normally use a single, global, sorted queue rather than an instance of FIFOQueue in each input port. The kernel.util package, shown in figure 2.10, provides the CalendarQueue class, which gives an efficient and flexible implementation of such a sorted queue.

2.2.5 Discussion of the Data Transfer Mechanism

This data transfer mechanism has a number of interesting features. First, note that the actual transfer of data does not involve relations, so a model of computation could be defined that did not rely on relations. For example, a global name server might be used to address recipient receivers. To construct highly dynamic networks, such as wireless communication systems, it may be more intuitive to model a system as an aggregation of unconnected actors with addresses. A name server would return a reference to a receiver given an address. This could be accomplished simply by overriding the getRemoteReceivers() method of IOPort or TypedIOPort, or by providing an alternative method for getting references to receivers. The subclass of IOPort would also have to ensure the creation of the appropriate number of receivers. The base class relies on the width of the port to determine how many receivers to create, and the width is zero if there are no relations linked.

Note further that the mechanism here supports bidirectional ports. An IOPort may return true to both the isInput() and isOutput() methods.

2.3 Execution

The Executable interface, shown in figure 2.12, is implemented by the Director class, and is extended by the Actor interface. An *actor* is an executable entity. There are two types of actors, AtomicActor, which extends ComponentEntity, and CompositeActor, which extends CompositeEntity. As the names imply, an AtomicActor is a single entity, while a CompositeActor is an aggregation of actors. Two further extensions implement a type system, TypedAtomicActor and TypedCompositeActor.

The Executable interface defines how an object can be invoked. There are eight methods. The preinitialize() method is assumed to be invoked exactly once during the lifetime of an execution of a model and before the type resolution (see the type system chapter), and the initialize() methods is assumed to be invoked once after the type resolution. The initialize() method may be invoked again to restart an execution, for example, in the *-chart model (see the FSM domain). The prefire(), fire(), and postfire() methods will usually be invoked many times. The fire() method may be invoked several times between invocations of prefire() and postfire(). The stopFire() method is invoked to request suspension of firing. The wrapup() method will be invoked exactly once per execution, at the end of the execution.

The terminate() method is provided as a last-resort mechanism to interrupt execution based on an external event. It is not called during the normal flow of execution. It should be used only to stop runaway threads that do not respond to more usual mechanism for stopping an execution.

An *iteration* is defined to be one invocation of prefire(), any number of invocations of fire(), and one invocation of postfire(). An *execution* is defined to be one invocation of preinitialize(), followed by one invocation of initialize(), followed by any number of iterations, followed by one invocation of wrapup(). The methods preinitialize(), initialize(), prefire(), fire(), postfire(), and wrapup() are called the *action methods*. While, the action methods in the executable interface are executed in order during the normal flow of an iteration, the terminate() method can be executed at any time, even during the execution of the other methods.

The preinitialize() method of each actor gets invoked exactly once. Typical actions of the preinitialize() method include creating receivers and defining the types of the ports. Higher-order function actors should construct their models in this method. The preinitialize() method cannot produce output data since type resolution is typically not yet done. It also gets invoked prior to any static scheduling that might occur in the domain, so it can change scheduling information.

The initialize() method of each actor gets invoked once after type resolution is done. It may be invoked again to restart the execution of an actor. Typical actions of the initialize() method include creating and initializing private data members. An actor may produce output data and schedule events in this method.

The prefire() method may be invoked multiple times during an execution, but only once per iteration. The prefire() returns true to indicate that the actor is ready to fire. In other words, a return value of true indicates "you can safely invoke my fire method," while a false value from prefire means "My preconditions for firing are not satisfied. Call prefire again later when conditions have change." For example, a dynamic dataflow actor might return false to indicate that not enough data is available on the input ports for a meaningful firing to occur.

The fire() method may be invoked multiple times during an iteration. In most domains, this method defines the computation performed by the actor. Some domains will invoke fire() repeatedly until some convergence condition is reached. Thus, fire() should not change the state of the actor.

