Chapter 1 from: C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, H. Zheng "Heterogeneous Concurrent Modeling and Design in Java (Volume 3: Ptolemy II Domains)," Technical Memorandum UCB/ERL M04/17, University of California, Berkeley, CA USA 94720, June 24, 2004.

1

DE Domain

Authors: *Lukito Muliadi*
Winthrop Williams
Edward A. Lee

1.1 Introduction

The discrete-event (DE) domain supports time-oriented models of systems such as queueing systems, communication networks, and digital hardware. In this domain, actors communicate by sending *events*, where an event is a data value (a token) and a *time stamp*. A DE scheduler ensures that events are processed chronologically according to this time stamp by firing those actors whose available input events are the oldest (having the earliest time stamp of all pending events).

A key strength in our implementation is that simultaneous events (those with identical time stamps) are handled systematically and deterministically. Another second key strength is that the global event queue uses an efficient structure that minimizes the overhead associated with maintaining a sorted list with a large number of events.

1.1.1 Model Time

In the DE model of computation, time is *global*, in the sense that all actors share the same global time. The *current time* of the model is often called the *model time* or *simulation time* to avoid confusion with current real time.

As in most Ptolemy II domains, actors communicate by sending tokens through ports. Ports can be input ports, output ports, or both. Tokens are sent by an output port and received by all input ports connected to the output port through relations. When a token is sent from an output port, it is packaged as an event and stored in a global event queue. By default, the time stamp of an output is the model time, although specialized DE actors can produce events with future time stamps.

Actors may also request that they be fired now, or at some time in the future, by calling the fireAtCurrentTime(), fireAt(), or fireAtRelativeTime() methods of the director. Each of these places a *pure*

event (one with a time stamp, but no data) on the event queue. A pure event can be thought of as setting an alarm clock to be awakened in the future. Sources (actors with no inputs) are thus able to be fired despite having no inputs to trigger a firing. Moreover, actors that introduce *delay* (outputs have larger time stamps than the inputs) can use this mechanism to schedule a firing in the future to produce an output. The fireAtCurrentTime() method provides a mechanism for achieving a *zero delay* by atomically getting the current model time and queuing an event with that time stamp. This permits I/O actors to have themselves fired in real-time whenever data arrives at a physical I/O port.

In the global event queue, events are sorted based on their time stamps. An event is removed from the global event queue when the *model time* reaches its time stamp, and if it has a data token, then that token is put into the destination input port.

At any point in the execution of a model, the events stored in the global event queue have time stamps greater than or equal to the model time. The DE director is responsible for advancing (i.e. incrementing) the model time when all events with time stamps equal to the current model time have been processed (i.e. the global event queue only contains events with time stamps strictly greater than the current time). The current time is advanced to the smallest time stamp of all events in the global event queue.

1.1.2 Simultaneous events

An important aspect of a DE domain is the prioritizing of simultaneous events. This gives the domain a dataflow-like behavior for events with identical time stamps. It is done by assigning a *depth* to each actor and a *microstep* to each phase of execution within a given time stamp. Each depth is a non-negative integer, uniquely assigned; i.e. no two actors are assigned the same depth.

The depth of an actor determines the *priority* of events destined to that actor, relative to other events with the same time stamp and the same microstep. The highest priority events are those destined to actors with the lowest depth.

Consider the simple topology shown in figure 1.1. Assume that actor *Y* is not a delay actor, meaning that its output events have the same time stamp and microstep as its input events (this is suggested by the dotted arrow). Suppose that actor *X* produces an event with time stamp τ. That event is available at ports *B* and *D*, so the scheduler could choose to fire actors *Y* or *Z*. Which should it fire? Intuition tells us it should fire the upstream one first, *Y*, because that firing may produce another event with time stamp τ at port *D* (which is presumably a multiport). It seems logical that if actor *Z* is going to get one event on each input channel with the same time stamp, then it should see those events in the same firing. Thus, if there are simultaneous events at *B* and *D*, then the one at *B* will have higher priority.

The depths are determined by a *topological sort* of a *directed acyclic graph* (DAG) of the actors. The DAG of actors follows the topology of the graph, except when there are declared delays. Once the DAG is constructed, it is sorted topologically. This simply means that an ordering of actors is assigned

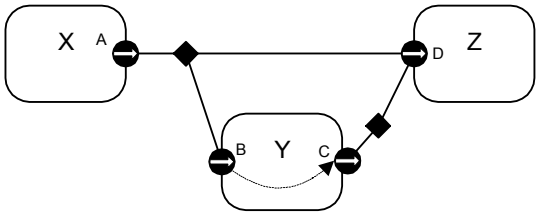

FIGURE 1.1. If there are simultaneous events at B and D, then the one at B will have higher priority because it may trigger another simultaneous event at D.

such that an upstream actor in the DAG is earlier in the ordering than a downstream actor. The depth of an actor is defined to be its position in this topological sort, starting with zero. For example, in figure 1.1, *X* will have depth 0, *Y* will have depth 1, and *Z* will have depth 2.

In general, a DAG has several correct topological sorts. The topological sort is not unique, meaning that the depths assigned to actors are somewhat arbitrary. But an upstream actor will always have a lower depth than a downstream actor, unless there is an intervening delay actor. Thus, given simultaneous input events with the same microstep, an upstream actor will always fire before a downstream actor. Such a strategy ensures that the execution is *deterministic*, assuming the actors only communicate via events. In other words, even though there are several possible choices that a scheduler could make for an ordering of firings, all choices that respect the priorities yield the same results.

There are situations where constructing a DAG following the topology is not possible. Consider the topology shown in figure 1.2. It is evident from the figure that the topology is not acyclic. Indeed, figure 1.2 depicts a *zero-delay loop* where topological sort cannot be done. The director will refuse to run the model, and will terminate with an error message.

The TimedDelay actor in DE is a domain-specific actor that asserts a delay relationship between its input and output. Thus, if we insert a TimedDelay actor in the loop, as shown in figure 1.3, then constructing the DAG becomes once again possible. The TimedDelay actor breaks the precedences. Below we will explain how you can write custom actors that have the same property.

