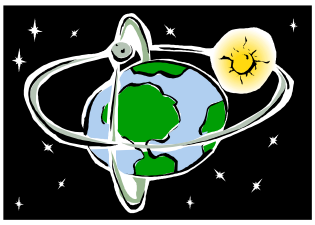


DISCRETE EVENT MODELING IN PTOLEMY II

Lukito Muliadi

Department of Electrical Engineering and Computer Science
University of California, Berkeley

lmuliadi@eecs.berkeley.edu

5/18/99

Abstract

This report describes the discrete-event semantics and its implementation in the Ptolemy II software architecture. The discrete-event system representation is appropriate for time-oriented systems such as queueing systems, communication networks, and hardware systems. A key strength in our discrete-event implementation is that simultaneous events are handled systematically and deterministically. A formal and rigorous treatment of this property is given. One of the Ptolemy II major features is in the heterogenous modeling of systems. Composition of the DE domain with different domains in Ptolemy II is discussed. The performance of a discrete-event simulator depends heavily on the algorithm by which events are mantained and sorted in the event queue. In the Ptolemy II DE domain, the event queue is implemented as the calendar queue data structure. The calendar queue is an extremely fast implementation of the priority queue with time complexity equal to O(1) for both enqueue and dequeue operations. A comprehensive description of this data structure is given. Based on the tagged signal model, the formal semantics of our discrete event implementation are given in terms of the firing functions of actors.

Acknowledgments

I would like to express my sincere gratitude to my research advisor, Professor Edward A. Lee, for introducing me to this exciting research field and helping me throughout the course of my master study in U.C. Berkeley. Without his endless support and excellent guidance, I would not have finished this project.

I also like to thank the Ptolemy group for all the wonderful discussions which help making my project a success. I am also grateful to Bilung Lee and Jie Liu, not only for being such great cubicle-mates, but also for their excellent comments on the draft of this report and for their help with the technical details of this research.

This research is part of the Ptolemy project, which is supported by the Defense Advanced Research Projects Agency (DARPA), the State of California MICRO program, and the following companies: The Cadence Design Systems, Hewlett Packard, Hitachi, Hughes Space and Communications, Motorola, NEC, and Philips.

Finally, I like to thank both of my parents for their continuous love and encouragement throughout my educational career. This report is dedicated to them.

Discrete-Event Modeling in Ptolemy II

Lukito Muliadi

1.Introduction

The Ptolemy II [1] Discrete Event (DE) domain provides a general environment for time-oriented simulations of systems such as queueing systems, communication networks, and hardware systems. In this domain, actors communicate by sending tokens across connections. The token sent and the time at which it was sent form an event in the DE domain. Upon receiving an event, the destination actor is activated and a reaction takes place. The reaction may change the internal state of the actor and possibly generate new events, resulting in further reactions. A DE domain scheduler ensures that events are processed chronologically.

For example, we can model the arrivals of buses and passengers at a given bus stop as discrete events. By constructing a model with appropriate actors, we can calculate various statistical properties, e.g. the distribution of the waiting time of the passengers. An applet implementing this example can be seen in http://ptolemy.eecs.berkeley.edu/ptolemyII/ptII0.2/ptII0.2devel/ptolemy/domains/de/demo/inspection/index.html.

2.Discrete Event System Representation

First we give an overview of the infrastructure and terminology of Ptolemy II, the environment for our implementation. Then we will show how to build a model in the Ptolemy II DE domain. A key strength in our implementation is that simultaneous events are handled systematically and deterministically.

2.1. Ptolemy II infrastructure

Ptolemy II [3] is a complete, from the ground up, redesign of the Ptolemy 0.x software environment [4], which supports heterogeneous modeling and design of concurrent systems. Modeling is the act of representing a system or subsystem formally.Design is the act of defining a system or subsystem. A principle of Ptolemy II is that the choice of models of computation strongly affect the quality of a system design [2]. A model of computation is the set of “laws of physics” that govern the interaction of components in the model. The Ptolemy II software provides a rich variety of models of computation that deal with concurrency and time in different ways. Each model of computation is called a domain in Ptolemy II. All of these can give meaning to the bubble-and-arc syntax shown in figure 1. One possible semantics for the syntax in figure 1 is that of discrete events. This semantics is implemented as

the DE domain in Ptolemy II. In discrete-event models of computation, an arc represents sets of events placed in time, and a bubble represents a function that takes an event and produces another event.

The Ptolemy II software is divided into packages. The core packages provide functionality for constructing and traversing an abstract clustered graph, encapsulating data, variables, and parameters, sequencing execution, and performing graph and mathematical algorithms. A complete description of these packages can be obtained in the Ptolemy II design document [3].

2.2. The Execution Sequence of A DE Model

A typical DE model consists of a network of actors and a DE director governing the execution of the model. An example is shown in figure 2. Actors contain ports, and ports are connected to each other via relations.

Recall that an event in the DE domain is an aggregation of a token and a time stamp. 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 the global event queue. The time stamp associated with this token is determined by the sending actor. In the global event queue, these events are sorted based on their time stamps and their destination input ports. An event is removed from the global event queue and put into its destination input port, when the *model time* (explained below) reaches its time stamp.

In the DE model of computation, time is *global*, in the sense that all actors share the same global time. We denote this shared global time by the *current time*. Some also call this the *simulation time* or

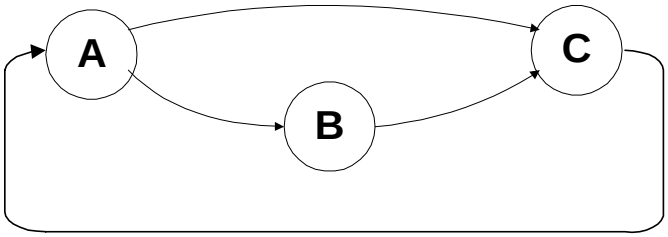


Figure 1. A single buble-and-arc syntax that has several possible semantics (interpretation).

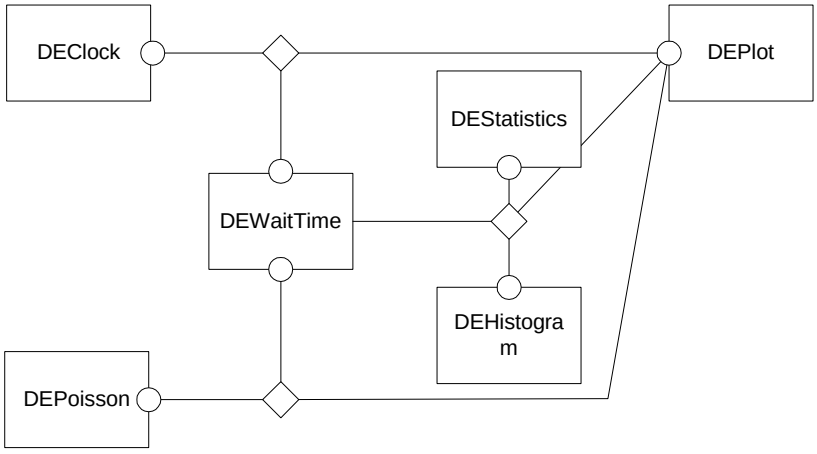


Figure 2. A sample of topology in the DE domain

model time. At all times, all events stored in the global event queue have time stamps greater than or equal to the current time. The DE director is responsible for advancing (i.e. incrementing) the current time when all events with time stamps equal to the current 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. This advancement is done at the beginning of each iteration.

At each iteration, after advancing the current time, the director chooses some events in the global event queue based on their time stamps and destination ports. The DE director chooses events according to these rules:

- Find a non-empty set of events, E, associated with the smallest time stamp.
- If the set of events contains more than one event, find the one with highest priority, e_{max}. Simultaneous events (i.e. events whose time stamps are equal) are prioritized carefully by means of topological sort on input ports. This is done to ensure deterministic behaviour.
- Finally, choose all events in set E that have the same destination actor with the event e_{max}. Note that the destination actor of an event is the actor containing the destination input port of that event.

The chosen events are then removed from the global event queue and inserted into the appropriate input ports of the destination actor. Then, the director fires the destination actor, i.e. invokes the fire() method of the destination actor. We say that the actor is fired. Notice that the firing may produce simultaneous events at the current time. I.e. the actor reacts instantenously.

Then, the DE director checks whether there are any events in the global event queue with time stamps equal to the current time. If there are not, the iteration finishes. Otherwise, the DE director again chooses some events in the global event queue according to the above rules and fires the destination actor. So, intuitively speaking, a iteration iteratively processes events in the global event queue with time stamp equal to the current time and ends when there is no more such events. In summary, the flowchart of an iteration in the DE domain is shown in figure 3.

Before one of the iterations described above can be run, there have to be initial events in the global event queue. The actors in the DE model may produce initial events in their `initialize()` method. In Ptolemy II implementation, the initial events are in the form of *refire requests*. A refire request is a request generated by an actor to the DE director that it be fired again in the future. The refire request is implemented by the `fireAt(actor, time)` method, where the `actor` and `time` arguments indicate the actor to be refired and the time at which the refiring will take place, respectively. In a typical situation, only the source actors of a DE model produce these intial events by calling the `fireAt()` method in their `initialize()` method. Furthermore, these source actors needs to generate refire requests (by calling the fireAt() method) in their `fire()` method. Otherwise, there will not be events in the global event queue destined to these source actors and, consequently, they will never be fired again. Note that source actors do not receive input events from any other actors; they simply do not have input ports. The refire request is an example of a `pure event`. A pure event is an event whose token value is unimportant (e.g. null). Indeed, requesting a refire can be thought as setting an alarm clock to be awakened in the future, so only the time stamp matters. This simple scheme gives us initial events in the global event queue. In addition, we can define the *start time* to be the smallest time stamp of these intial events. The start time is set once at the beginning of model execution and it is fixed throughout the whole execution. Since the first step of an iteration (figure 3) is setting the current time to be the smallest time stamp of the events in the global event queue, it can be shown easily that the current time is equal to the start time at the first iteration.

The execution of the DE model consists of a sequence of these iterations. The execution stops when either one of these conditions become true:

- The current time reaches a certain value denoted by the stop time, which is a parameter of the DE director.
- The global event queue becomes empty.

When this happens, the execution ends by calling the wrapup methods of all actors.

In summary, the execution of a DE model consists of three phases, namely:

- *initialization* phase: This phase initializes the execution of the DE model by calling the initialize() method of all actors.
- *iterations* phase: This phase consists of one or more iterations. An iteration of a DE model is the iteration described above.
- *wrapup* phase: This phase ends the execution of the DE model by calling the wrapup() method of

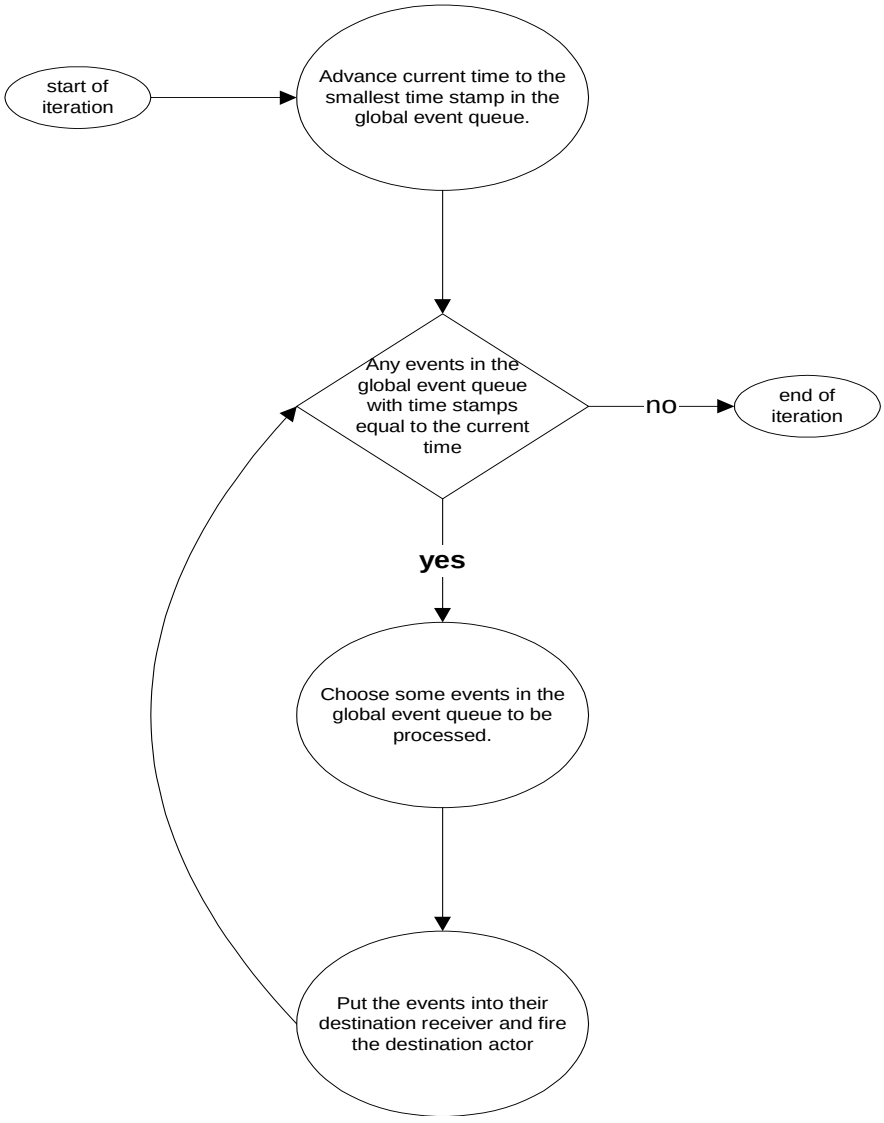


Figure 3. A flowchart depicting one iteration in the DE system

all actors.