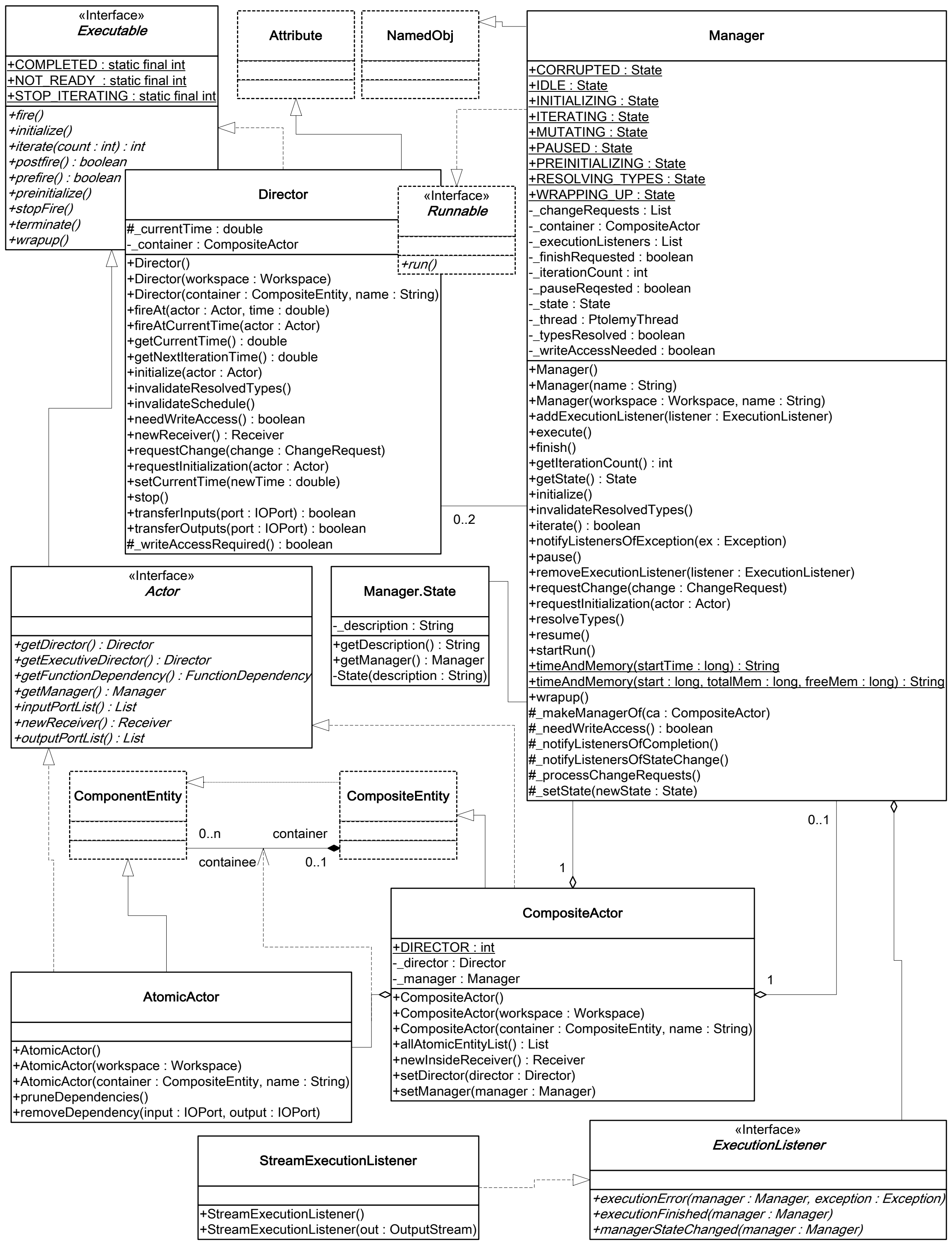
«Interface»
Executable
+COMPLETED : static final int
+NOT_READY : static final int
+STOP_ITERATING : static final int
+fire()
+initialize()
+iterate(count : int) : int
+postfire() : boolean
+prefire() : boolean
+preinitialize()
+stopFire()
+terminate()
+wrapup()
Attribute
NamedObj
Manager
+CORRUPTED : State
+IDLE : State
+INITIALIZING : State
+ITERATING : State
+MUTATING : State
+PAUSED : State
+PREINITIALIZING : State
+RESOLVING_TYPES : State
+WRAPPING_UP : State
-_changeRequests : List
-_container : CompositeActor
-_executionListeners : List
-_finishRequested : boolean
-_iterationCount : int
-_pauseReqested : boolean
-_state : State
-_thread : PtolemyThread
-_typesResolved : boolean
-_writeAccessNeeded : boolean
+Manager()
+Manager(name : String)
+Manager(workspace : Workspace, name : String)
+addExecutionListener(listener : ExecutionListener)
+execute()
+finish()
+getIterationCount() : int
+getState() : State
+initialize()
+invalidateResolvedTypes()
+iterate() : boolean
+notifyListenersOfException(ex : Exception)
+pause()
+removeExecutionListener(listener : ExecutionListener)
+requestChange(change : ChangeRequest)
+requestInitialization(actor : Actor)
+resolveTypes()
+resume()
+startRun()
+timeAndMemory(startTime : long) : String
+timeAndMemory(start : long, totalMem : long, freeMem : long) : String
+wrapup()
#_makeManagerOf(ca : CompositeActor)
#_needWriteAccess() : boolean
#_notifyListenersOfCompletion()
#_notifyListenersOfStateChange()
#_processChangeRequests()
#_setState(newState : State)
Director
#_currentTime : double
-_container : CompositeActor
+Director()
+Director(workspace : Workspace)
+Director(container : CompositeEntity, name : String)
+fireAt(actor : Actor, time : double)
+fireAtCurrentTime(actor : Actor)
+getCurrentTime() : double
+getNextIterationTime() : double
+initialize(actor : Actor)
+invalidateResolvedTypes()
+invalidateSchedule()
+needWriteAccess() : boolean
+newReceiver() : Receiver
+requestChange(change : ChangeRequest)
+requestInitialization(actor : Actor)
+setCurrentTime(newTime : double)
+stop()
+transferInputs(port : IOPort) : boolean
+transferOutputs(port : IOPort) : boolean
#_writeAccessRequired() : boolean
«Interface»
Runnable
+run()
0..2
«Interface»
Actor
+getDirector() : Director
+getExecutiveDirector() : Director
+getFunctionDependency() : FunctionDependency
+getManager() : Manager
+inputPortList() : List
+newReceiver() : Receiver
+outputPortList() : List
Manager.State
-_description : String
+getDescription() : String
+getManager() : Manager
-State(description : String)
ComponentEntity
CompositeEntity
0..n
container
containee
0..1
0..1
1
CompositeActor
+DIRECTOR : int
-_director : Director
-_manager : Manager
+CompositeActor()
+CompositeActor(workspace : Workspace)
+CompositeActor(container : CompositeEntity, name : String)
+allAtomicEntityList() : List
+newInsideReceiver() : Receiver
+setDirector(director : Director)
+setManager(manager : Manager)
1
AtomicActor
+AtomicActor()
+AtomicActor(workspace : Workspace)
+AtomicActor(container : CompositeEntity, name : String)
+pruneDependencies()
+removeDependency(input : IOPort, output : IOPort)
StreamExecutionListener
+StreamExecutionListener()
+StreamExecutionListener(out : OutputStream)
«Interface»
ExecutionListener
+executionError(manager : Manager, exception : Exception)
+executionFinished(manager : Manager)
+managerStateChanged(manager : Manager)

FIGURE 2.12. Basic classes in the actor package that support execution.

Instead, update the state in postfire().

In opaque composite actors, the fire() method is responsible for transferring data from the opaque ports of the composite actor to the ports of the contained actors, calling the fire() method of the director, and transferring data from the output ports of the composite actor to the ports of outside actors. See section 2.3.4 below.

In some domains, the fire method initiates an open-ended computation. The stopFire() method may be used to request that firing be ended and that the fire() method return as soon as practical.

The postfire() method will be invoked exactly once during an iteration, after all invocations of the fire() method in that iteration. An actor may return false in postfire to request that the actor should not be fired again. It has concluded its mission. However, a director may elect to continue to fire the actor until the conclusion of its own iteration. Thus, the request may not be immediately honored.

The wrapup() method is invoked exactly once during the execution of a model, even if an exception causes premature termination of an execution. Typically, wrapup() is responsible for cleaning up after execution has completed, and perhaps flushing output buffers before execution ends and killing active threads.

The terminate() method may be called at any time during an execution, but is not necessarily called at all. When terminate() is called, no more execution is important, and the actor should do everything in its power to stop execution right away. This method should be used as a last resort if all other mechanisms for stopping an execution fail.

2.3.1 Director

A *director* governs the execution of a composite entity. A *manager* governs the overall execution of a model. An example of the use of these classes is shown in figure 2.13. In that example, a top-level entity, E0, has an instance of Director, D1, that serves the role of its local director. A *local director* is responsible for execution of the components within the composite. It will perform any scheduling that might be necessary, dispatch threads that need to be started, generate code that needs to be generated, etc. In the example, D1 also serves as an executive director for E2. The *executive director* associated with an actor is the director that is responsible for firing the actor.

A composite actor that is not at the top level may or may not have its own local director. If it has a local director, then it defined to be opaque (isOpaque() returns *true*). In figure 2.13, E2 has a local director and E3 does not. The contents of E3 are directly under the control of D1, as if the hierarchy

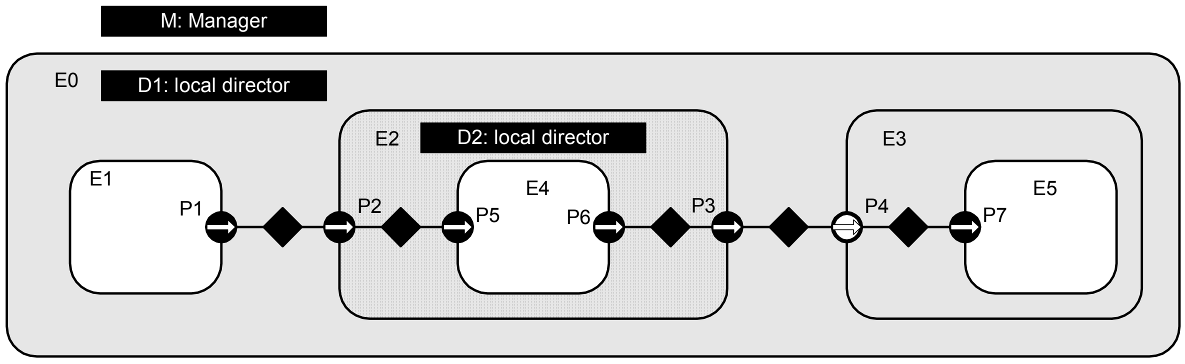

FIGURE 2.13. Example application, showing a typical arrangement of actors, directors, and managers.

were flattened. By contrast, the contents of E2 are under the control of D2, which in turn is under the control of D1. In the terminology of the previous generation, Ptolemy Classic, E2 was called a *wormhole*. In Ptolemy II, we simply call it a opaque composite actor. It will be explained in more detail below in section 2.3.4.