Note in particular that the TimedDelay actor breaks the precedences *even if its delay parameter is set to zero*. Thus, the DE domain is perfectly capable of modeling feedback loops with zero time delay, but the model builder has to specify the order in which events should be processed by placing a TimedDelay actor with a zero value for its parameter. When modeling multiple zero-delay feedback paths, simultaneity of the fed back signals is modeled by having the same number of TimedDelay actors in each feedback path.

1.1.3 Iteration

At each iteration, after advancing the current time, the director chooses all events in the global event queue that have the smallest time stamps, microstep, and depth (tested in that order). The chosen

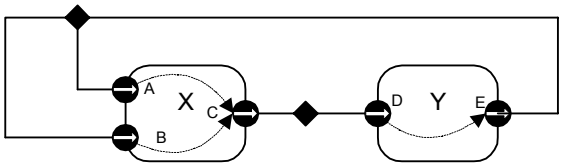

FIGURE 1.2. An example of a directed zero-delay loop.

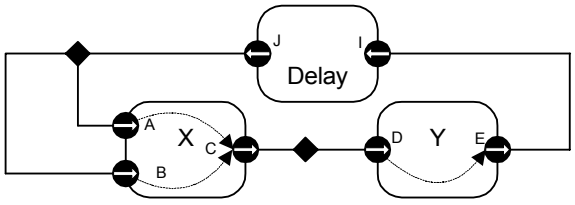

FIGURE 1.3. A Delay actor can be used to break a zero-delay loop.

events are then removed from the global event queue and their data tokens are inserted into the appropriate input ports of the destination actor. Then, the director iterates the destination actor; i.e. it invokes prefire(), fire(), and postfire(). All of these events are destined to the same actor, since the depth is unique for each actor.

A firing may produce additional events at the current model time (the actor reacts *instantaneously*, or has *zero delay*). There also may be other events with time stamp equal to the current model time still pending on the event queue. The DE director repeats the above procedure until there are no more events with time stamp equal to the current time. This concludes one iteration of the model. An iteration, therefore, processes all events on the event queue with the smallest time stamp.

1.1.4 Getting a Model Started

Before one of the iterations described above can be run, there have to be initial events in the global event queue. Actors may produce initial pure events or regular output events in their initialize() method. Thus, to get a model started, at least one actor must produce events. All the domain-polymorphic timed sources described in the Actor Libraries chapter produce pure events, so these can be used in DE. We can define the *start time* to be the smallest time stamp of these initial events.

1.1.5 Pure Events at the Current Time

An actor calls fireAt() to schedule a pure event. The pure event is a request to the scheduler to fire the actor sometime in the future. However, the actor may choose to call fireAt() with the time argument equal to the current time. In fact, the preferred method for domain-polymorphic source actors to get started is to have code like the following in their initialize() method:

```
Director director = getDirector();
director.fireAt(this, director.getCurrentTime());
```

This will schedule a pure event on the event queue with microstep zero and depth equal to that of the calling actor.

An actor may also call fireAt() with the current time in its fire() method. This is a request to be refired later *in the current iteration*. This is managed by queueing a pure event with microstep one greater than the current microstep. In fact, this is the only situation in which the microstep is incremented beyond zero.

A pure event at the current time can also be scheduled by code like the following:

```
Director director = getDirector();
director.fireAtCurrentTime(this);
```

This code is equivalent to the previous example when used within standard actor methods like initialize() and fire(). This is because the director never advances model time while an actor is being initialized or fired. However, when methods (such as an I/O callback) queue events at the current time, they need to use the latter code. This is because the director runs in a separate thread from the callback and, in the former code, will occasionally advance the model time between the call to getCurrentTime() and the call to fireAt().

1.1.6 Stopping Execution

Execution stops when one of these conditions become true:

- The current time reaches the *stop time*, set by calling the setStopTime() method of the DE director.
- The global event queue becomes empty and the *stopWhenQueueIsEmpty* parameter of the director is true.

Events at the stop time are processed before stopping the model execution. The execution ends by calling the wrapup() method of all actors. Wrapup() is called even when execution has been stopped due to an exception. Therefore, throwing an exception in the wrapup() method of an actor is not recommended as this exception will mask the original exception, making the source of the original exception difficult to locate.

It is also possible to explicitly invoke the iterate() method of the manager for some fixed number of iterations. Recall that an iteration processes all events with a given time stamp, so this will run the model through a specified number of discrete time steps.

Note that an actor can prevent execution from stopping properly if it blocks in its fire() method. An actor which blocks in fire() should have a stopFire() method which, when called, notifies the fire() method to cease blocking and return.

1.2 Overview of The Software Architecture

The UML static structure diagram for the DE kernel package is shown in figure 1.4. For model builders, the important classes are DEDirector and DEIOPort. At the heart of DEDirector is a global event queue that sorts events according to their time stamps and priorities.

The DEDirector uses an efficient implementation of the global event queue, a calendar queue data structure [20]. In theory, the time complexity for this particular implementation is O(1) in both enqueue and dequeue operations. This means that the time complexity for enqueue and dequeue operations is independent of the number of pending events in the global event queue. However, to realize this performance, it is necessary for the distribution of events to match certain assumptions. Our calendar queue implementation observes events as they are dequeued and adapts the structure of the queue according to their statistical properties. Nonetheless, the calendar queue structure will not prove optimal for all models. For extensibility, alternative implementations of the global event queue can be realized by implementing the DEEventQueue interface and specifying the event queue using the appropriate constructor for DEDirector.

The DEEvent class carries tokens through the event queue. It contains their time stamp, their microstep, and the depth of the destination actor, as well as a reference to the destination actor. It implements the java.lang.Comparable interface, meaning that any two instances of DEEvent can be compared. The private inner class DECQEventQueue.DECQComparator, which is provided to the calendar queue at the time of its construction, performs the requisite comparisons of events.

The DEIOPort class is be used by actors that are specialized to the DE domain. It supports annotations that inform the scheduler about delays through the actor. It also provides two additional methods, overloaded versions of broadcast() and send(). The overloaded versions have a second argument for the time delay, allowing actors to send output data with a time delay (relative to current time).