2.3. Simultaneous Events

An important aspect of a DE model is the handling of simultaneous events. The way we choose to handle this is by assigning ranks to input ports. The ranks are drawn from the set of non-negative integers. They are uniquely assigned, i.e. no two distinct input ports are assigned the same rank. Simultaneous events are handled by processing the events destined to ports with the lowest ranks. The ranks are determined by a *topological sort* of the input ports.

2.3.1. Problem Definition

We define the following sets:

- *InputPorts*: The set of all input ports in a model.
- *OutputPorts*: The set of all output ports in a model.
- *Actors*: The set of all actors in a model.

Also, define the following functions:

- *inputs*: $Actors \rightarrow \wp(InputPorts)$, such that *inputs*(*a*) is the set of input ports owned by actor *a*. (Note: $\wp$(A) is the set of all subsets of A, i.e. the powerset of A)
- *outputs*: $Actors \rightarrow \wp(OutputPorts)$, such that *outputs*(*a*) is the set of output ports owned by actor *a*.
- *connect*: $OuputPorts \rightarrow \wp(InputPorts)$, such that *connect*(*outport*) is the set of input ports connected to the output port *outport*.

Each actor can also define relations between its ports. There are two sorts of relations:

- `inport.triggers(outport)`, where `inport` and `outport` are an input port and an output port of the same actor, respectively. This relation should be asserted if an event sent into `inport` has the possibility of inducing a simultaneous event on the `outport`.
- `inport_1.before(inport_2)`, where `inport_1` and `inport_2` are input ports of the same actor. This relation means that if there are simultaneous events sent into `inport_1` and `inport_2`, then the scheduler will guarantee that the one(s) for `inport_1` will be processed first.

Consider, for example, a WaitingTime actor as shown in figure 4. This actor measures the time that events at the input port *waiter* have to wait for events at the input port *waitee*. Specifically, there will be one output event for each *waiter* input event and the event is generated at the next closest arrival of an event at *waitee*. Therefore, an event at the input port waitee might generate an output event if there has been at least one 'pending' *waiter* event. Thus, the WaitingTime actor needs to assert

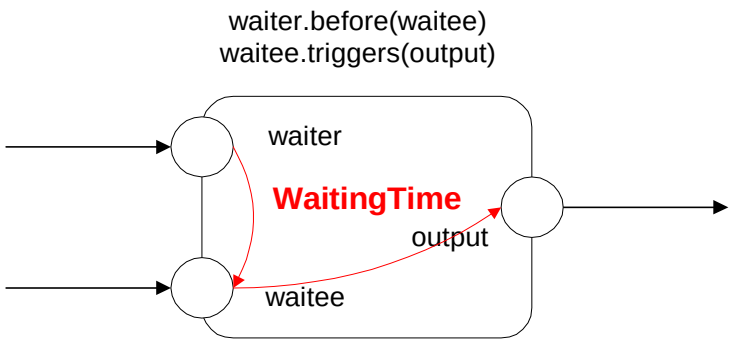


Figure 4. A WaitingTime which contains a before and a trigger relationship between its ports.

`waitee.triggers(output)`. Now, suppose that there are simultaneous events at the input ports waitee and waiter. A sensible approach is to have the *waiter* event 'served' by the *waitee* event and to produce the output event right at that moment. I.e. the *waiter* event does not wait for the next *waitee* event. To ensure that the *waiter* event can be 'served' by the *waitee* event, the simultaneous *waiter* event has to be visible when the *waitee* event is, i.e. events at the input port *waiter* have higher priority than ones at the input port *waitee*. Thus, the WaitingTime actor needs to assert `waiter.before(waitee)`.

For the `before` and `trigger` relations, we define the following two functions:

- *triggers*: *InputPorts* $\rightarrow \wp$(*OutputPorts*), such that *triggers*(*inport*) is the set of output ports generating simultaneous events due to an event in the input port *inport*. The *inport* and *triggers*(*inport*) must belong to the same actor, i.e.

$$\forall inport \in InputPorts, (\exists actor \in Actors, inport \in inputs(actor) \wedge triggers(inport) \subset outputs(actor)) \quad (1)$$

- *before*: *InputPorts* $\rightarrow \wp$(*InputPorts*), such that *before*(*inport*) is the set of input ports that have lower precedence than the input port *inport*. Again, the *inport* and *before*(*inport*) must belong to the same actor, i.e.

$$\forall inport \in InputPorts, (\exists actor \in Actors, inport \in inputs(actor) \wedge before(inport) \subset inputs(actor)) \quad (2)$$

2.3.2. Problem Statement

The process of assigning ranks by topological sort is then equivalent to finding the function *rank* : *InputPorts* $\rightarrow$N $\cup\{0\}$, satisfying the following constraints:

- For each input port *inport*, all input ports connected to the output ports triggered by *inport* must have rank greater than the rank of *inport*, i.e.

$$\forall inport \in InputPorts, \forall outport \in triggers(inport), \forall inport2 \in connect(outport), \quad (3)$$
$$rank(inport) < rank(inport2)$$

- For each input port *inport*, the input ports contained in *before*(*inport*) must have ranks greater than the rank of *inport*, e.g.

$$\forall inport \in InputPorts, \forall inport2 \in before(inport), rank(inport) < rank(inport2) \quad (4)$$

In general, the function *rank* satisfying these constraints will not be unique. Therefore, the result of the topological sort is non-deterministic, but it can be shown that the behaviour of the overall system will not be affected by the choice of *rank*.

2.3.3. Example

To make the idea more concrete, consider the topology shown in figure 5. Suppose the actors assert the following relations:

- X: `A.before(B)` and `B.triggers(C)`
- Y: `D.triggers(E)`
- Z: `F.before(G)` and `F.triggers(H)`

Putting this information into our framework, we have the following (Note that functions are described by ordered pairs of argument and value):

- InputPorts = {A, B, D, F, G}
- OutputPorts = {C, E, H}
- Actors = {X, Y, Z}

- inputs = {(X, {A, B}), (Y, {D}), (Z, {F, G})}
- outputs = {(X, {C}), (Y, {E}), (Z, {H})}
- connects = {(C, {D, F}), (E, {G}), (H, {A, B})}
- triggers = {(A, {}), (B, {C}), (D, {E}), (F, {H}), (G, {})}
- before = {(A, {B}), (B, {}), (D, {}), (F, {G}), (G, {})}

From (3) we have the following constraints:

- rank(B) < rank(D)
- rank(B) < rank(F)
- rank(D) < rank(G)
- rank(F) < rank(A)
- rank(F) < rank(B)

and then from (4) we have the followings:

- rank(A) < rank(B)
- rank(F) < rank(G)

Combining some inequalities, we have:

$$\text{rank}(A) < \text{rank}(B) < \text{rank}(F) < \text{rank}(A) \quad (5)$$

which cannot be satisfied for any function rank(). Indeed, figure 5 depicts a zero-delay directed loop where topological sort cannot be done. The director will refuse to run the model, and will terminate with an error message.

However, the error described above is conservative, i.e. there are situations where a model containing zero-delay directed loop can be useful in design. This is due to the fact that the triggers relation has to be asserted whenever a simultaneous reaction is possible. The precise condition for the executability of a directed loop, L, is that there exists a finite power N and $\delta > 0$ such that L^N contains at least a delay of δ time units [7]. To run models containing zero-delay directed loop, the topology has to be modified by inserting a DEDelay actor (with delay parameter equal to zero) in the zero-delay directed loop. A DEDelay actor with delay parameter equal to zero passes all tokens unmodified (i.e. delays them by zero time) and it does not assert the triggers relation between its input and output port. (Strictly speaking, any actor that does not assert the triggers relation between its input and output port will be able to break the zero delay loop. In addition, we want the actor to pass tokens unmodified in order to preserve the semantics of the model. The DEDelay actor is provided for convenience.) The

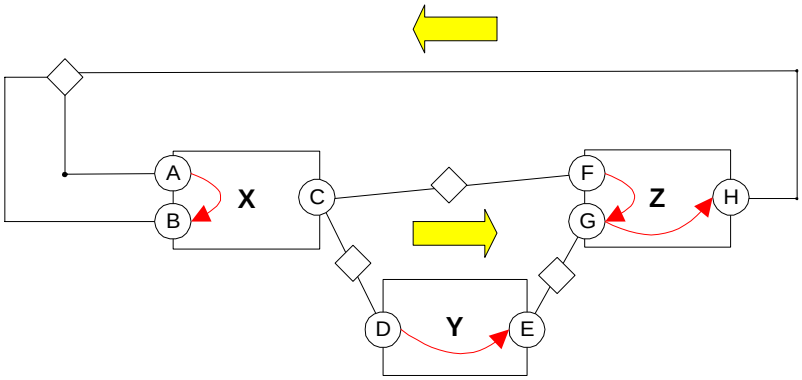


Figure 5. An example of directed zero-delay loop

resulting topology is shown in figure 6.

Consequently, we have the followings:

- InputPorts = {A, B, D, F, G, J}
- OutputPorts = {C, E, H, I}
- Actors = {X, Y, Z, DEDelay}
- inputs = {(X, {A, B}), (Y, {D}), (Z, {F, G}), (DEDelay, {J})}
- outputs = {(X, {C}), (Y, {E}), (Z, {H}), (DEDelay, {I})}
- connects = {(C, {D, F}), (E, {G}), (H, {J}), (I, {A, B})}
- triggers = {(A, {}), (B, {C}), (D, {E}), (F, {H}), (G, {})}
- before = {(A, {B}), (B, {}), (D, {}), (F, {G}), (G, {})}

From (3) we obtain the following constraints:

- rank(B) < rank(D)
- rank(B) < rank(F)
- rank(D) < rank(G)
- rank(F) < rank(J)

and similarly from (4):

- rank(A) < rank(B)
- rank(F) < rank(G)

These constraints can indeed be satisfied. For example, the following function will work, rank = {(A, 0), (B, 1), (D, 2), (F, 3), (G, 4), (J, 5)}.

2.3.4. Determinacy

Recall that our scheme of handling simultaneous events is designed to preserve determinacy and to give result that corresponds well with intuition. The intuition is that *an actor should be able to see all simultaneous events in its ports during a single firing.*

2.3.4.1. The triggers relation

The following illustrates how the triggers relation affects the handling of simultaneous events. Again, refer to figure 6. Suppose actor X is fired and it produces an event from the output port C. This

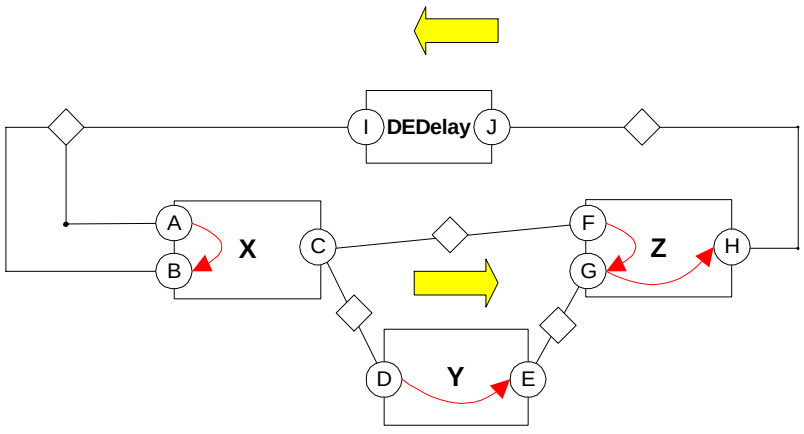


Figure 6. Modified zero-delay directed loop

will result in simultaneous events at input ports F and D. Without our scheme of ranking the input ports, we have the choice of either firing actor Z or actor Y, and some discrete-event simulators make this choice arbitrarily. Now also suppose that actor Y produces a simultaneous event at the output port E due to the input event at the input port D.