We define the *director* (vs. local director or executive director) of an actor to be either its local director (if it has one) or its executive director (if it does not). A composite actor that is not at the top level has as its executive director the director of the container. Every executable actor has a director except the top-level composite actor, and that director is what is returned by the getDirector() method of the Actor interface (see figure 2.12).

When any action method is called on an opaque composite actor, the composite actor will generally call the corresponding method in its local director. This interaction is crucial, since it is domain-independent and allows for communication between different models of computation. When fire() is called in the director, the director is free to invoke iterations in the contained topology until the stopping condition for the model of computation is reached.

The postfire() method of a director returns false to stop its execution normally. It is the responsibility of the next director up in the hierarchy (or the manager if the director is at the top level) to conclude the execution of this director by calling its wrapup() method.

The Director class provides a default implementation of an execution, although specific domains may override this implementation. In order to ensure interoperability of domains, they should stick fairly closely to the sequence.

Two common sequences of method calls between actors and directors are shown in figure 2.14 and 2.15. These differ in the shaded areas, which define the domain-specific sequencing of actor firings. In figure 2.14, the fire() method of the director selects an actor, invokes its prefire() method, and if that returns true, invokes its fire() method some number of times (domain dependent) followed by its postfire() method. In figure 2.15, the fire() method of the director invokes the prefire() method of all the actors before invoking any of their fire() methods.

When a director is initialized, via its initialize() method, it invokes initialize() on all the actors in the next level of the hierarchy, in the order in which these actors were created. The wrapup() method works in a similar way, *deeply* traversing the hierarchy. In other words, calling initialize() on a composite actor is guaranteed to initialize in all the objects contained within that actor. Similarly for wrapup().

The methods prefire() and postfire(), on the other hand, are not deeply traversing functions. Calling prefire() on a director does not imply that the director call prefire() on all its actors. Some directors may need to call prefire() on some or all contained actors before being able to return, but some directors may not need to call prefire() on any contained objects at all. A director may even implement short-circuit evaluation, where it calls prefire() on only enough of the contained actors to determine its own return value. Postfire() works similarly, except that it may only be called after at least one successful call to fire().

The fire() method is where the bulk of work for a director occurs. When a director is fired, it has complete control over execution, and may initiate whatever iterations of other actors are appropriate for the model of computation that it implements. It is important to stress that once a director is fired, outside objects do not have control over when the iteration will complete. The director may not iterate any contained actors at all, or it may iterate the contained actors forever, and not stop until terminate() is called. Of course, in order to promote interoperability, directors should define a finite execution that they perform in the fire() method.

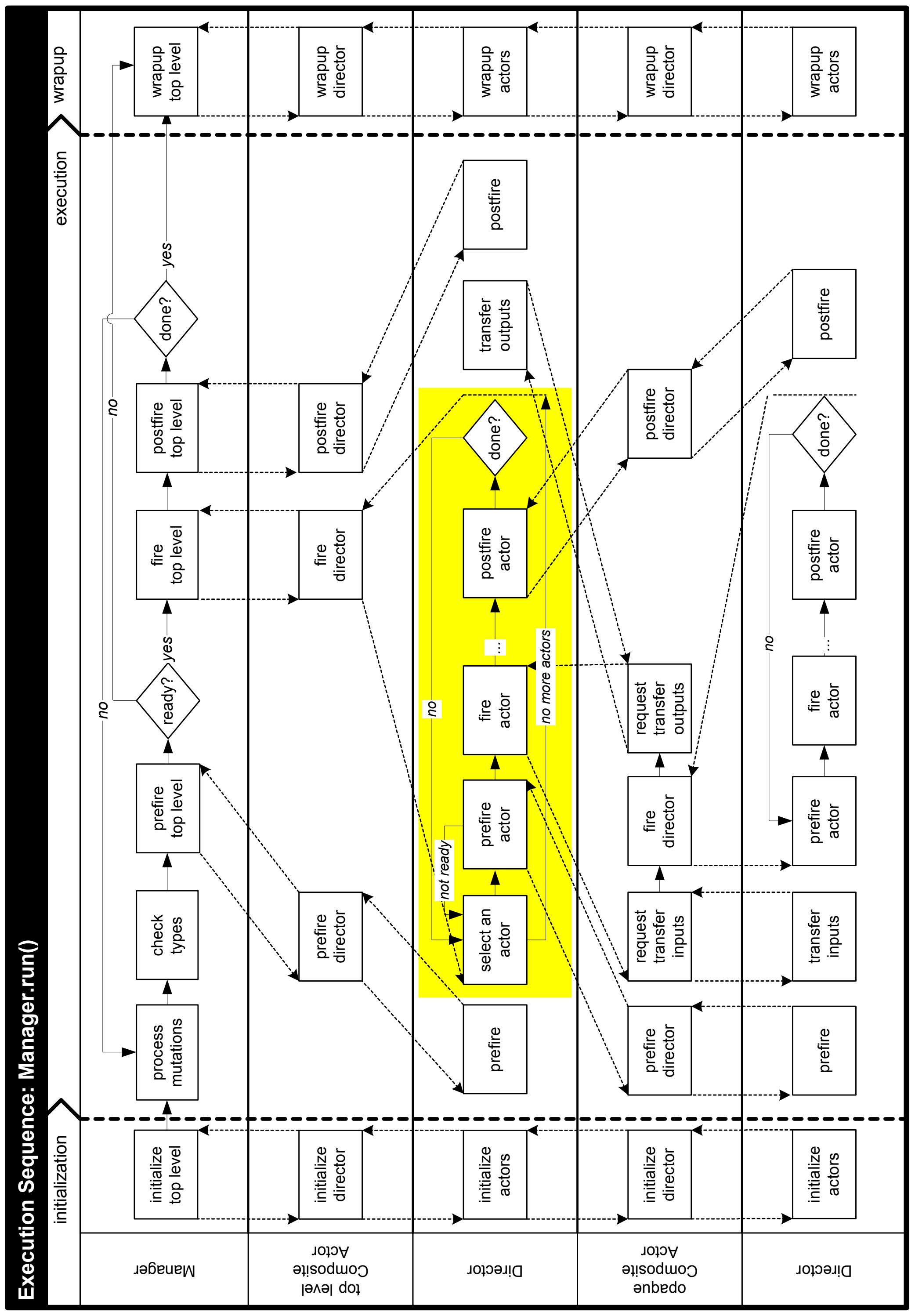

FIGURE 2.14. Example execution sequence implemented by run() method of the Director class.