Domain polymorphic actors, such as those described in the Actor Libraries chapter, have as ports instances of TypedIOPort, not DEIOPort, and therefore cannot produce events in the future directly by

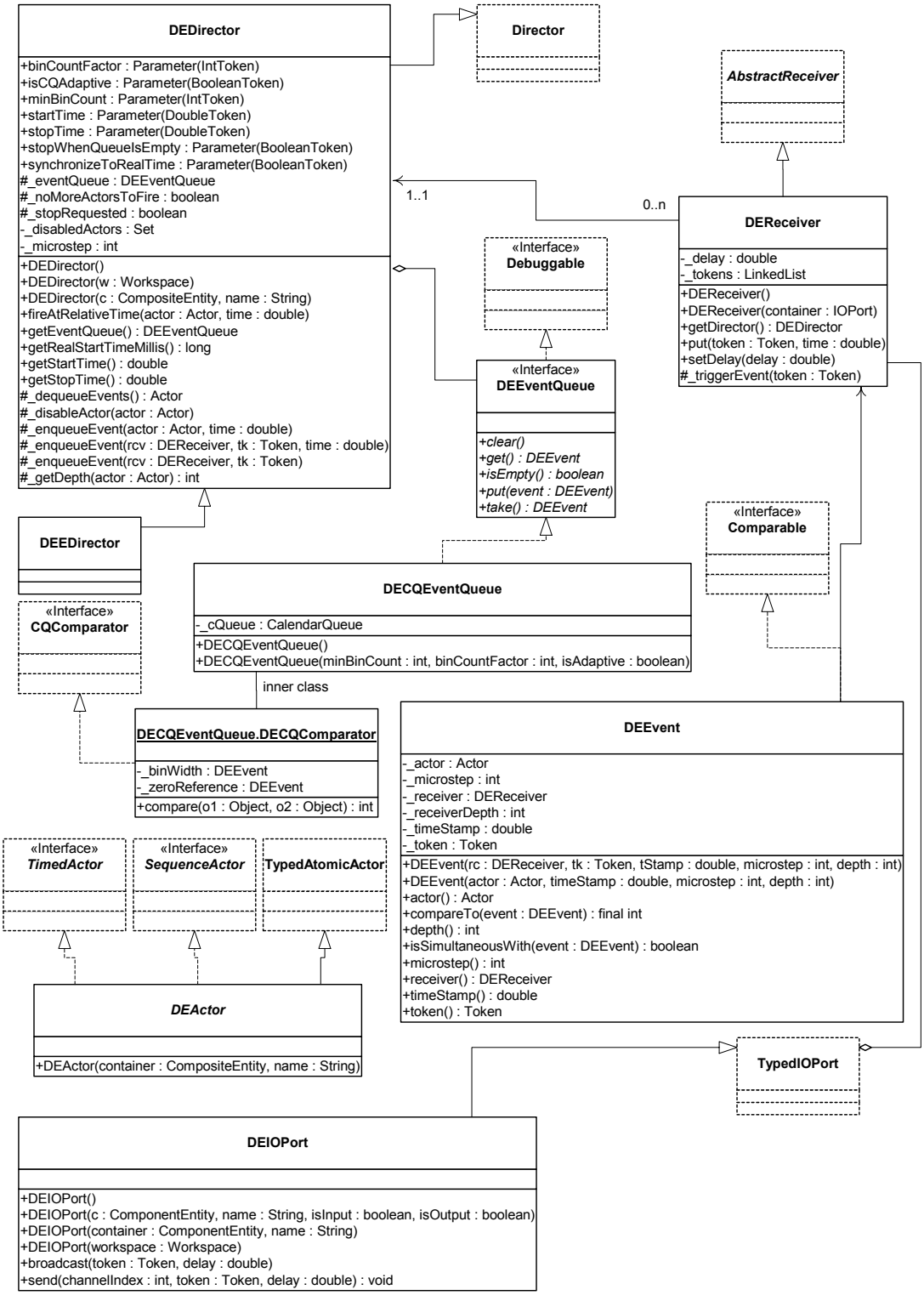

FIGURE 1.4. UML static structure diagram for the DE kernel package.

sending it through output ports. Note that tokens sent through TypedIOPort are treated as if they were sent through DEIOPort with the time delay argument equal to zero. Domain polymorphic actors can produce events in the future indirectly by using the fireAt() and fireAtRelativeTime() methods of the director. By calling fireAt() or fireAtRelativeTime(), the actor requests a refiring in the future. The actor can then produce a delayed event during the refiring.

The DEActor class, which forms the base class of many domain-specific actors, actually provides very little functionality, and there is rarely a reason to use it over TypedAtomicActor. The only functionality it adds to its base class, TypedAtomicActor, is to override the newPort() method to return an instance of DEIOPort rather than TypedIOPort. However, most actors have no use for the newPort() method because users do not dynamically create ports for them. Hence, the DEActor class can usually be ignored.

1.3 The DE Actor Library

The DE domain has a small library of actors in the ptolemy.domains.de.lib package, shown in figure 1.5. Some of them use domain-specific infrastructure, such as DEIOPort. The DETransformer base class for actors provides an input and output port that are instances of DEIOPort. The TimedDelay and Server actors influence the firing priorities as explained below by specifying function dependencies. The Merge actor merges events sequences in chronological order.

1.4 Mutations

The DE director tolerates changes to the model during execution. The change should be queued using requestChange(). While invoking those changes, the method invalidateSchedule() is expected to be called, notifying the director that the topology it used to calculate the priorities of the actors is no longer valid. This will result in the priorities being recalculated the next time prefire() is invoked.

An example of a mutation is shown in figures 1.6 and 1.7. Figure 1.7 defines a class that constructs a simple model in its constructor. The model consists of a clock connected to a recorder. The method insertClock() creates an anonymous inner class that extends ChangeRequest[1]. Its execute() method disconnects the two existing actors, creates a new clock and a merge actor, and reconnects the actors as shown in figure 1.6.