If actor Y were to be fired first, then actor Z will see an additional event at input port G as well as one at input port F when it is fired. On the other hand, if actor Z were to be fired first, then it will only see one event at input port F. So, making an arbitrary choice of whether to fire actor Y or actor Z first will result in non-determinacy.

Under topological sort, the `D.triggers(E)` relation translates to *rank(D) < rank(F)*. Therefore, our scheme ensures determinacy by giving a higher priority to input port D compared to input port F, which results in firing actor Y first. Choosing to fire actor Y first is more intuitive because we want actors to see the most possible number of tokens when fired, e.g. actor Z sees both events when fired.

2.3.4.2. The before relation

The before relation affects the handling of simultaneous events in a slightly more subtle way. Consider the topology shown in figure 7. Suppose actor X is fired and it produces an event from its output

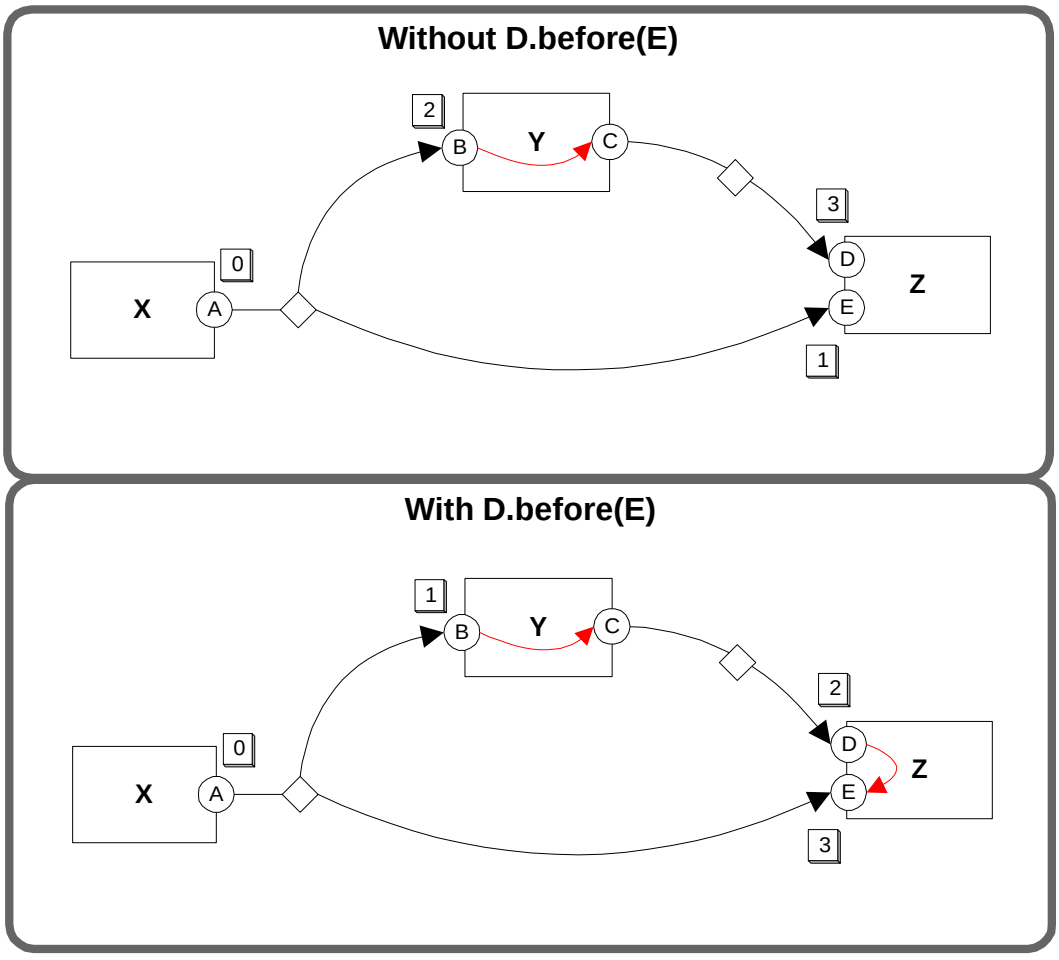


Figure 7. The comparison of the result of the topological sort in the absence of the before relation.

port A. This will result in simultaneous events in the input ports B and E, denoted by E_B and E_E, respectively. Suppose that the semantics of actor Z is that *whenever it sees events in the input port E, it want to make sure that it also sees all possible simultaneous events in the input port D.* (This is similar to the WaitingTime actor shown in figure 4.) The rank of the input ports obtained from the topological sort is shown in the raised boxes near the input ports.

Observe the difference in the sequence of actor firing when the before relation is omitted and when it is included. In the first case, the input port E has higher priority than the input port B. This results in the firing of actor Z, followed by the firing of actor Y. Notice that actor Y will react instantenously, and will produce an event, E_D, at the input port D. The time stamp of event E_D is equal to that of event E_E, yet E_D is presented to Z after E_E is presented and consumed by Z in an earlier firing. This is clearly a violation of the semantics of actor Z described above. Depending on how the code for actor Z is written, this may result in non-determinacy or even an exception being thrown.

In the latter case, the input port B has higher priority than the input port E. Therefore, actor Y will be fired first, followed by actor Z. When actor Z is fired, it will correctly see both simultaneous events in the input ports D and E. The code for actor Z can then be written in a way that assumes the presence of all simultaneous events in the input port D whenever there are events at the input port E. Therefore, the before relation offers a way for an actor to assert precedence relation between its input ports and the scheduler to sequence actor firings systematically and correctly.

2.3.5. Domain polymorphic actors

The triggers and before relations between ports are specific to the DE domain, therefore domain polymorphic actors do not know of and do not specify these relations. Given a domain polymorphic actor A, the DE director assumes the following triggers relations:

$$\forall inport \in inputs(A), \forall outport \in outputs(A), outport \in triggers(inport) \quad (6)$$

That is, there is a trigger relation between each input port to each output port of the domain polymorphic actor A. This is the right behaviour, since some domains in Ptolemy II are not timed-domains, and therefore, a non-DE actor (e.g. domain polymorphic actor) should be treated as a functional actor with zero delay in computation time. Note that the DE director does not assume any before relations, since it is assumed that the order that these functional actors sees their arguments is irrelevant.

3.Overview of The Software Architecture

The UML static structure diagram for the DE kernel package is shown in figure 8. For users interested in building a model using the Ptolemy II DE domain, the important classes are DEDirector, DEActor and DEIOPort.

At the heart of DEDirector is a global event queue that sorts events according to their time stamps and ranks (i.e. topological depth). In a typical DE simulation, this process proves to be the bottleneck. Therefore, by default, DEDirector uses an extremely efficient implementation of the global event queue, which is a calendar queue data structure [9]. 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. For extensibility, different implementations of the global event queue can be used and experimented with simply by implementing the DEEventQueue interface.

The DEActor class provides convenient methods to access time, since time is an essential part of a

timed domain like DE. Nonetheless, actors in a DE model are not required to be derived from the DEActor class. Simply deriving from TypedAtomicActor gives you the same capability, but without

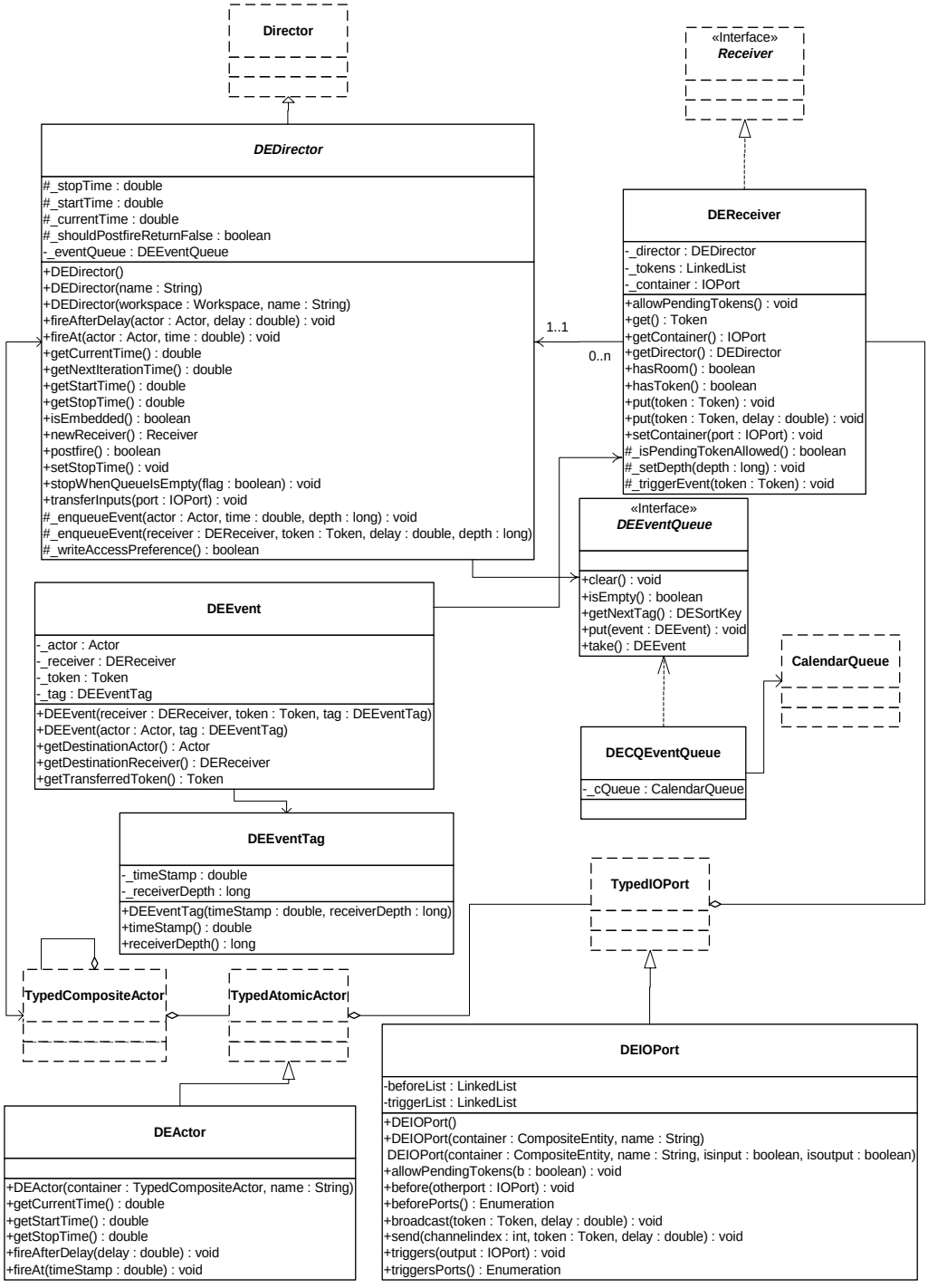


Figure 8. UML static diagram for the DE kernel package

the convenience. In the latter case, time is accessible through the director.

The DEIOPort class can be used by actors that are specialized to the discrete-event (DE) domain. It supports annotations that inform the scheduler about delays and about priorities for handling simultaneous inputs. 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 use a regular IOPort, and therefore cannot produce events in the future directly by sending it through output ports. Note that tokens sent through regular IOPort 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()` method of DEDirector. By calling the fireAt() method, the actor request the director to be refired in the future. The actor then produces the event during the refiring.

4.Writing a DE Actor

The Ptolemy II DE domain is shipped with a library of actors contained in the de.lib package. Ptolemy II also supports domain polymorphic actors. These domain polymorphic actors can be used in various compatible domains, including the DE domain. Recall that domain polymorphic actors are assumed to be functional (i.e. zero delay in computation time) by the DE director.