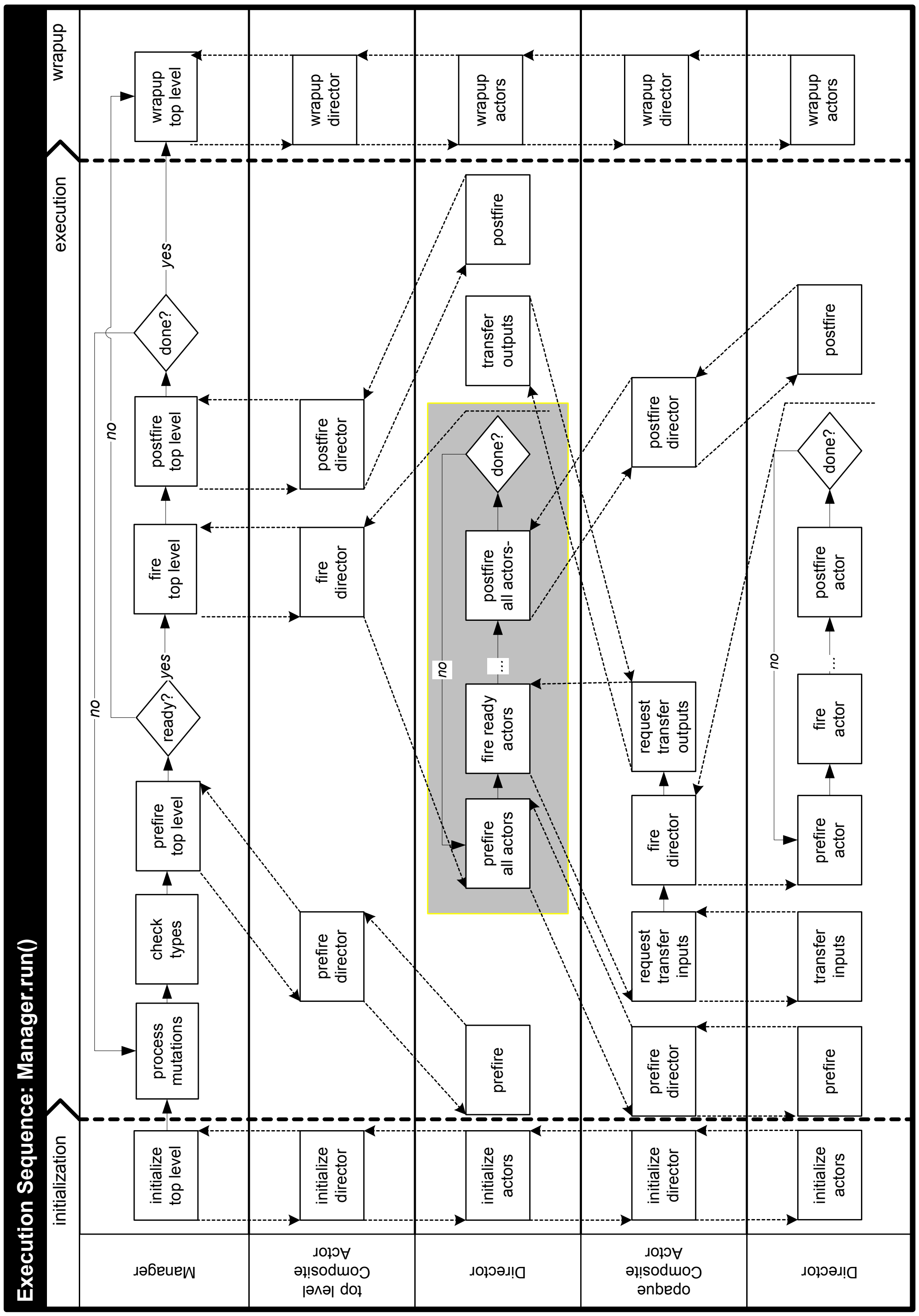

FIGURE 2.15. Alternative execution sequence implemented by run() method of the Director class.

In case it is not practical for the fire() method to define a bounded computation, the stopFire() method is provided. A director should respond when this method is called by returning from its fire() method as soon as practical.

In some domains, the firing of a director corresponds exactly to the sequential firing of the contained actors in a specific predetermined order. This ordering is known as a *static schedule* for the actors. Some domains support this style of execution. There is also a family of domains where actors are associated with threads.

2.3.2 Manager

While a director implements a model of computation, a *manager* controls the overall execution of a model. The manager interacts with a single composite actor, known as a *top level composite actor*. The Manager class is shown in figure 2.12. Execution of a model is implemented by three methods, execute(), run() and startRun(). The startRun() method spawns a thread that calls run(), and then immediately returns. The run() method calls execute(), but catches all exceptions and reports them to listeners (if there are any) or to the standard output (if there are no listeners).

More fine grain control over the execution can be achieved by calling initialize(), iterate(), and wrapup() on the manager directly. The execute() method, in fact, calls these, repeating the call to iterate() until it returns false. The iterate method invokes prefire(), fire() and postfire() on the top-level composite actor, and returns false if the postfire() in the top-level composite actor returns false.

An execution can also be ended by calling terminate() or finish() on the manager. The terminate() method triggers an immediate halt of execution, and should be used only if other more graceful methods for ending an execution fail. It will probably leave the model in an inconsistent state, since it works by unceremoniously killing threads. The finish() method allows the system to continue until the end of the current iteration in the top-level composite actor, and then invokes wrapup(). Finish() encourages actors to end gracefully by calling their stopFire() method.

Execution may also be paused between top-level iterations by calling the pause() method. This method sets a flag in the manager and calls stopFire() on the top-level composite actor. After each top-level iteration, the manager checks the flag. If it has been set, then the manager will not start the next top-level iteration until after resume() is called. In certain domains, such as the process networks domain, there is not a very well defined concept of an iteration. Generally these domains do not rely on repeated iteration firings by the manager. The call to stopFire() requests of these domains that they suspend execution.

2.3.3 ExecutionListener

The ExecutionListener interface provides a mechanism for a manager to report events of interest to a user interface. Generally a user interface will use the events to notify the user of the progress of execution of a system. A user interface can register one or more ExecutionListeners with a manager using the method addExecutionListener() in the Manager class. When an event occurs, the appropriate method will get called in all the registered listeners.

Two kinds of events are defined in the ExecutionListener interface. A listener is notified of the completion of an execution by the executionFinished() method. The executionError() method indicates that execution has ended with an error condition. The managerStateChanged() indicates to the listener that the manager has changed state. The new state can be obtained by calling getState() on the manager.

A default implementation of the ExecutionListener interface is provided in the StreamExecution-

Listener class. This class reports all events on the standard output.

2.3.4 Opaque Composite Actors

One of the key features of Ptolemy II is its ability to hierarchically mix models of computation in a disciplined way. The way that it does this is to have actors that are composite (non-atomic) and opaque. Such an actor was called a *wormhole* in the earlier generation of Ptolemy. Its ports are opaque and its contents are not visible via methods like deepEntityList().