When the insertClock() method is called, a change request is queue with the top-level composite actor, which delegates the request to the manager. The manager executes the request after the current iteration completes. Thus, the change will always be executed between non-equal time stamps, since an iteration consists of processing all events at the current time stamp.

Actors that are added in the change request are automatically initialized. Note, however, one subtlety. The next to last line of the insertClock() method is:

```
_rec.input.createReceivers();
```

1. Often a more convenient way to generate mutations is to construct a MoML description of the mutation and issue a MoMLChangeRequest. We are describing here a more direct, low-level mechanism. Note that if you are using actor-oriented classes, you may need to modify this example to propagate the changes from a class definition to instances and/or subclasses, if the changes are made to a class definition. If you use MoML, the propagation is handled for you by the MoML parser.

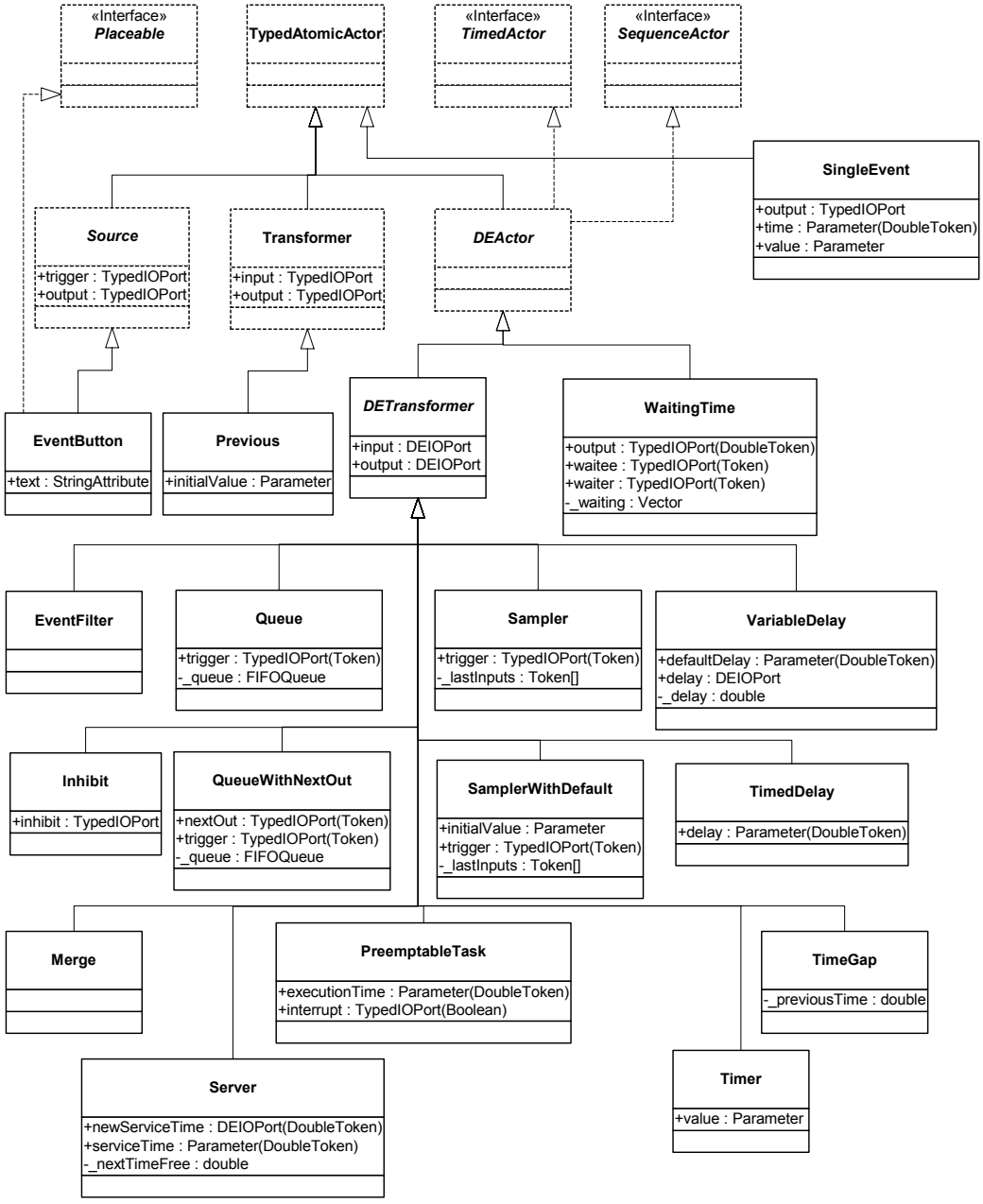

FIGURE 1.5. The library of DE-specific actors.

This method call is necessary because the connections of the recorder actor have changed, but since the actor is not new, it will *not* be reinitialized. Recall that the preinitialize() and initialize() methods are guaranteed to be called only once, and one of the responsibilities of the preinitialize() method is to create the receivers in all the input ports of an actor. Thus, whenever connections to an input port change during a mutation, the mutation code itself must call createReceivers() to reconstruct the receivers. Note that this will result in the loss of any tokens that might already be queued in the preexisting receivers of the ports. It is because of this possible loss of data that the creation of receivers is not done automatically. The designer of the mutation should be aware of the possible loss of data.

There are two additional subtleties about mutations. One involves events left on the queue and the other involves locked resources.

If an actor produces events in the future via DEIOPort, then the destination actor will be fired even if it has been removed from the topology by the time the execution reaches that future time. This may not always be the expected behavior. The Delay actor in the DE library behaves this way, so if its destination is removed before processing delayed events, then it may be invoked at a time when it has no container. Most actors will tolerate this and will not cause problems. But some might have unexpected behavior. To prevent this behavior, the mutation that removes the actor should also call the disableActor() method of the director.