One key point to remember when writing an actor is that state changes (e.g. change of the values of instance variables) should only occur in the postfire() method. Most of the work in an actor occurs is in its fire() method. So, the pattern is that in the fire() method, the actor performs a computation and produces outputs based on the state variables. If state variables need to be updated based on values contained by variables local to the `fire()` method, then the updated values have to be saved first in temporary state variables, and later update the state variables in the `postfire()` method. The reason for this is to accomodate some domains in Ptolemy II (e.g. CT domain) that might invoke fire() multiple times to iterate to a fixed point, and the state of actors needs to be fixed between these fixed-point iterations.

In rare cases, actors can also decide that they do not want to be fired, even though there are events destined at them. In that case, they can return false in the prefire() method. For example, DEServer returns false in its prefire() method if it is fired while the server is still busy processing the previous request.

Recall that one iteration of a DE model processes all events in the global event queue with time stamps equal to the current time. Since there might be simultaneous events at the current time destined to multiple actors, one iteration of a DE model may translates into multiple iterations of the contained actors. The choice of which actors to fire first depends on the topological sort of input ports as explained above. An iteration of the contained actor is defined to be the sequence of method calls: prefire(), fire() and postfire() of the actor. This complicated scheduling is hidden underneath the Ptolemy II DE domain infrastructure. During its iteration, an actor only needs to worry about the following:

- Obtain tokens from its input ports. These tokens along with the current time as their time stamp act as input events for the actor.
- Produce output events into its output ports. These events have time stamps greater than or equal to the current time.
- Update its current state in the postfire() method.

4.1. Basic Examples

```
/* An actor that generates events according to Poisson process. */

package ptolemy.domains.de.lib;

import ptolemy.actor.*;
import ptolemy.domains.de.kernel.*;
import ptolemy.kernel.*;
import ptolemy.kernel.util.*;
import ptolemy.data.*;
import ptolemy.data.expr.*;
import java.util.Enumeration;

//////////////////////////////////////////////////////////////////////////
//// Poisson
/**
Generate events according to Poisson process. The first event is
always at time zero. The mean inter-arrival time and value of the
events are given as parameters.
FIXME: at current implementation, the first event is not at time zero, rather
it'll depend on the initialization value of current time field in the
director.

@author Lukito Muliadi
@version $Id$
*/
public class Poisson extends DEActor {

    /** Construct a Poisson actor with the default parameters.
     *  @param container The composite actor that this actor belongs to.
     *  @param name The name of this actor.
     *  @exception IllegalActionException If the entity cannot be contained
     *   by the proposed container.
     *  @exception NameDuplicationException If the container already has an
     *   actor with this name.
     */
    public Poisson(TypedCompositeActor container, String name)
            throws NameDuplicationException, IllegalActionException  {
        super(container, name);
        output = new TypedIOPort(this, "output", false, true);
        meanTime = new Parameter(this, "lambda", new DoubleToken(0.1));
        value = new Parameter(this, "value");
    }

    ////////////////////////////////////////////////////////////////////
    ////                         Ports and Parameters              ////

    /** The output port, which has type at least that of the value
     *  parameter.
     */
    public TypedIOPort output;

    /** The mean inter-arrival time.  The value is a DoubleToken. */
    public Parameter meanTime;

    /** The output value, which can contain any type of token. */
    public Parameter value;

    ////////////////////////////////////////////////////////////////////
    ////                         public methods                    ////

    /** React to a change in a parameter.  This method is called by
     *  a contained parameter when its value changes. The method updates
     *  private variables that cache the parameter values.
```

```
     *  @param attribute The attribute that changed.
     */
    public void attributeChanged(Attribute attribute) {
        if (attribute == meanTime) {
            _lambda = ((DoubleToken)meanTime.getToken()).doubleValue();
        }
    }

    /** Clone the actor into the specified workspace. This calls the
     *  base class and then sets the ports and parameters.
     *  @param ws The workspace for the new object.
     *  @return A new actor.
     */
    public Object clone(Workspace ws) {
   try {
        Poisson newobj = (Poisson)super.clone(ws);
        newobj.output = (TypedIOPort)newobj.getPort("output");
        newobj.value = (Parameter)newobj.getAttribute("value");
        newobj.meanTime = (Parameter)newobj.getAttribute("meanTime");
        return newobj;
        } catch (CloneNotSupportedException ex) {
            // Errors should not occur here...
            throw new InternalErrorException(
                    "Clone failed: " + ex.getMessage());
        }
    }

    /** Produce the initializer event that will cause the generation of
     *  the first event at time zero.
     *  @exception IllegalActionException If there is no director.
     */
    public void initialize() throws IllegalActionException {
        super.initialize();
        double curTime = getCurrentTime();
   fireAt(0.0);
        _exp = -Math.log((1-Math.random()))*_lambda;

    }

    /** Produce an output event at the current time, and then schedule
     *  a firing in the future.
     *  @exception IllegalActionException If there is no director.
     */
    public void fire() throws IllegalActionException {

   // send a token via the output port.
   output.broadcast(value.getToken());

   fireAfterDelay(_exp);
    }

    /** Update the delay value for the next arrival event.
     *  @exception IllegalActionException If the base class throws it.
     */
    public boolean postfire() throws IllegalActionException {
        _exp = -Math.log((1-Math.random()))*_lambda;
        return super.postfire();
    }


    /** Return the type constraint that the output type must be
     *  greater than or equal to the type of the value parameter.
     *  If the the value parameter has not been set, then it is
     *  set to type BooleanToken with value <i>true</i>.
     */
    // FIXME: This should be simplified when infrastructure support improves.
    public Enumeration typeConstraints() {
   if (value.getToken() == null) {
```

```
            try {
                value.setToken(new BooleanToken(true));
            } catch (IllegalActionException ex) {
                // Should not occur (no type constraints)
                throw new InternalErrorException(ex.getMessage());
            }
    }
    LinkedList result = new LinkedList();
    Class paramType = value.getToken().getClass();
         Inequality ineq = new Inequality(new TypeConstant(paramType),
                 output.getTypeTerm());
    result.insertLast(ineq);
    return result.elements();
     }

     ////////////////////////////////////////////////////////////////
     ////                         private variables                ////

     private double _lambda;
     // An exponentially distributed random variable which denotes the
     // inter-arrival time.
     private double _exp;
}
```

Consider the code for the Poisson actor as shown above. The Poisson actor is a source actor that generates events that are distributed in time according to Poisson arrival process. Specifically, the time-interval between events is exponentially distributed.

We will consider each part of this actor. The constructor instantiates the ports and the parameters. The parameter infrastructure is described in the Ptolemy II design document [3]. Ports and parameters are defined as public fields to give easy access when building a model.

In the initialize method, the actor requests a refire at time 0.0 by method call fireAt(0.0). It also initializes the state variable. In our case, the state variable is a random variable with exponential distribution which denotes the time delay until the next output event.

In the fire method, the actor produces the output event and request a refire from the director after some specified delay that is determined by the state variable. The refire request is done by the method call fireAfterDelay(_exp). Notice that there are two ways to request a refire, fireAt() and fireAfterDelay().

In the postfire method, the state is updated. In our case, the state variable is updated with the new value for the exponentially distributed random variable.

4.2. DE thread actor

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 9. The two code listings implement two actors with equivalent behavior. The left one implements it as a threaded actor, while the 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 description, 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.* More formally, it can be described by a finite-state-machine with state diagram shown in

figure 10.

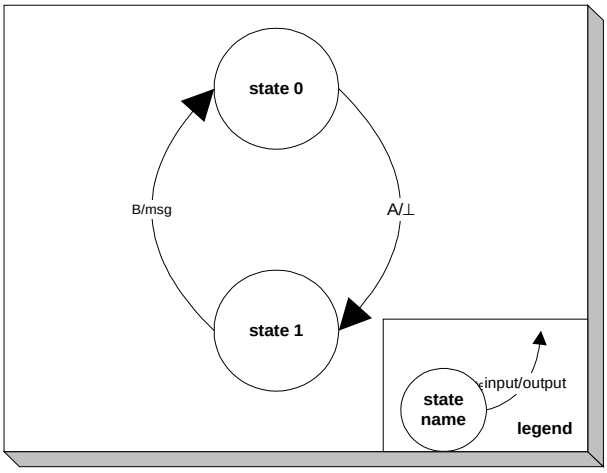
Note that the standard description needs a state variable `state`, unlike the case in the threaded description. In general the threaded description encodes the state information in the position of the code, while the standard description encodes it explicitly using state variables. While it is true that 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.

The infrastructure for this feature is shown in Figure 11. To write an actor in the threaded fashion, one simply derives from the DEThreadActor class and implements the `void run()` method. In most cases, the content of the `run()` method is enclosed in the infinite '`while(true)`' loop since most of useful threads 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 sus-

```
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 9. Code listings for two style of writing the ABRecognizer actor.



pends the thread until there is at least one input token in at least one of the input ports, while the latter suspends until there is at least one input token in any one of the specified input ports ignoring all other tokens. The word ‘new’ in waitForNewInputs() indicates that tokens are cleared from 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 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.

The code shown in figure 12 implements the run method of the ABRO actor with the following behavior:

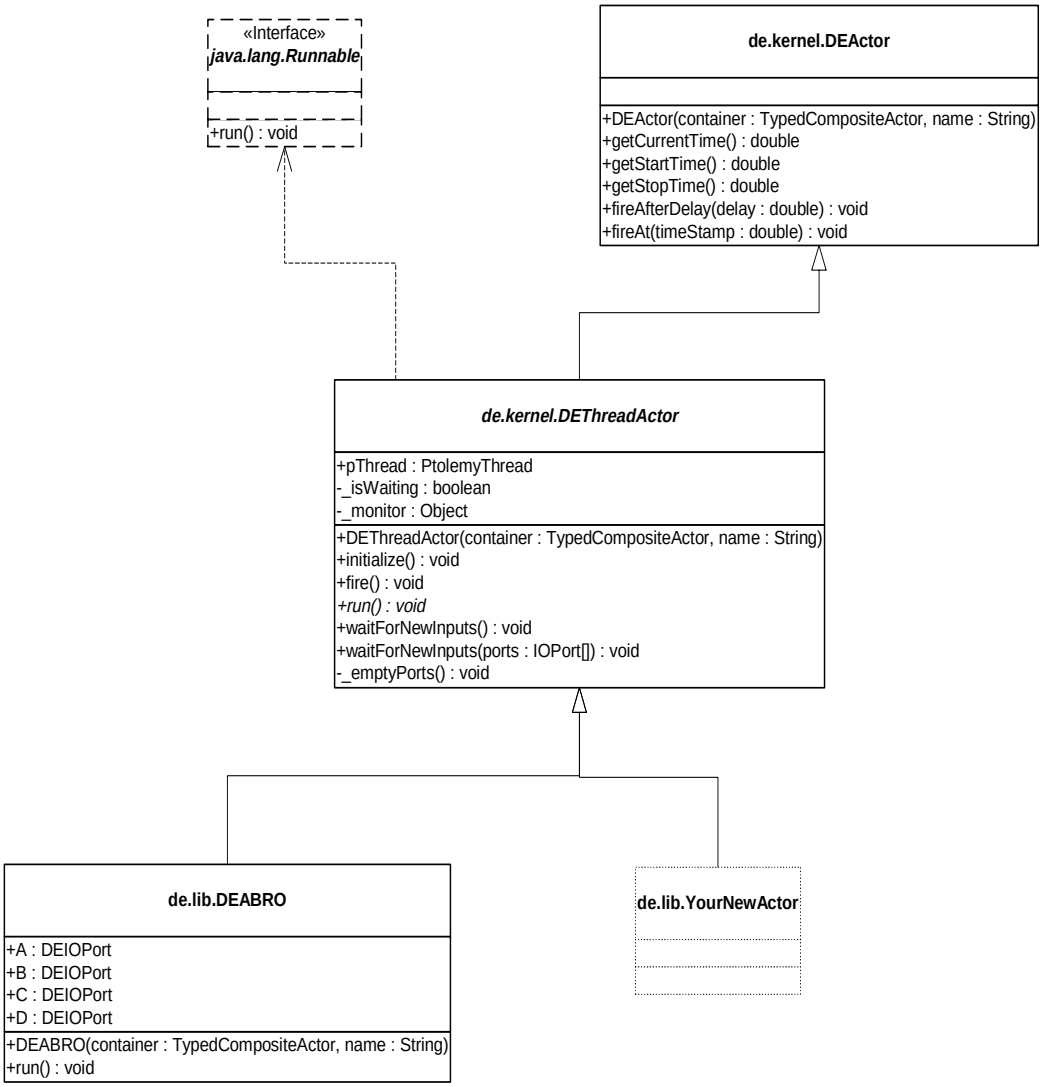


Figure 11. UML diagram for DEThreadActor

Emit an output O as soon as two inputs A and B have occured. *(7)*

Reset this behavior each time the input R occurs.

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 precludes parallelism.

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

```
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 12. The run() method of the ABRO actor.

threaded actor finish executing and suspend itself (e.g. by calling the waitForNewInputs() method). 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.