Recall that an instance of CompositeActor that is at the top level of the hierarchy must have a local director in order to be executable. A CompositeActor at a lower level of the hierarchy may also have a local director, in which case, it is opaque (isOpaque() returns *true*). It also has an executive director, which is simply the director of its container. For a composite opaque actor, the local director and executive director need not follow the same model of computation. Hence hierarchical heterogeneity.

The ports of a composite opaque actor are opaque, but it is a composite (it can contain actors and relations). This has a number of implications on execution. Consider the simple example shown in figure 2.16. Assume that both E0 and E2 have local directors (D1 and D2), so E2 is opaque. The ports of E2 therefore are opaque, as indicated in the figure by their solid fill. Since its ports are opaque, when a token is sent from the output port P1, it is deposited in P2, not P5.

In the execution sequences of figures 2.14 and 2.15, E2 is treated as an atomic actor by D1; i.e. D1 acts as an executive director to E2. Thus, the fire() method of D1 invokes the prefire(), fire(), and postfire() methods of E1, E2, and E3. The fire() method of E2 is responsible for transferring the token from P2 to P5. It does this by delegating to its local director, invoking its transferInputs() method. It then invokes the fire() method of D2, which in turn invokes the prefire(), fire(), and postfire() methods of E4.

During its fire() method, E2 will invoke the fire() method of D2, which typically will fire the actor E4, which may send a token via P6. Again, since the ports of E2 are opaque, that token goes only as far as P3. The fire() method of E2 is then responsible for transferring that token to P4. It does this by delegating to its *executive* director, invoking its transferOutputs() method.

The CompositeActor class delegates transfer of its inputs to its local director, and transfer of its outputs to its executive director. This is the correct organization, because in each case, the director appropriate to the model of computation of the destination port is the one handling the transfer. It can therefore handle it in a manner appropriate to the receiver in that port.

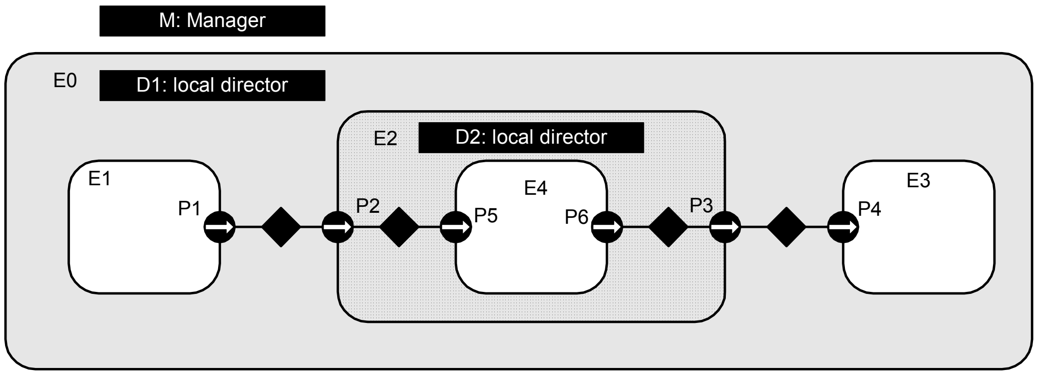

FIGURE 2.16. An example of an opaque composite actor. E0 and E2 both have local directors, not necessarily implementing the same model of computation.

Note that the port P3 is an output, but it has to be capable of receiving data from the inside, as well as sending data to the outside. Thus, despite being an output, it contains a receiver. Such a receiver is called an *inside receiver*. The methods of IOPort offer only limited access to the inside receivers (only via the getInsideReceivers() method and getReceivers(*relation*), where *relation* is an inside linked relation).

In general, a port may be both an input and an output. An opaque port of a composite opaque actor, thus, must be capable of storing two distinct types of receivers, a set appropriate to the inside model of computation, obtained from the local director, and a set appropriate to the outside model of computation, obtained from its executive director. Most methods that access receivers, such as hasToken() or hasRoom(), refer only to the outside receivers. The use of the inside receivers is rather specialized, only for handling composite opaque actors, so a more basic interface is sufficient.

2.4 Scheduler and Process Support

The ptolemy.actor.util package shown in figure 2.10 provides some infrastructure for domain designers by supporting efficient queues and dependency analysis. In addition, the actor package has two other subpackages, actor.sched, which provides rudimentary support for domains that use static schedulers to control the invocation of actors, and actor.process, which provides support for domains where actors are processes. The UML diagrams for these are shown in figure 2.17 and figure 2.18. This section describes some of this infrastructure.

2.4.1 Function Dependency

The FunctionDependency class and its subclasses in figure 2.10 provides support for domains that analyze data dependencies for scheduling or for checking correctness. In particular, an instance of FunctionDependency is associated with every actor and can be obtained from the getFunctionDependecy() method of the Actor interface (see figure 2.12). The instance of FunctionDependency describes the *function dependency* that output ports of the associated actor have on its input ports. An output port has a function dependency on an input port if in its fire() method, it sends tokens on the output port that depend on tokens gotten from the input port.

The FunctionDependency class uses a graph to describe the function dependency, where the nodes of the graph correspond to the ports and an edge indicates a function dependency. The edges go from input ports to output ports that depend on them. For atomic actors, this function dependency graph by default indicates that each output port depends on all input ports (this is called *complete dependency*). For some atomic actors, such as the TimedDelay actor in the DE domain, an output in a firing does not depend on an input port. Such actors override the pruneDependencies() method of AtomicActor (see figure 2.12) to remove dependencies between these ports. For example, the TimedDelay actor of the DE domain declares that its *output* port is independent of its *input* port by defining this method:

```
public void pruneDependencies() {
    super.pruneDependencies();
    removeDependency(input, output);
}
```

For composite actors, getFunctionDependency() returns an instance of FunctionDependencyOfCompositeActor (see figure 2.10). This class provides both the abstracted view, which gives the function dependency that output ports of the actor have on its input ports, and a *detailed view* from which it constructs this information. The detailed view is a graph where the nodes correspond to the ports of the composite actor *and* to the ports of all deeply contained opaque actors, and the edges represent either the communication dependencies implied by the connections within the composite actor or the function dependencies of the contained opaque actors. The detailed view can be used by a director to construct a schedule. Also, the detailed view may reveal dependency loops, which in many domains means that the model cannot be executed. To check whether there are such loops, use the getCycleNodes() method. The method returns an array of IOPorts in such loops, of an empty array if there are no such loops.