If an actor locks a resource, such as an I/O port or DatagramSocket, it typically releases this resource in its wrapup() method. However, when the actor is removed while the model is executing, wrapup() never gets called. This case can be handled by overriding the setContainer() method with the following code:

```
public void setContainer(CompositeEntity container)
          throws IllegalActionException, NameDuplicationException {
     if (container != getContainer()) {
       wrapup();
     }
   super.setContainer(container);
}
```

When overriding setContainer() in this way, it is best to make wrapup() idem potent because future implementations of the director might automatically unlock resources of removed actors.

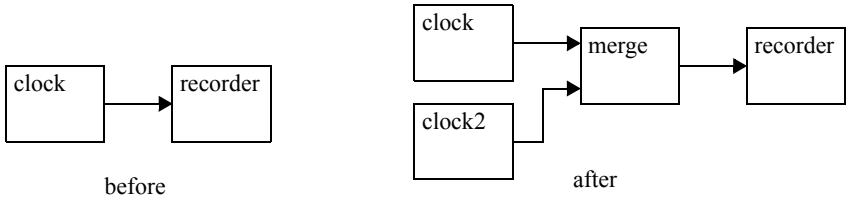

FIGURE 1.6. Topology before and after mutation for the example in figure 1.7.

```
package ptolemy.domains.de.lib.test;

import ptolemy.kernel.util.*;
import ptolemy.kernel.*;
import ptolemy.actor.*;
import ptolemy.actor.lib.*;
import ptolemy.domains.de.kernel.*;
import ptolemy.domains.de.lib.*;

public class Mutate {

    public Manager manager;

    private Recorder _rec;
    private Clock _clock;
    private TypedCompositeActor _top;
    private DEDirector _director;

    public Mutate() throws IllegalActionException,
            NameDuplicationException {
       _top = new TypedCompositeActor();
       _top.setName("top");
       manager = new Manager();
       _director = new DEDirector();
       _top.setDirector(_director);
       _top.setManager(manager);

       _clock = new Clock(_top, "clock");
       _clock.values.setExpression("[1.0]");
       _clock.offsets.setExpression("[0.0]");
       _clock.period.setExpression("1.0");
       _rec = new Recorder(_top, "recorder");
       _top.connect(_clock.output, _rec.input);
    }

    public void insertClock() {
       // Create an anonymous inner class
       ChangeRequest change = new ChangeRequest(_top, "test2") {
          public void _execute() throws IllegalActionException,
                NameDuplicationException {
             _clock.output.unlinkAll();
             _rec.input.unlinkAll();
             Clock clock2 = new Clock(_top, "clock2");
             clock2.values.setExpression("[2.0]");
             clock2.offsets.setExpression("[0.5]");
             clock2.period.setExpression("2.0");
             Merge merge = new Merge(_top, "merge");
             _top.connect(_clock.output, merge.input);
             _top.connect(clock2.output, merge.input);
             _top.connect(merge.output, _rec.input);
             // Any pre-existing input port whose connections
             // are modified needs to have this method called.
             _rec.input.createReceivers();
             _director.invalidateSchedule();
          }
       };
       _top.requestChange(change);
    }
}
```

FIGURE 1.7. An example of a class that constructs a model and then mutates it.

1.5 Writing DE Actors

It is very common in DE modeling to include custom-built actors. No pre-defined actor library seems to prove sufficient for all applications. For the most part, writing actors for the DE domain is no different than writing actors for any other domain. Some actors, however, need to exercise particular control over time stamps and actor priorities. Such actors use instances of DEIOPort rather than TypedIOPort. The first section below gives general guidelines for writing DE actors and domain-polymorphic actors that work in DE. The second section explains in detail the priorities, and in particular, how to write actors that implement delays. The final section discusses actors that operate as a Java thread.

1.5.1 General Guidelines

The points to keep in mind are:

- When an actor fires, not all ports have tokens, and some ports may have more than one token. The time stamps of the events that contained these tokens are no longer explicitly available. The current model time (obtained by the getCurrentTime() method of the director) is assumed to be the time stamp of the events.
- If the actor leaves unconsumed tokens on its input ports, then it will be iterated again before model time is advanced. This ensures that the current model time is in fact the time stamp of the input events. However, occasionally, an actor will want to leave unconsumed tokens on its input ports, and not be fired again until there is some other new event to be processed. To get this behavior, it should return *false* from prefire(). This indicates to the DE director that it does not wish to be iterated.
- If the actor returns *false* from postfire(), then the director will not fire that actor again. Events that are destined for that actor are discarded.
- When an actor produces an output token, the time stamp for the output event is taken to be the current model time. If the actor wishes to produce an event at a future model time, one way to accomplish this is to call the director's fireAt() method to schedule a future firing, and then to produce the token at that time. A second way to accomplish this is to use instances of DEIOPort and use the overloaded send() or broadcast() methods that take a time delay argument.
- If an actor contains a callback method or a private thread and this callback or thread wishes to produce an event now or at a future model time, then a reliable way to achieve this is to call either the fireAtCurrentTime() method or the fireAtRelativeTime() method. These methods may safely be called asynchronously, yielding real-time liveness. By contrast, fireAt() must be called from within the director thread that calls standard actor methods such as prefire(), fire(), and postfire().
- The DEIOPort class (see figure 1.4) can produce events in the future, but there is an important subtlety with using these methods. Once an event has been produced, it cannot be retracted. In particular, even if the actor which produced the event (or the destination actor of the event) is deleted before model time reaches that of the future event, the event will be delivered to the destination. If you use fireAt(), fireAtCurrentTime(), or fireAtRelativeTime() instead to generate delayed events, then if the actor is deleted (or returns *false* from postfire()) before the future event, then the future event will not be produced.
- By convention in Ptolemy II, actors update their state only in the postfire() method. In DE, the fire() method is only invoked once per iteration, so there is no particular reason to stick to this convention. Nonetheless, we recommend that you do in case your actor becomes useful in other domains. The simplest way to ensure this is follow the following pattern. For each state variable,

such as a private variable named *_count*,

```
private int _count;
```

create a shadow variable

```
private int _countShadow;
```

Then write the methods as follows:

```
public void fire() {
   _countShadow = _count;
   ... perform some computation that may modify _countShadow ...
}
public boolean postfire() {
   _count = _countShadow;
   return super.postfire();
}
```

This ensures that the state is updated only in postfire().