Future work in this area may involve extending the infrastructure to support various concurrency constructs, such as preemption, parallel execution, etc. It might also be interesting to explore new concurrency semantics similar to the threaded DE, but without the 'forced' serialization.

5.Composing DE with Other Domains

One of the major concepts in Ptolemy II is modeling heterogeneous systems through the use of 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 different model of computation. This mechanism is supported through the use of opaque composite actors. An example is shown in figure 13. The outermost domain is DE and it contains seven actors, two of them are opaque and composite. The opaque composite actors, by themselves, contain subsystems, which in this case are DE and CT.

5.1. 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

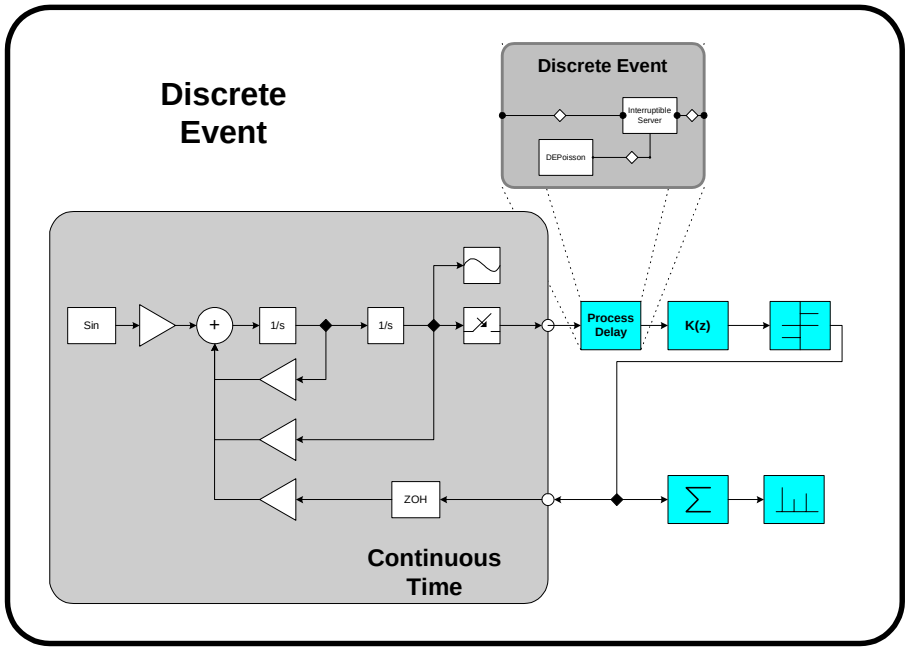


Figure 13. An example of heterogeneous and hierarchical composition. The CT subsystem and DE subsystem are inside an outermost DE system. This example is developed by Jie Liu [10].

the current time of the DE subsystem by t_{inner} and the current time of the outer domain by t_{outer}. The steps involved in the DE subsystem iteration is shown in figure 14. The steps are very similar to those of DE system iteration (figure 3), except for the two extra steps, which are:

- transferring tokens from the ports of the opaque composite actors into the ports of the contained DE subsystem
- requesting a refire at the smallest time stamp in the event queue of the DE subsystem.

The first of these 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 call super.transferInputs() (i.e. the default implementation in the Director class). 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

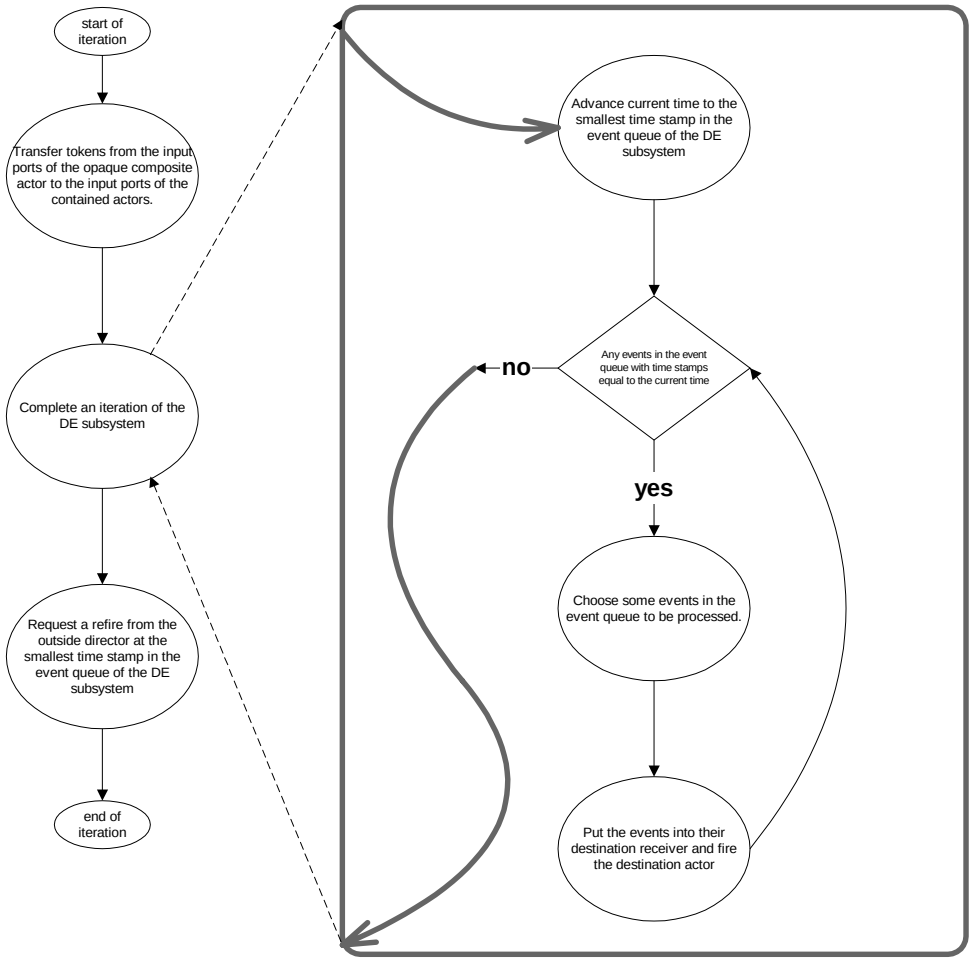


Figure 14. The flowchart for one iteration of the DE subsystem

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}.

The latter is done in the postfire() method of the DE 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}. Suppose we do not have this refire request, then the following is exactly what we don't want to happen. If the DE subsystem is fired at time 0.0 and it produces some events at time 0.1 and then the control of execution returns to the director of the outer domain. Note that t_{inner} is currently equal to 0.0. If there are no events destined to the DE subsystem until, say, time 10.0, then the DE subsystem will not be fired until t_{outer} equal to 10.0. Therefore the events at time 0.1 in the global event queue of the DE subsystem will not be processed until t_{outer} equal to 10.0. It is obvious that our approach will result in the refiring of the DE subsystem at t_{outer} equal to 0.1 and this is the correct thing to do.

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 be equal t_{outer} because we always request a refire according to that time stamp.

5.2. Another Domain inside DE

Due to its nature, the opaque composite actor 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.

6.Technical Details

6.1. Calendar Queue

Recall that in a typical DE simulator, the process of sorting events in the global event queue proves to be the bottleneck. In the DE domain in Ptolemy II, the global event queue uses the calendar queue implementation[5], by default. The calendar queue implementation is located in the `ptolemy.actor.util.CalendarQueue` class. The UML diagram is shown in figure (15). In the DE domain, appropriate interfaces have been designed such that different implementation of the global event queue can be used. This can be very useful, for example, when we want to do a performance comparison between different sorting algorithms. An overview of the calendar queue data structure is given here for completeness.

A calendar queue is a fast implementation of a priority queue. Its time complexity is O(1) in both enqueue and dequeue operations. It can be used to sort elements drawn from a totally ordered metric set. Recall that a set S is totally ordered if $\forall\ x,y \in S$, either $x \leq y$ or $y \geq x$. Also recall that a function d and the set S forms a metric space when d satisfies the following conditions:

- $\forall\ x,y \in S$, $d(x, y) \geq 0$.
- $\forall\ x,y \in S$, $d(x, y) = d(y, x)$.
- $\forall\ x,y,z \in S$, $d(x,z) \leq d(x,y) + d(y,z)$. (Triangle inequality)

For example, the set of all real numbers, R, is a totally ordered metric set with $d(x,y) = | x-y |$ ($|w|$ denotes the absolute value of w). In our case, events have time stamps drawn from the set R, and are

sorted based on their time stamps.

The Calendar Queue data structure achieves its performance by partitioning elements into a set of

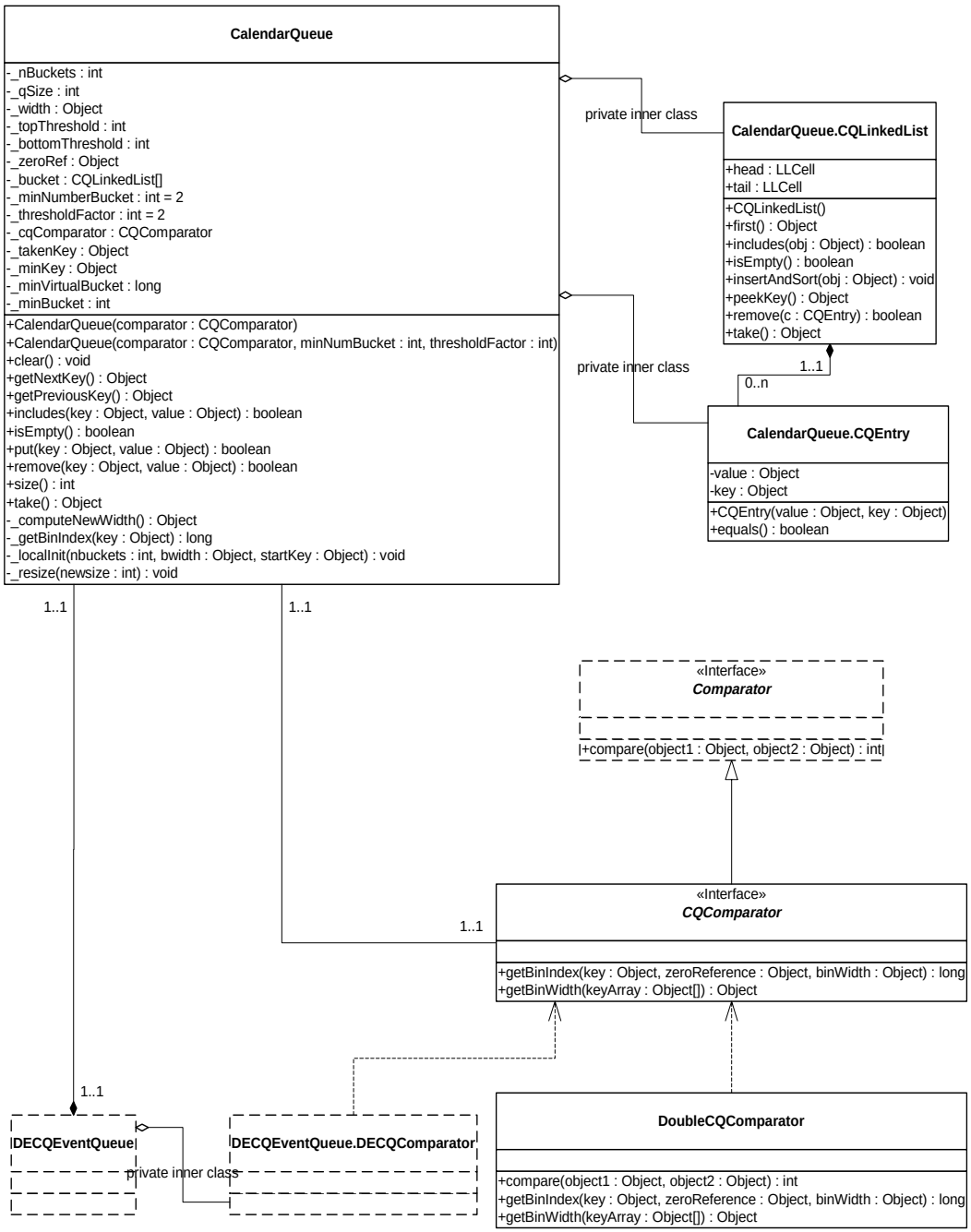


Figure 15. Calendar queue implementation in ptolemy.actor.util package

equally spaced bins. Then, *bubble sort* is performed separately in each bin on the reduced number of elements. It is similar in operation to a calendar where the set of days in a year is partitioned into months; one month for each page. Each bin is then conceptually equal to a monthly-page containing days in a particular month. An enqueue operation is similar to scheduling an appointment by first turning to the page for the appropriate month, then finding the date in that page. A dequeue operation is also similar in the same fashion.