2.4.2 Statically Scheduled Domains

The StaticSchedulingDirector class extends the Director base class to add a scheduler. The scheduler (an instance of the Scheduler class) creates an instance of the Schedule class which represents a statically determined sequence of actor firings. The scheduler also caches the schedule as necessary

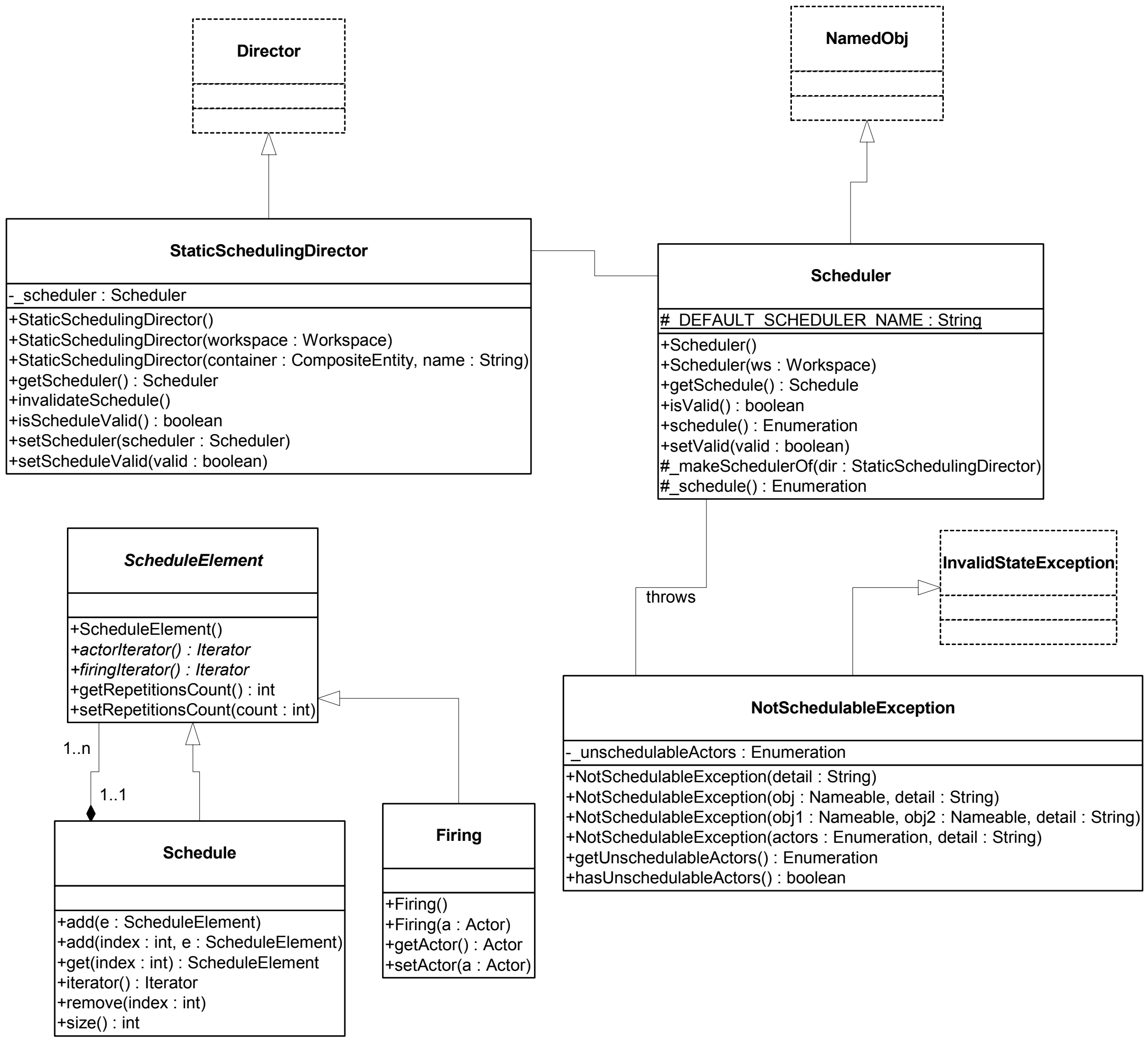
Director
NamedObj
StaticSchedulingDirector
-_scheduler : Scheduler
+StaticSchedulingDirector()
+StaticSchedulingDirector(workspace : Workspace)
+StaticSchedulingDirector(container : CompositeEntity, name : String)
+getScheduler() : Scheduler
+invalidateSchedule()
+isScheduleValid() : boolean
+setScheduler(scheduler : Scheduler)
+setScheduleValid(valid : boolean)
Scheduler
#_DEFAULT_SCHEDULER_NAME : String
+Scheduler()
+Scheduler(ws : Workspace)
+getSchedule() : Schedule
+isValid() : boolean
+schedule() : Enumeration
+setValid(valid : boolean)
#_makeSchedulerOf(dir : StaticSchedulingDirector)
#_schedule() : Enumeration
ScheduleElement
+ScheduleElement()
+actorIterator() : Iterator
+firingIterator() : Iterator
+getRepetitionsCount() : int
+setRepetitionsCount(count : int)
throws
InvalidStateException
NotSchedulableException
-_unschedulableActors : Enumeration
+NotSchedulableException(detail : String)
+NotSchedulableException(obj : Nameable, detail : String)
+NotSchedulableException(obj1 : Nameable, obj2 : Nameable, detail : String)
+NotSchedulableException(actors : Enumeration, detail : String)
+getUnschedulableActors() : Enumeration
+hasUnschedulableActors() : boolean
1..n
1..1
Schedule
+add(e : ScheduleElement)
+add(index : int, e : ScheduleElement)
+get(index : int) : ScheduleElement
+iterator() : Iterator
+remove(index : int)
+size() : int
Firing
+Firing()
+Firing(a : Actor)
+getActor() : Actor
+setActor(a : Actor)

FIGURE 2.17. UML static structure diagram for the actor.sched package.

until it is invalidated by the director. This means that domains with a statically determined schedule (such as CT and SDF) need only implement the action methods in the director and a scheduler with the appropriate scheduling algorithm.

The Schedule base class contains a list of schedule elements, each with a repetitions factor that determines the number of times that element will be repeated. Since a schedule itself is a schedule ele-