In a similar fashion, delayed outputs (produced by either mechanism) should be produced in the postfire() method, since a delayed outputs are persistent state. Thus, fireAt() should be called in postfire() only, as should the overloaded send() and broadcast() of DEIOPort.

1.5.2 Examples

Simplified Delay Actor. An example of a domain-specific actor for DE is shown in figure 1.8. This actor delays input events by some amount specified by a parameter. The domain-specific features of the actor are shown in bold. They are:

- It uses DEIOPort rather than TypedIOPort.
- It overrides the pruneDependencies() method to issue the following statement:

  ```
  removeDependency(input, output);
  ```

  This statement declares to the director that this actor implements a delay from input to output. The director uses this to break the precedences when constructing the DAG to find priorities.
- It uses an overloaded send() method, which takes a delay argument, to produce the output. Notice that the output is produced in the postfire() method, since by convention in Ptolemy II, persistent state is not updated in the fire() method, but rather is updated in the postfire() method.

Server Actor. The Server actor in the DE library (see figure 1.5) uses a rich set of behavioral properties of the DE domain. A server is a process that takes some amount of time to serve "customers." While it is serving a customer, other arriving customers have to wait. This actor can have a fixed service time (set via the parameter *serviceTime*, or a variable service time, provided via the input port *newServiceTime*). A typical use would be to supply random numbers to the *newServiceTime* port to generate ran-

dom service times. These times can be provided at the same time as arriving customers to get an effect where each customer experiences a different, randomly selected service time.

The (compacted) code is shown in figure 1.9. This actor extends DETransformer, which has two public members, *input* and *output*, both instances of DEIOPort. It also overrides pruneDependencies() to remove dependencies (and hence declare delays).

The actor keeps track of the time at which it will next be free in the private variable _nextTimeFree. This is initialized to minus infinity to indicate that whenever the model begins executing, the server is free. The prefire() method determines whether the server is free by comparing this private variable against the current model time. If it is free, then this method returns true, indicating to the scheduler that it can proceed with firing the actor. If the server is not free, then the prefire() method checks to see whether there is a pending input, and if there is, requests a firing when the actor will become free. It then returns false, indicating to the scheduler that it does not wish to be fired at this time.

The fire() method is invoked only if the server is free. It first checks to see whether the newServiceTime port is connected to anything, and if it is, whether it has a token. If it does, the token is read and used to update the *serviceTime* parameter[1]. No more than one token is read, even if there are more in the input port, in case one token is being provided per pending customer.

```
package ptolemy.domains.de.lib.test;

public class SimpleDelay extends TypedAtomicActor {

    public SimpleDelay(TypedCompositeActor container, String name)
            throws NameDuplicationException, IllegalActionException {
        super(container, name);
        input = new DEIOPort(this, "input", true, false);
        output = new DEIOPort(this, "output", false, true);
        delay = new Parameter(this, "delay", new DoubleToken(1.0));
        delay.setTypeEquals(BaseType.DOUBLE);
    }

    public Parameter delay;
    public DEIOPort input;
    public DEIOPort output;
    private Token _currentInput;

    public void fire() throws IllegalActionException {
        _currentInput = input.get(0);
    }

    public boolean postfire() throws IllegalActionException {
        output.send(0, _currentInput,
                ((DoubleToken)delay.getToken()).doubleValue());
        return super.postfire();
    }

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

FIGURE 1.8. A domain-specific actor in DE.

1. This actor could now use a PortParameter, but that class did not exist when it was written.

```
public class Server extends DETransformer {
    public DEIOPort newServiceTime;
    public Parameter serviceTime;
    private Token _currentInput;
    private double _nextTimeFree = Double.NEGATIVE_INFINITY;
    public Server(TypedCompositeActor container, String name)
            throws NameDuplicationException, IllegalActionException  {
        super(container, name);
        serviceTime = new Parameter(this, "serviceTime", new DoubleToken(1.0));
        serviceTime.setTypeEquals(BaseType.DOUBLE);
        newServiceTime = new DEIOPort(this, "newServiceTime", true, false);
        newServiceTime.setTypeEquals(BaseType.Double);
        output.setTypeAtLeast(input);
    }
    ... attributeChanged() method ...
    public void initialize() throws IllegalActionException {
        super.initialize();
        _nextTimeFree = Double.NEGATIVE_INFINITY;
    }
    public void fire() throws IllegalActionException {
        if (newServiceTime.getWidth() > 0 && newServiceTime.hasToken(0)) {
            DoubleToken time = (DoubleToken)(newServiceTime.get(0));
            serviceTime.setToken(time);
        }
        if (input.getWidth() > 0 && input.hasToken(0)) {
            _currentInput = input.get(0);
            double delay =
                ((DoubleToken)serviceTime.getToken()).doubleValue();
            _nextTimeFree = ((DEDirector)getDirector()).getCurrentTime()
                + delay;
        } else {
            _currentInput = null;
        }
    }
    public void initialize() throws IllegalActionException {
        super.initialize();
        _nextTimeFree = Double.NEGATIVE_INFINITY;
    }
    public boolean prefire() throws IllegalActionException {
        DEDirector director = (DEDirector)getDirector();
        if (director.getCurrentTime() >= _nextTimeFree) {
            return true;
        } else {
            // Schedule a firing if there is a pending
            // token so it can be served.
            if (input.hasToken(0)) {
                director.fireAt(this, _nextTimeFree);
            }
            return false;
        }
    }
    public boolean postfire() throws IllegalActionException {
        if (_currentInput != null) {
            double delay =
                ((DoubleToken)serviceTime.getToken()).doubleValue();
            output.send(0, _currentInput, delay);
        }
        return super.postfire();
    }
    public void pruneDependencies() {
        super.pruneDependencies();
        removeDependency(input, output);
        removeDependency(newServiceTime, output);
    }
}
```

FIGURE 1.9. Code for the Server actor. For more details, see the source code.

The fire() method then continues by reading an input token, if there is one, and updating _nextTimeFree. The input token that is read is stored temporarily in the private variable _currentInput. The postfire() method then produces this token on the output port, with an appropriate delay. This is done in the postfire() method rather than the fire() method in keeping with the policy in Ptolemy II that persistent state is not updated in the fire() method. Since the output is produced with a future time stamp, then it is persistent state.

Note that when the actor will not get input tokens that are available in the fire() method, it is essential that prefire() return false. Otherwise, the DE scheduler will keep firing the actor until the inputs are all consumed, which will never happen if the actor is not consuming inputs!

Like the SimpleDelay actor in figure 1.8, this one produces outputs with future time stamps, using the overloaded send() method of DEIOPort that takes a delay argument. There is a subtlety associated with this design. If the model mutates during execution, and the Server actor is deleted, it cannot retract events that it has already sent to the output. Those events will be seen by the destination actor, even if by that time neither the server nor the destination are in the topology! This could lead to some unexpected results, but hopefully, if the destination actor is no longer connected to anything, then it will not do much with the token.

1.5.3 Thread Actors[1]

In some cases, it is useful to describe an actor as a thread that waits for input tokens on its input ports. The thread suspends while waiting for input tokens and is resumed when some or all of its input ports have input tokens. While this description is functionally equivalent to the standard description explained above, it leverages on the Java multi-threading infrastructure to save the state information.

Consider the code for the ABRecognizer actor shown in figure 1.10. The two code listings implement two actors with equivalent behavior. The left one implements it as a threaded actor, while the