To achieve best performance, the number of bins is dynamically adjusted to be less than the number of elements and greater than half of the number of elements. Therefore, on average, there are 1 to 2 elements in each bin. The spacing between bins is also chosen and maintained so that the distribution of the elements in the bins is as close to a uniform distribution as possible. Thus, we would like to avoid having some bins containing many elements, while others have few or no elements. It can be shown that the time complexity for finding the right bin, b_j, is O(1), while that of sorting the elements in bin b_j is equal to $O(n_j)$, where n_j is the number of element in bin b_j *only*. Therefore, the objective is to have as few elements as possible (e.g. 2 elements) in each bins, and the total time complexity becomes O(1). The price that we pay for being able to do extremely fast enqueue and dequeue operations comes in the form of complex algorithm needed to find the optimum number of bins and spacing between them. But this algorithm is only run when the number of elements changes by some factor (e.g. 2). So, we have shifted the sensitivity of the sorting algorithm from *the number of elements* in the queue to *the distribution and the variation of the number of elements* in the queue. It is also important to constraint the number of bins in order to not waste storage-space. Consequently, the bin-maintenance algorithm tries to achieve the optimal trade-off between time-complexity and storage-space.

Our implementation of the calendar queue algorithm uses an array of CQLinkedList as the implementation for the set of bins. As a design decision, we use our own customized version of CQLinkedList instead of the LinkedList class in the collections package. This is done for the sake of flexibility needed to effectively manipulate the data structure during the bin maintenance algorithm. The speedup compared to using the LinkedList class is around a factor of two.

The interface is designed with modularity in mind. The elements accepted by the queue are instances of class Object, which is as general as you can have in Java. One of the arguments for the constructors of the CalendarQueue class is an object implementing the CQComparator interface. This object gives the rule by which elements are sorted in the queue. In the DE domain, an inner class of the DECQEventQueue class, the DECQComparator provides rule by which events are sorted in the global event queue (which is implemented by the DECQEventQueue class).

Future work in the Ptolemy II CalendarQueue class might involve improving the bin-maintenance algorithm to account for the non-uniform distribution of elements.

6.2. Discrete Event Semantics

This section provides an overview of the operational semantics of the discrete event model of computation using the tagged signal model, which is a meta model for comparing models of computations [6]. We start by defining some notations.

6.2.1. Events and Signals

An *event* e is a member of the set $E = T \times V$, where T is the set of all tags and V is the set of all values. In our case, T = R, the set of all real numbers. A *signal* $s \in S$ is a set of events, therefore $S = \wp(E)$. A *functional signal* is a partial function from T to V. An N-tuple of signal $\mathbf{s} \in S^N$ is denoted by $\mathbf{s} = [s_1, s_2, ..., s_N]$. S^0 is the set with single element, which we denote by σ. An empty signal, denoted by λ, is one with no events. The N-tuple of empty signal is denoted by Λ^N.

6.2.2. Discrete and Zeno Signals

$T(s) \subseteq T$ is the set of *distinct* tags in a signal s. $T(\mathbf{s}) \subseteq T$ is the set of tags appearing in *any* signals in the tuple **s**. A *discrete signal* (or *tuple*) is one where T(s) (or T(**s**)) is order-isomorphic to a subset of integers. The set of discrete signals is denoted by S_d, while the set of discrete tuples is denoted by $S_d^{(N)}$. A signal s_Z is a *Zeno signal* if there exists $t_1, t_2 \in T(s_Z)$ such that $| T(s_Z) \cap [t_1, t_2] | = \infty$ (where |X| denotes the number of elements in the set X). I.e. there exists two tags such that there are inifinitely many tags between them. A *Zeno tuple* **s** is one which $s_1, s_2, ..., s_N \in S_d$, but $\mathbf{s} = [s_1, s_2, ..., s_N] \notin S_d^{(N)}$. For example,

$$T(s_1) = \{ 0, 1, 2, ... \} \Rightarrow s_1 \in S_d \tag{8}$$

$$T(s_2) = \{ 1/2, 3/4, 7/8, ... \} \Rightarrow s_2 \in S_d$$

$$T(\mathbf{s}) = \{ 0, 1/2, 3/4, 7/8, ..., 1, 2, ... \} \Rightarrow \mathbf{s} \notin S_d^{(2)} \text{ (but } \mathbf{s} \in S_d^2)$$

It can be shown that a Zeno signal (or tuple) is not a discrete signal (or tuple).

6.2.3. Merging signals

The *merge* of m signals is defined to be

$$M(s_1, s_2, ..., s_m) = s_1 \cup s_2 \cup ... \cup s_m \tag{9}$$

The merge of an m tuple is the merge of its component signals, i.e.

$$M(\mathbf{s}) = M([s_1, s_2, ..., s_m]) = M(s_1, s_2, ..., s_m) \tag{10}$$

It can be shown that if **s** is a Zeno tuple then M(**s**) is a Zeno signal.

The *two-way biased merge* of two signals is defined by

$$M_{2b}(s_1, s_2) = s_1 \cup (s_2 - \bar{s}_2) \tag{11}$$

where $\bar{s}_2$ is the largest subset of s_2 such that $T(\bar{s}_2) \subseteq T(s_1)$. I.e. if s_1 and s_2 have events with the same tag, the two-way biased merge of s_1 and s_2 includes the one from s_1 only. The *m-way biased merge* of m signals is defined recursively as follows

$$M_b(s) = s \tag{12}$$

$$M_b(s_1, s_2, ..., s_m) = M_{2b}(s_1, M_b(s_2, ..., s_m)) \tag{13}$$

E.g., the 3-way biased merge of 3 signals is defined by

$$M_b(s_1, s_2, s_3) = M_{2b}(s_1, M_b(s_2, s_3)) = M_{2b}(s_1, M_{2b}(s_2, Mb(s_3))) = M_{2b}(s_1, M_{2b}(s_2, s_3)) \tag{14}$$

Again, the biased merge of a tuple is the biased merge of its component signal, i.e.

$$M_b(\mathbf{s}) = M_b([s_1, s_2, ..., s_m]) = M_b(s_1, s_2, ..., s_m) \tag{15}$$

It can be shown that the biased merge of functional signals is a functional signal. On the other hand, the merge of functional signals is not a functional signal, in general.

6.2.4. Processes

A process P is a subset of S^N. An N-tuple of signals $\mathbf{s} \in S^N$ is said to *satisfy* the process, P, when $\mathbf{s} \in P$; **s** is called a behavior of the process. Thus, a process is a set of possible behaviors.

6.2.4.1. Projection

Let $J = [j_1, j_2, ..., j_M]$ be an ordered set of M distinct indexes in the range $1 \le j \le N$, define the projection $\pi_J(\mathbf{s})$ of $\mathbf{s} = [s_1, s_2, ..., s_N] \in S^N$ onto S^M by

$$\pi_J(\mathbf{s}) = [s_{j1}, s_{j2}, ..., s_{jM}] \tag{16}$$

Given a process $P \subseteq S^N$, define its projection onto S^M by

$$\pi_J(P) = \{\mathbf{s} \in S^M: \exists \mathbf{s}' \in P \text{ such that } \pi_I(\mathbf{s}') = \mathbf{s}\} \tag{17}$$

6.2.4.2. Inputs, outputs and functional processes

The index set {1, 2, ..., N} can be partitioned into three disjoint subsets I, O, and such that

$$\{1, ..., N\} = I \cup O \cup R \tag{18}$$

where I is the indices of the input signals, O the output signals and R the irrelevant signals.

A process P is said to be functional if there exists a function $F : S^{|I|} \rightarrow S^{|O|}$ relating the input signals to the ouput signal, such that for all $\mathbf{s} \in P$,

$$F(\pi_I(\mathbf{s})) = \pi_O(\mathbf{s}) \tag{19}$$

Therefore, a functional process P is completely characterized by the tuple (F, I, O, R).

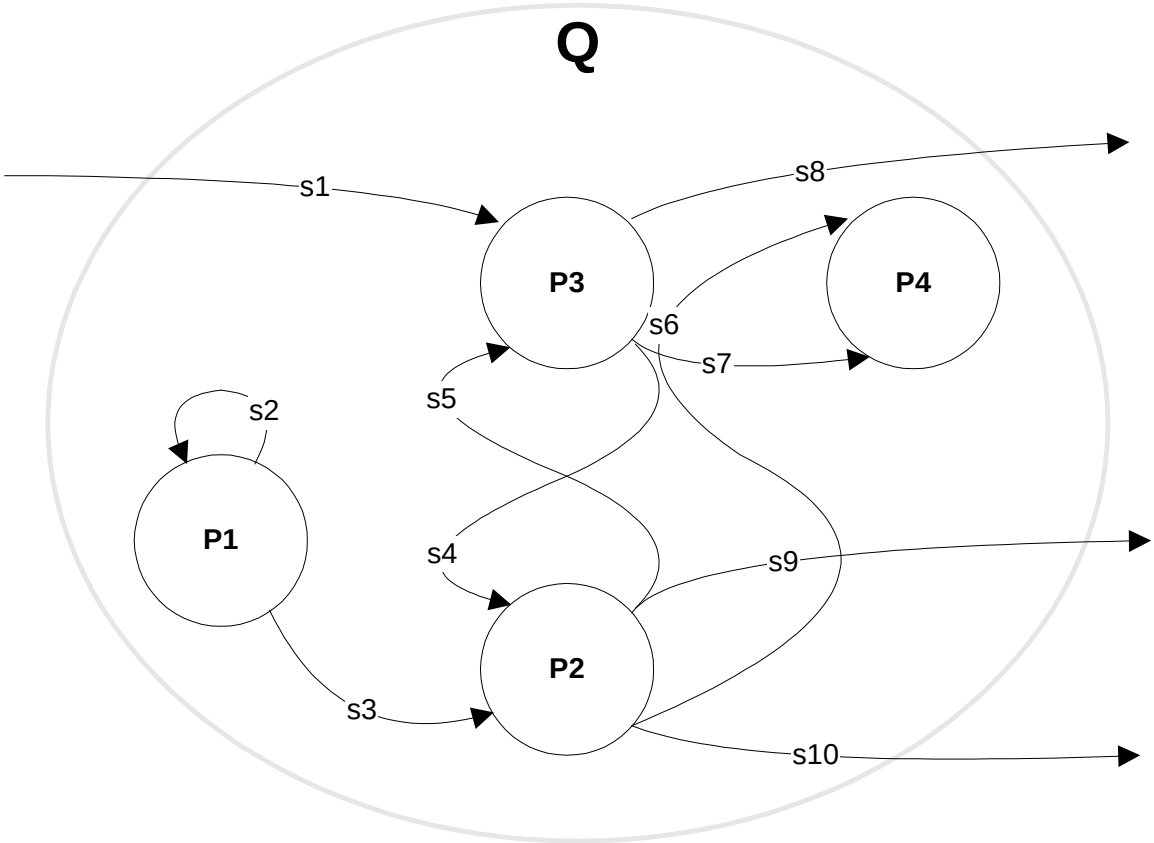


Figure 16. A network of processes

6.2.4.3. Source processes

A source process is a process with outputs but no inputs, e.g. P1 in figure 16. Note that the feedback loop is needed only in the implementation and therefore can be ignored in this denotational framework. A functional source process can be completely characterized by the tuple (F, ∅, O, R).