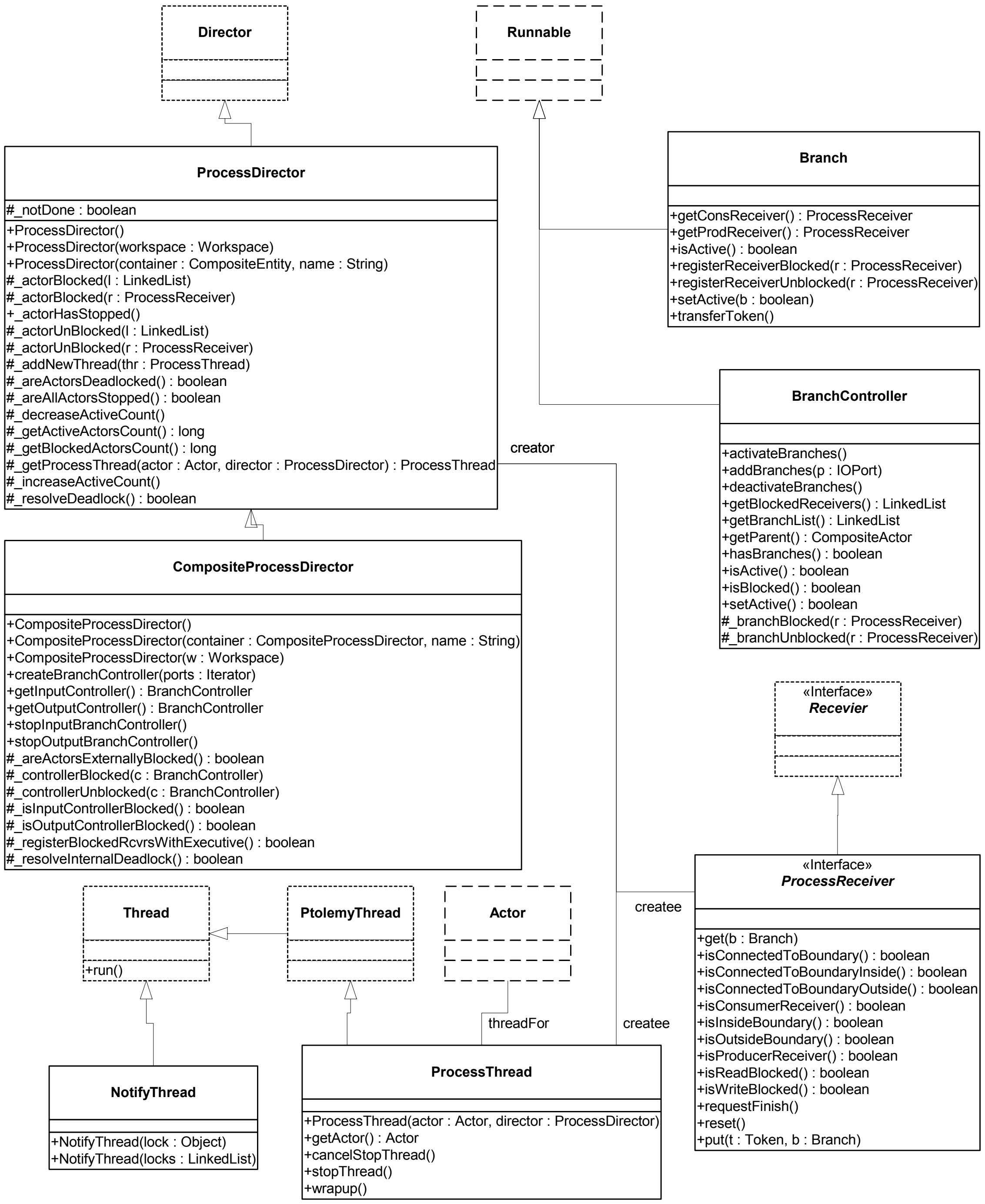
Director
Runnable
ProcessDirector
#_notDone : boolean
+ProcessDirector()
+ProcessDirector(workspace : Workspace)
+ProcessDirector(container : CompositeEntity, name : String)
#_actorBlocked(l : LinkedList)
#_actorBlocked(r : ProcessReceiver)
+_actorHasStopped()
#_actorUnBlocked(l : LinkedList)
#_actorUnBlocked(r : ProcessReceiver)
#_addNewThread(thr : ProcessThread)
#_areActorsDeadlocked() : boolean
#_areAllActorsStopped() : boolean
#_decreaseActiveCount()
#_getActiveActorsCount() : long
#_getBlockedActorsCount() : long
#_getProcessThread(actor : Actor, director : ProcessDirector) : ProcessThread
#_increaseActiveCount()
#_resolveDeadlock() : boolean
Branch
+getConsReceiver() : ProcessReceiver
+getProdReceiver() : ProcessReceiver
+isActive() : boolean
+registerReceiverBlocked(r : ProcessReceiver)
+registerReceiverUnblocked(r : ProcessReceiver)
+setActive(b : boolean)
+transferToken()
BranchController
+activateBranches()
+addBranches(p : IOPort)
+deactivateBranches()
+getBlockedReceivers() : LinkedList
+getBranchList() : LinkedList
+getParent() : CompositeActor
+hasBranches() : boolean
+isActive() : boolean
+isBlocked() : boolean
+setActive() : boolean
#_branchBlocked(r : ProcessReceiver)
#_branchUnblocked(r : ProcessReceiver)
creator
CompositeProcessDirector
+CompositeProcessDirector()
+CompositeProcessDirector(container : CompositeProcessDirector, name : String)
+CompositeProcessDirector(w : Workspace)
+createBranchController(ports : Iterator)
+getInputController() : BranchController
+getOutputController() : BranchController
+stopInputBranchController()
+stopOutputBranchController()
#_areActorsExternallyBlocked() : boolean
#_controllerBlocked(c : BranchController)
#_controllerUnblocked(c : BranchController)
#_isInputControllerBlocked() : boolean
#_isOutputControllerBlocked() : boolean
#_registerBlockedRcvrsWithExecutive() : boolean
#_resolveInternalDeadlock() : boolean
«Interface»
Recevier
Thread
+run()
PtolemyThread
Actor
createe
«Interface»
ProcessReceiver
+get(b : Branch)
+isConnectedToBoundary() : boolean
+isConnectedToBoundaryInside() : boolean
+isConnectedToBoundaryOutside() : boolean
+isConsumerReceiver() : boolean
+isInsideBoundary() : boolean
+isOutsideBoundary() : boolean
+isProducerReceiver() : boolean
+isReadBlocked() : boolean
+isWriteBlocked() : boolean
+requestFinish()
+reset()
+put(t : Token, b : Branch)
threadFor
createe
NotifyThread
+NotifyThread(lock : Object)
+NotifyThread(locks : LinkedList)
ProcessThread
+ProcessThread(actor : Actor, director : ProcessDirector)
+getActor() : Actor
+cancelStopThread()
+stopThread()
+wrapup()

FIGURE 2.18. UML static structure diagram for the actor.process package.

ment, schedules can be defined recursively. Another type of schedule element is a firing, which represents a firing of a single actor. An iterator over all firings contained by a schedule is returned by the firingIterator() method on the schedule. In the iterator, the schedule is expanded recursively, with each firing repeated the appropriate number of times.[1]

2.4.3 Process Domains

Many domains, such as CSP, PN and DDE, consist of independent processes that are communicating in some way. These domains are collectively termed *process domains*. The actor.process package provides the following base classes that can be used to implement process domains.

ProcessThread. In a process domain, each actor represents an independently executing process. In Ptolemy II, this is achieved by creating a separate Java thread for each actor [113][70]. Each of these threads is an instance of ptolemy.actor.ProcessThread.

The thread for each actor is started in the prefire() method of the director. After starting, this thread repeatedly calls the prefire(), fire(), and postfire() methods of its associated actor. This sequence continues until the actor's postfire() method of returns false. The only way for an actor to terminate gracefully in PN is by returning from its fire() method and then returning false in its postfire() method. If an actor finishes execution as above, then the thread calls the wrapup() method of the actor. Once this method returns, the thread informs the director about the termination of this actor and finishes its own execution. The actor will not be fired again unless the director creates and starts a new thread for the actor.

ProcessReceiver. In the process domains, receivers represent the communication and synchronization points between different threads. To facilitate creating these domains, receivers in process domains should implement the ProcessReceiver interface. This interface provides extended information about status of the receiver, and the threads that may be interacting with the receiver.

ProcessDirector and CompositeProcessDirector. These classes are base classes for directors in the process-based domains. It provides some basic infrastructure for creating and managing threads. Most importantly, it provides a strategy pattern for handling deadlock between threads. Subclasses usually override methods in this class to handle deadlock in a domain-dependent fashion. In order to detect deadlocks, this base class maintains a count of how many actors in the system are executing and how many are blocked for some reason. This method of detecting deadlock is suggested in [69]. When no threads are able to run, the director calls the _resolveDeadlock() method to attempt to resolve the deadlock.