```
public class ABRecognizer extends DEThreadActor {
    StringToken msg = new StringToken("Seen AB");

    // the run method is invoked when the thread
    // is started.
    public void run() {
        while (true) {
            waitForNewInputs();
            if (inportA.hasToken(0)) {
                IOPort[] nextInport = {inportB};
                waitForNewInputs(nextInport);
                outport.broadcast(msg);
            }
        }
    }
```

```
public class ABRecognizer extends DEActor {
    StringToken msg = new StringToken("Seen AB");

    // We need an explicit state variable in
    // this case.
    int state = 0;

    public void fire() {
        switch (state) {
            case 0:
                if (inportA.hasToken(0)) {
                    state = 1;
                    break;
                }
            case 1:
                if (inportB.hasToken(0)) {
                    state = 0;
                    outport.broadcast(msg);
                }
        }
    }
}
```

FIGURE 1.10. Code listings for two style of writing the ABRecognizer actor.

1. This section describes techniques that have not been widely used, and are not extensively tested.

right one implements it as a standard actor. We will from now on refer to the left one as the threaded description and the right one as the standard description. In both descriptions, the actor has two input ports, `inportA` and `inportB`, and one output port, `outport`. The behavior is as follows.

> *Produce an output event at outport as soon as events at inportA and inportB occurs in that particular order, and repeat this behavior.*

Note that the standard description needs a state variable `state`, unlike the case in the threaded description. In general the threaded description encodes the state information in the position of the code, while the standard description encodes it explicitly using state variables. While it is true that the context switching overhead associated with multi-threading application reduces the performance, we argue that the simplicity and clarity of writing actors in the threaded fashion is well worth the cost in some applications.

To write an actor in the threaded fashion, one simply derives from the DEThreadActor class and implements the run() method. In many cases, the content of the run() method is enclosed in the infinite '`while(true)`' loop since many useful threaded actors do not terminate.

The waitForNewInputs() method is overloaded and has two flavors, one that takes no arguments and another that takes an IOPort array as argument. The first suspends the thread until there is at least one input token in at least one of the input ports, while the second suspends until there is at least one input token in any one of the specified input ports, ignoring all other tokens.

In the current implementation, both versions of waitForNewInputs() clear all input ports before the thread suspends. This guarantees that when the thread resumes, all tokens available are new, in the sense that they were not available before the waitForNewInput() method call.

The implementation also guarantees that between calls to the waitForNewInputs() method, the rest of the DE model is suspended. This is equivalent to saying that the section of code between calls to the waitForNewInput() method is a critical section. One immediate implication is that the result of the method calls that check the configuration of the model (e.g. hasToken() to check the receiver) will not be invalidated during execution in the critical section. It also means that this should not be viewed as a way to get parallel execution in DE. For that, consider the DDE domain.

It is important to note that the implementation serializes the execution of threads, meaning that at any given time there is only one thread running. When a threaded actor is running (i.e. executing inside its run() method), all other threaded actors and the director are suspended. It will keep running until a waitForNewInputs() statement is reached, where the flow of execution will be transferred back to the director. Note that the director thread executes all non-threaded actors. This serialization is needed because the DE domain has a notion of global time, which makes parallelism much more difficult to achieve.

The serialization is accomplished by the use of monitor in the DEThreadActor class. Basically, the fire() method of the DEThreadActor class suspends the calling thread (i.e. the director thread) until the threaded actor suspends itself (by calling waitForNewInputs()). One key point of this implementation is that the threaded actors appear just like an ordinary DE actor to the DE director. The DEThreadActor base class encapsulates the threaded execution and provides the regular interfaces to the DE director. Therefore the threaded description can be used whenever an ordinary actor can, which is everywhere.

The code shown in figure 1.11 implements the run method of a slightly more elaborate actor with the following behavior:

Emit an output O as soon as two inputs A and B have occurred. Reset this behavior each time the input R occurs.

Recent work has extended the DE Director to support parallel execution in the form of actors containing private threads and callbacks. Future work in this area may involve extending the infrastructure to support additional concurrency constructs, such as preemption, other forms of parallel execution, etc. It might also be interesting to explore new concurrency semantics similar to the threaded DE, but without the 'forced' serialization.

1.6 Composing DE with Other Domains

One of the major concepts in Ptolemy II is modeling heterogeneous systems through the use of