6.2.5. Causality in Discrete-Event Systems

A key concept in discrete-event systems is causality which intuitively means that the output events do not have time stamp less than that of the input events that caused them. For any $\mathbf{s} \in S_d^N$, let $\mathbf{s}(t) = [s_1(t), ..., s_N(t)]$ where $s_i(t)$ is the subset of s_i with tag equal to t. Define the Cantor metric as follows:

$$d(\mathbf{s}, \mathbf{s}') = \sup_t\{1/2^t : \mathbf{s}(t) \neq \mathbf{s}'(t), t \in T\} \quad (20)$$

Another definition is

$$d(\mathbf{s}, \mathbf{s}') = 1/2^{\tau} \quad (21)$$

where τ is equal to the smallest tag where **s** and **s'** differ. The smallest tag, τ, is equal to $-\infty$ when **s** and **s'** are identical. The Cantor metric along with set S results in a metric space.

For a function $F : S^m \rightarrow S^n$, F is causal if

$$\forall \mathbf{s}, \mathbf{s}' \in S^m, d(F(\mathbf{s}), F(\mathbf{s}')) \leq d(\mathbf{s}, \mathbf{s}') \quad (22)$$

It is strictly causal if

$$\forall \mathbf{s}, \mathbf{s}' \in S^m, d(F(\mathbf{s}), F(\mathbf{s}')) < d(\mathbf{s}, \mathbf{s}') \quad (23)$$

It is delta causal if there exists $\delta \in R$ such that

$$\delta < 1 \text{ and } \forall \mathbf{s}, \mathbf{s}' \in S^m, d(F(\mathbf{s}), F(\mathbf{s}')) \leq \delta d(\mathbf{s}, \mathbf{s}') \quad (24)$$

When F satisfies condition (24), then F is a *contraction mapping*.

A metric space is complete if every Cauchy sequence converges to a limit belonging to the metric space. It can be shown that $S_d^{(n)}$, the set of discrete tuples, is complete. The Banach fixed point theorem states that if $F : X \rightarrow X$ is a contraction mapping and X is complete, then there exists $x \in X$ such that $f(x) = x$. x is called the fixed point of F. Moreover, the theorem gives us a constructive way to find x, often called the fixed point algorithm. Start with any $x_0 \in X$; x is the limit of the sequence

$$x_1 = F(x_0), x_2 = F(x_1), ... \quad (25)$$

Consider a feedback loop shown in figure 17. Lee[7] shows that if the process P is functional and the function F obtained by (19) is delta causal, then the feedback loop has exactly one behavior (i.e. it is determinate). The behavior is found using the fixed point algorithm (25), which is exactly what VHDL and Verilog simulators do. The delta causal (contraction mapping) condition prevents Zeno condition where between two finite tags there are infinitely many tags. Zeno condition prevents the simulation to advance beyond a certain tag or time stamp. The delta causal condition can be relaxed by observing that it is sufficient that there exists a finite N such that F^N is delta causal.

6.2.6. Composition of Functional Processes

The composition of M processes, denoted by $\phi(P_1, P_2, ..., P_M)$, is treated as combining the constraints imposed by the processes $P_1, P_2, ..., P_M$. For example, $Q = \phi(P_1, P_2, P_3, P_4)$ in figure 16. Composition can be thought as combining M processes, $P_1, P_2, ..., P_M$, to construct a larger process, Q. I.e.

$\phi : \wp(S^{N1}) \times \wp(S^{N2}) \times ... \times \wp(S^{NM}) \rightarrow \wp(S^{N})$, where N_1, N_2, ..., N_M, N is the number of signals in the processes P_1, P_2, ..., P_M, Q, respectively.

We say that ϕ is *compositional* if the following four conditions hold:

- If P_1, P_2, ..., and P_M are functional processes, then $\phi(P_1, P_2, ..., P_M)$ is a functional process.
- If P_1, P_2, ..., and P_M are causal processes, then $\phi(P_1, P_2, ..., P_M)$ is a causal process.
- If P_1, P_2, ..., and P_M are strictly causal processes, then $\phi(P_1, P_2, ..., P_M)$ is a strictly causal process.
- If P_1, P_2, ..., and P_M are delta causal processes, then $\phi(P_1, P_2, ..., P_M)$ is a delta causal process.

We are interested in knowing whether some types of composition are compositional. Lee[6] has shown that acyclic composition (see figure 18) is compositional while cyclic composition (see figure 19) is compositional, only if the process in the feedback loop is delta causal, or some finite power of it is delta causal.

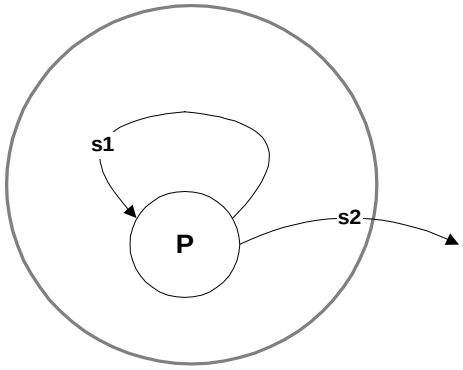


Figure 17. A process with loop connection.

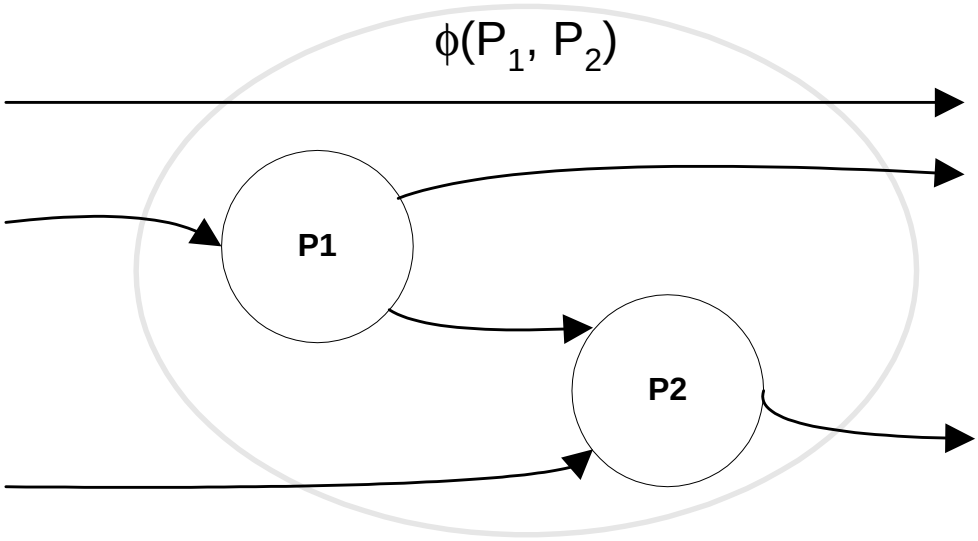


Figure 18. Acyclic composition

6.2.7. Operational Semantics

The operational semantics given corresponds to the one for the DE domain in Ptolemy II. The rank of input ports is encoded in the position of the signals in the signal tuple describing the composition. For example, consider figure 20. The ranks of the input ports is shown as numbers in the raised-boxes (Note that only the input ports have ranks). The composition can be described as a process Q and a behavior of the process Q is denoted by $\mathbf{s} = [s_1\ s_2\ ...\ s_N]$. For each input port with rank equal to *i*, the signal connected to the input port is named s_{i+1}. I.e. there is a one-to-one correspondence between the ranks of input ports and the name of the signals. The operational semantics resolves simultaneous events by choosing the ones that correspond to the signal with the smallest index.

Merge connections are disallowed in our framework. Consider the topology shown in figure 21. The signals s_1, s_2 and s_3 shares the same link, but they are all distinct. This discrepancy is not allowed in our framework. The fix is by introducing a merge actor (or process) that merges the incoming events and produce them in its output port.

Split connections are also disallowed in our framework. Consider the topology shown in figure 22. The input ports of the processes SinkA and SinkB have different ranks, but they are connected to the same signal, namely s_1. This breaks the one-to-one correspondence between the rank of input ports and the name of the signals. One way to fix this topology is by inserting a split actor (or process) that takes incoming events and produces a copy for each output port.

There is one aspect in the Ptolemy II DE domain that is not captured by this operational semantics. The order of simultaneous events destined to the same port are lost in the operational semantics. For example, consider figure 23. Suppose when the Source actor is fired, it produces three simultaneous events, namely E_1, E_2 and E_3 by invoking the `send()` method of the IOPort class three times with arguments v_1, v_2, and v_3, in that order (v_1, v_2 and v_3 are the values contained by the events E_1, E_2 and E_3, respectively). Although the time stamps of these events are equal, the implementation of the global event queue store them in the order they are sent. When the Sink actor is fired, it can invoke the `get()` method of the IOPort class and will get these three events in the same order these events were sent by

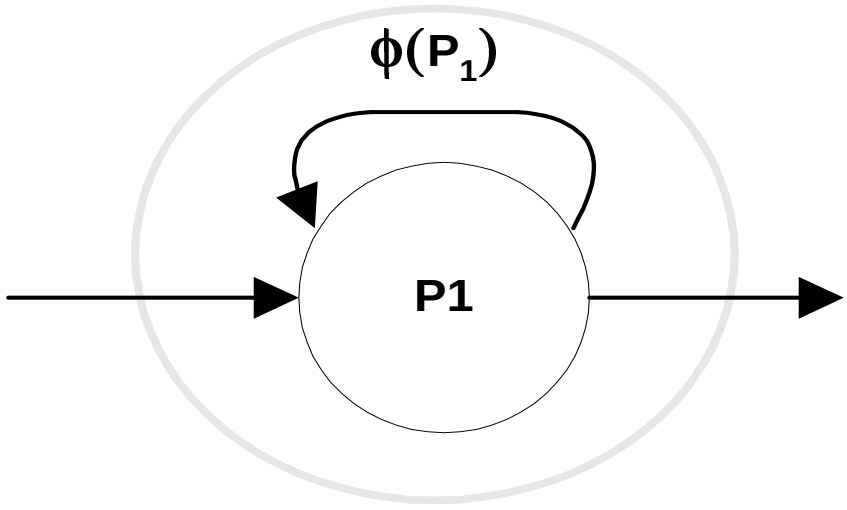


Figure 19. Cyclic composition

the Source actor. The tagged signal model does not contain an ordering relation among events with the same tags. (Recall that our tag denotes the time stamp which is an element from the set of real numbers) One way to capture this information is by extending our tag system, T = R×N, where N denotes the set of positive integers. The tag of an event in the extended system will then be the ordered pair of a time stamp and a positive integer denoting the order the event was sent with respect to other simultaneous events.

The operational semantics are expressed in terms of the *firing function* of process, f. Let $\Gamma = \{ f : S^m \times T \rightarrow S^n \times \Gamma \}$ be the set of all firing functions for m-input, n-output processes. The firing function takes as arguments an m-tuple of input signals and a time stamp, and returns an n-tuple of output signals and a new firing function, called *continuation*. The firing function is required to satisfy the following stuttering condition

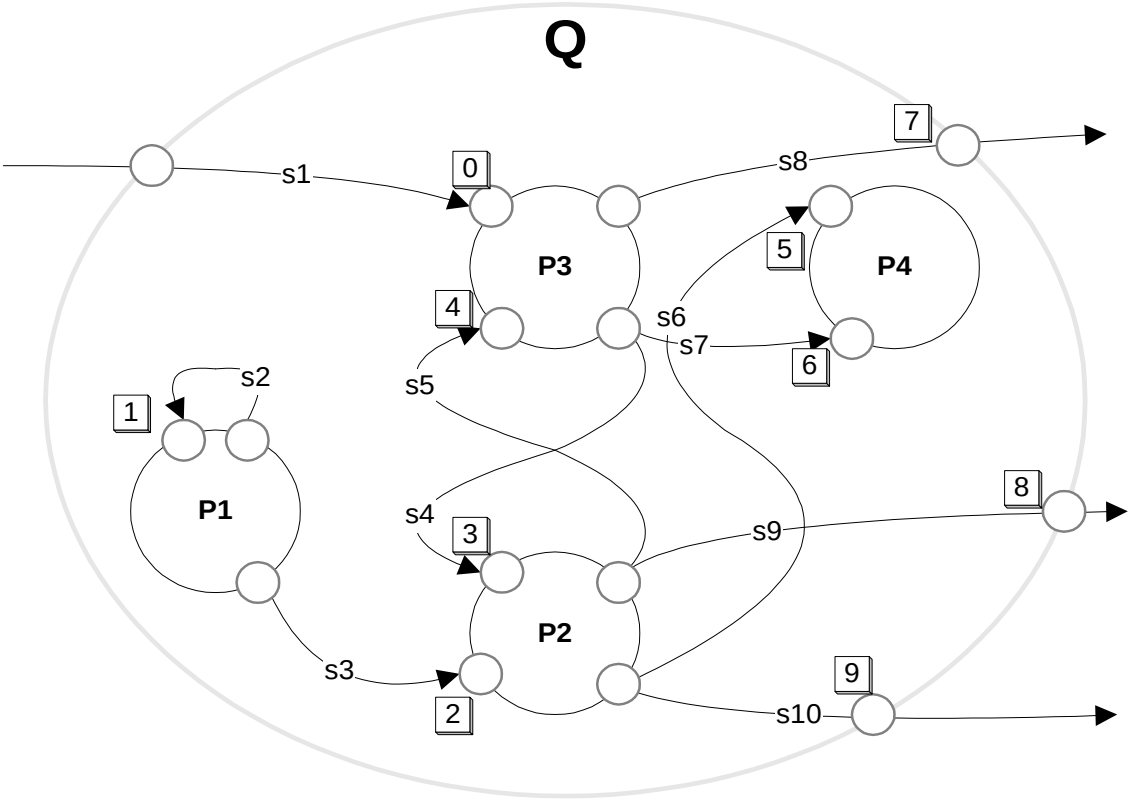


Figure 20. One-to-one correspondence between the rank of input ports and the name of the signals in a composition.