The initialize() method of the process director creates the receivers in the input ports of the actors, creates a thread for each actor and initializes these actors. It also initializes the count of active actors in the model to the number of actors in the composite actor. The prefire() method starts up all the created threads. This method returns true by default. The fire() method of a process director does not actually fire any contained actors. Instead, each actor is iterated by its associated process thread. The fire method simply blocks the calling thread until deadlock of the process threads occurs. In this case, the calling thread is unblocked and the fire method returns. The postfire() method simply returns true if the director was able to resolve the deadlock at the end of the fire method, or false otherwise. Returning true implies that if some new data is provided to the composite actor it can resume execution. Return-

1. Note that creating an iterator does not require expanding the data structure of the schedule into a list first.

ing false implies that this composite actor will not be fired again. In that case, the executive director or the manager will call the wrapup() method of the top-level composite actor, which in turn calls the wrapup() method of the director. This causes the director to terminate the execution of the composite actor.

Introduction to Java Threads. The process domains, like the rest of Ptolemy II, are written entirely in Java and take advantage of the features built into the language. In particular, they rely heavily on threads and on monitors for controlling the interaction between threads. In any multi-threaded environment, care has to be taken to ensure that the threads do not interact in unintended ways, and that the model does not deadlock. Note that deadlock in this sense is a bug in the *modeling environment*, which is different from the deadlock talked about before which may or may not be a bug in the model being executed.

A monitor is a mechanism for ensuring mutual exclusion between threads. In particular if a thread has a particular monitor, acquired in order to execute some code, then no other thread can simultaneously have that monitor. If another thread tries to acquire that monitor, it stalls until the monitor becomes available. A monitor is also called a *lock*, and one is associated with every object in Java.

Code that is associated with a lock is defined by the *synchronized* keyword. This keyword can either be in the signature of a method, in which case the entire method body is associated with that lock, or it can be used in the body of a method using the syntax:

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

This causes the code inside the brackets to be associated with the lock belonging to the specified object. In either case, when a thread tries to execute code controlled by a lock, it must either acquire the lock or stall until the lock becomes available. If a thread stalls when it already has some locks, those locks are not released, so any other threads waiting on those locks cannot proceed. This can lead to deadlock when all threads are stalled waiting to acquire some lock they need.

A thread can voluntarily relinquish a lock when stalling by calling *object.*wait() where *object* is the object to relinquish and wait on. This causes the lock to become available to other threads. A thread can also wake up any threads waiting on a lock associated with an object by calling notifyAll() on the object. Note that to issue a notifyAll() on an object it is necessary to own the lock associated with that object first. By careful use of these methods it is possible to ensure that threads only interact in intended ways and that deadlock does not occur.

Approaches to locking used in the process domains. One of the key coding patterns followed is to wrap each wait() call in a while loop that checks some flag. Only when the flag is set to false can the thread proceed beyond that point. Thus the code will often look like

```
synchronized(object) {
  ...
  while(flag) {
    object.wait();
  }
  ...
}
```

The advantage to this is that it is not necessary to worry about what other thread issued the notifyAll() on the lock; the thread can only continue when the notifyAll() is issued *and* the flag has been set to false.

One place that contention between threads often occurs is when a thread tries to acquire another lock only to issue a notifyAll() on it. To reduce the contention, it often easiest if the notifyAll() is issued from a new thread which has no locks that could be held if it stalls. This is often used in the CSP domain to wake up any threads waiting on receivers after a pause or when terminating the model. The `ptolemy.actor.process.NotifyThread` class can be used for this purpose. This class takes a list of objects in a linked list, or a single object, and issues a notifyAll() on each of the objects from within a new thread.

Synchronization Hierarchy. Previously we have discussed how model deadlock is resolved in process domains. Separate from these notions is a different kind of deadlock that can occur in a modeling environment if the environment is not designed properly. This notion of deadlock can occur if a system is not *thread safe*. Given the extensive use of Java threads throughout Ptolemy II, great care has been taken to ensure thread safety; we want no *bugs* to exist that might lead to deadlock based on the structure of the Ptolemy II modeling environment. Ptolemy II uses monitors to guarantee thread safety. A *monitor* is a method for ensuring mutual exclusion between threads that both have access to a given portion of code. To ensure mutual exclusion, threads must acquire a monitor (or *lock*) in order to access a given portion of code. While a thread owns a lock, no other threads can access the corresponding code.

There are several objects that serve as locks in Ptolemy II. In the process domains, there are four primary objects upon which locking occurs: Workspace, ProcessReceiver, ProcessDirector and AtomicActor. The danger of having multiple locks is that separate threads can acquire the locks in competing orders and this can lead to deadlock. A simple illustration is shown in figure 2.19. Assume that both lock *A* and lock *B* are necessary to perform a given set of operations and that both thread 1 and thread 2 want to perform the operations. If thread 1 acquires *A* and then attempts to acquire *B* while thread 2 does the reverse, then deadlock can occur.

There are several ways to avoid the above problem. One technique is to combine locks so that large sets of operations become atomic. Unfortunately this approach is in direct conflict with the whole purpose behind multi-threading. As larger and larger sets of operations utilize a single lock, the limit of the corresponding concurrent program is a sequential program!

Another approach is to adhere to a hierarchy of locks. A hierarchy of locks is an agreed upon order in which locks are acquired. In the above case, it may be enforced that lock *A* is always acquired before lock *B*. A hierarchy of locks will guarantee thread safety [70].

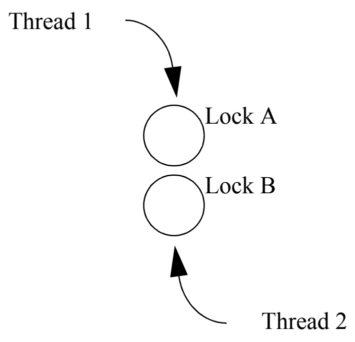

FIGURE 2.19. Deadlock Due to Unordered Locking.

The process domains have an unenforced hierarchy of locks. It is strongly suggested that users of Ptolemy II process domains adhere to this suggested locking hierarchy. The hierarchy specifies that locks be acquired in the following order:

$$\begin{tikzcd}\textbf{Workspace} \arrow[r] & \textbf{ProcessReceiver} \arrow[r] & \textbf{ProcessDirector} \arrow[r] & \textbf{AtomicActor}\end{tikzcd}$$

The way to apply this rule is to prevent synchronized code in any of the above objects from making a call to code that is to the left of the object in question.

There is one further rule that implementors of process domains should be aware of. A thread should give up all the read permissions on the workspace before calling the wait() method on the receiver object. This commonly happens in the get() and put() methods of process receivers, which implement the synchronization between threads. We require this because of the explicit modeling of mutual exclusion between the read and write activities on the workspace. If a thread holds read permission on the workspace and suspends while a second thread requires a write access on the workspace before performing the action that the first thread is waiting for, a deadlock results. Furthermore, a thread must also regain those read accesses after returning from the call to the wait() method. For this a wait(Object object) method is provided in the class Workspace that releases read accesses on the workspace, calls wait() on the argument object, and regains read access on the workspace before returning.