```
public void run() {
    try {
        while (true) {
            // In initial state..
            waitForNewInputs();
            if (R.hasToken(0)) {
                // Resetting..
                continue;
            }
            if (A.hasToken(0)) {
                // Seen A..
                IOPort[] ports = {B,R};
                waitForNewInputs(ports);
                if (!R.hasToken(0)) {
                    // Seen A then B..
                    O.broadcast(new DoubleToken(1.0));
                    IOPort[] ports2 = {R};
                    waitForNewInputs(ports2);
                } else {
                    // Resetting
                    continue;
                }
            } else if (B.hasToken(0)) {
                // Seen B..
                IOPort[] ports = {A,R};
                waitForNewInputs(ports);
                if (!R.hasToken(0)) {
                // Seen B then A..
                O.broadcast(new DoubleToken(1.0));
                IOPort[] ports2 = {R};
                waitForNewInputs(ports2);
            } else {
                // Resetting
                continue;
            }
        } // while (true)
    } catch (IllegalActionException e) {
        getManager().notifyListenersOfException(e);
    }
}
```

FIGURE 1.11. The run() method of the ABRO actor.

hierarchical heterogeneity. Actors on the same level of hierarchy obey the same set of semantics rules. Inside some of these actors may be another domain with a different model of computation. This mechanism is supported through the use of opaque composite actors. An example is shown in figure 1.12. The outermost domain is DE and it contains seven actors, two of them are opaque and composite. The opaque composite actors contain subsystems, which in this case are in the DE and CT domains.

1.6.1 DE inside Another Domain

The DE subsystem completes one iteration whenever the opaque composite actor is fired by the outer domain. One of the complications in mixing domains is in the synchronization of time. Denote the current time of the DE subsystem by t_{inner} and the current time of the outer domain by t_{outer}. An iteration of the DE subsystem is similar to an iteration of a top-level DE model, except that prior to the iteration tokens are transferred from the ports of the opaque composite actors into the ports of the contained DE subsystem, and after the end of the iteration, the director requests a refire at the smallest time stamp in the event queue of the DE subsystem. This presumes that the DE subsystem knows at what time stamp the it, or one of its contained actors, will wish to be refired. Future work may remove this limitation, allowing real-time events (such as from I/O) to propagate out of a DE subsystem. Currently the DE domain can handle such asynchronous events only if it is not inside another domain.

The transfer of tokens from the ports of the opaque composite actor into the ports of the contained DE subsystem actors is done in the transferInputs() method of the DE director. This method is extended from its default implementation in the Director class. The implementation in the DEDirector class advances the current time of the DE subsystem to the current time of the outer domain, then calls

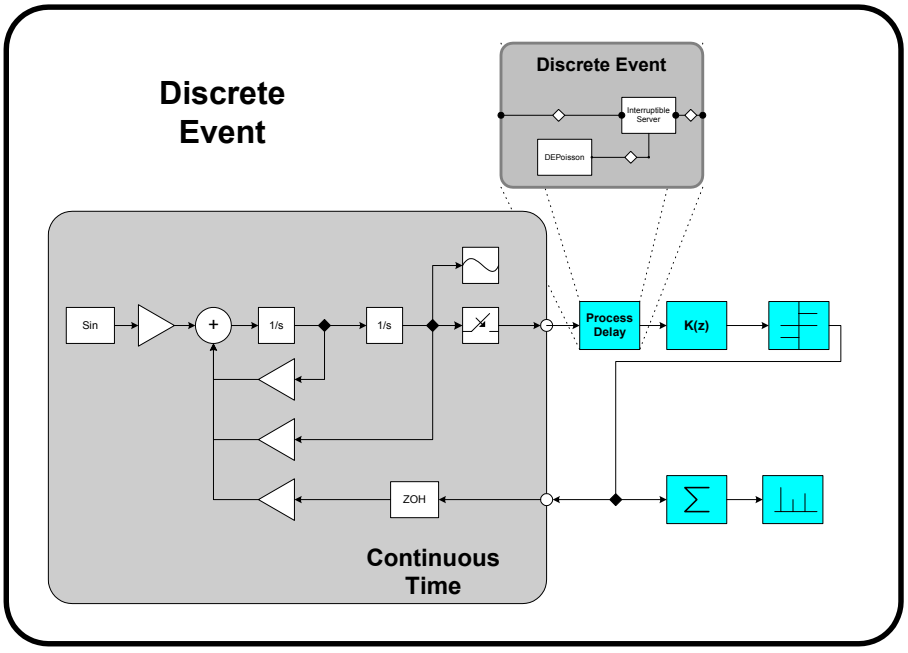

FIGURE 1.12. An example of heterogeneous and hierarchical composition. The CT subsystem and DE subsystem are inside an outermost DE system. This example is developed by Jie Liu [93].

super.transferInputs(). It is done in order to correctly associate tokens seen at the input ports of the opaque composite actor, if any, with events at the current time of the outer domain, t_{outer}, and put these events into the global event queue. This mechanism is, in fact, how the DE subsystem synchronize its current time, t_{inner}, with the current time of the outer domain, t_{outer}.(Recall that the DE director advances time by looking at the smallest time stamp in the event queue of the DE subsystem). Specifically, before the advancement of the current time of the DE subsystem t_{inner} is less than or equal to the t_{outer}, and after the advancement t_{inner} is equal to the t_{outer}.

Requesting a refiring is done in the postfire() method of the (inner) DE director by calling the fireAt() method of the executive (outer) director. Its purpose is to ensure that events in the DE subsystem are processed on time with respect to the current time of the outer domain, t_{outer}.

Note that if the DE subsystem is fired due to the outer domain processing a refire request, then there may not be any tokens in the input port of the opaque composite actor at the beginning of the DE subsystem iteration. In that case, no new events with time stamps equal to t_{outer} will be put into the global event queue. Interestingly, in this case, the time synchronization will still work because t_{inner} will be advanced to the smallest time stamp in the global event queue which, in turn, has to equal t_{outer} because we always request a refire according to that time stamp.

1.6.2 Another Domain inside DE

Due to its nature, any opaque composite actor inside DE is opaque and therefore, as far as the DE Director is concerned, behaves exactly like a domain polymorphic actor. Recall that domain polymorphic actors are treated as functions with zero delay in computation time. To produce events in the future, domain polymorphic actors request a refire from the DE director and then produce the events when it is refired.