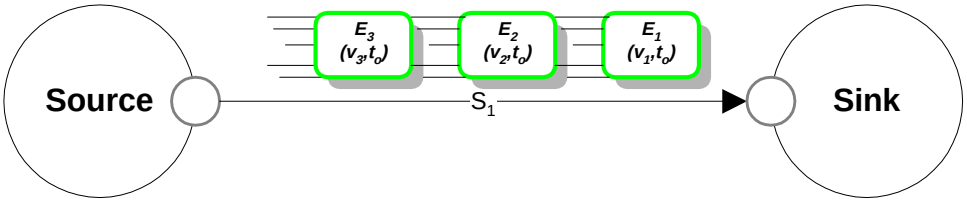


Figure 23. Simultaneous events, E_1, E_2, and E_3, destined to the same port.

$$f(\Lambda_m, t) = (\Lambda_n, f) \text{ for all } t \in T. \quad (26)$$

The firing function is a good implementation model of actors with m inputs and n outputs. When an actor is fired, its given input events and the current time. The actor may then produce output events and maybe change its state. The change of state is modeled by the *continuation* function, so on the next firing the firing function might do something different due to the state change.

6.2.7.1. Relationship between the firing function and the process function

We will give procedure that given the *firing function*, $f : S^m \times T \rightarrow S^n \times \Gamma$, returns the *process function* $F : S^m \rightarrow S^n$. Consider a single actor with m inputs and n outputs. Let $\mathbf{s} \in S^m$ be the input. Construct $\mathbf{s}' = F(\mathbf{s}) \in S^n$ according to procedure (27).

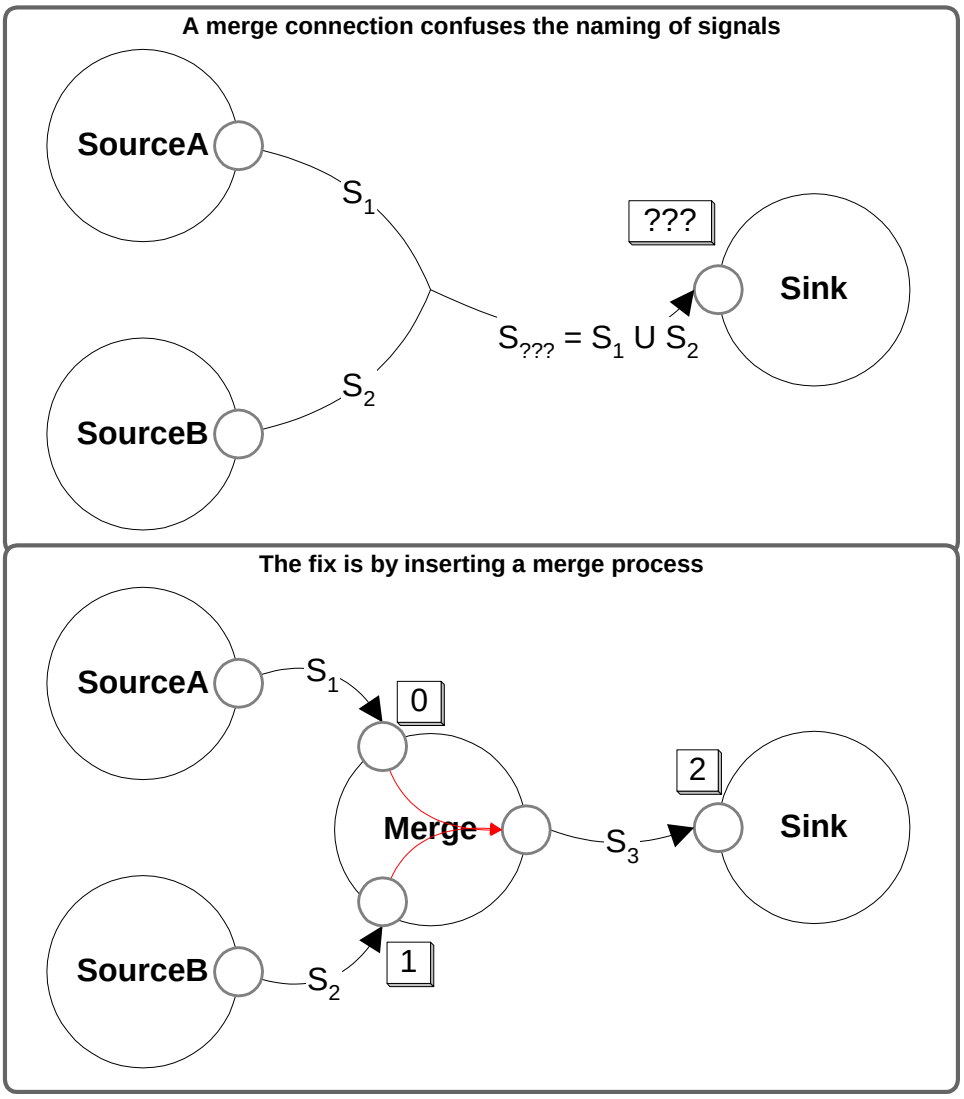


Figure 21. A topology containing a merge connection. The appropriate fix is shown.

```
// The procedure fills s' with the resulting output events due to
// the input events s.
```

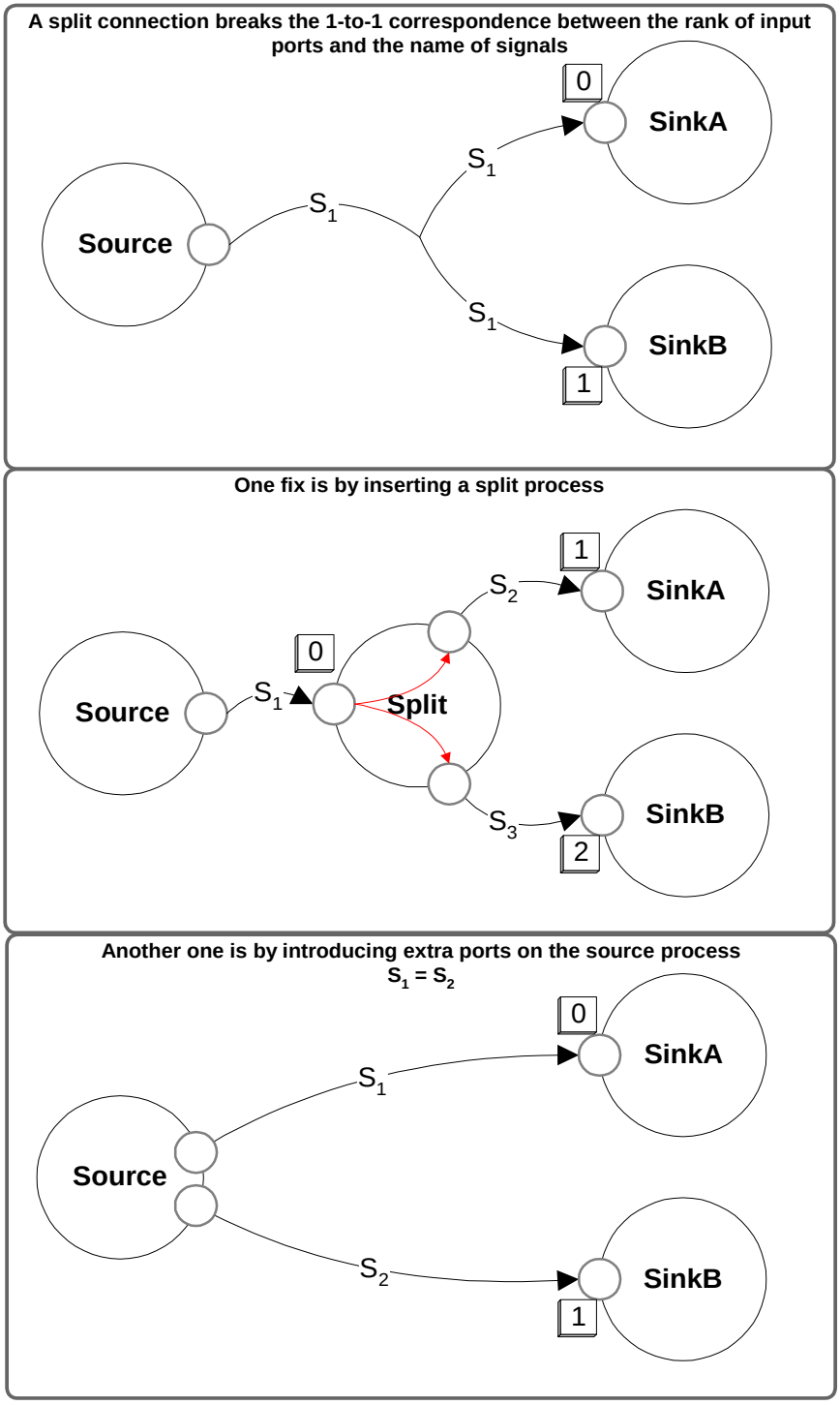


Figure 22. A topology containing a split connection. Two fixes are shown.

```
// Initialize s' to an empty tuple.
s' = Λn
// Keep looping until all events in the input signals s is consumed.
while (s ≠ Λm) {
   // Obtain the smallest time stamp in the input signal s
   let τ = min(T(Mb(s)))
   // s(τ) contains the events at time τ to be processed.
   // s'' stores the output events obtained from the actor firing at
   // time τ.
   // The firing function f is updated to the continuation function.
   let (s'', f) = f(s(τ), τ)
   // Remove the processed events, s(τ), from the input signal, s.
   let s = s - s(τ)
   // Append the output events from this firing, s'', to s'.
   let s' = s' ∪ s''
}
```
(27)

For a given firing function f, define its closure $\bar{f} \subseteq \Gamma$ to be the set of all firing functions reachable from f. It can be shown that F is causal (or strictly causal) if all firing functions in $\bar{f}$ are causal (or stricly causal). Also that F is delta causal if there exists a $\delta < 1$ such that for all $f' \in \bar{f}$, **s**, **s**' $\in S^m$, and $\tau \in T$,

$$d(\mathbf{q}, \mathbf{q}') \leq \delta d(\mathbf{s}, \mathbf{s}'), \tag{28}$$

where **q** and **q**' satisfies

$$(\mathbf{q}, f'') = f'(\mathbf{s}, \tau)$$

$$(\mathbf{q}', f''') = f'(\mathbf{s}', \tau)$$

Next we consider source processes with n outputs. Recall that source processes are implemented using feedback loop, so the firing function (which is much closer to the implementation than the process function) is of this form: $f : S \times T \rightarrow S^n \times \Gamma$. Note that the firing function takes as one of its arguments a 1-tuple of signal as the input from the feedback loop. Therefore, procedure (27) can be modified to obtain procedure (29) that given the *firing function*, $f : S \times T \rightarrow S^n \times \Gamma$, returns the *process function* $F : S^0 \rightarrow S^n$ (Note that S^0 contains only one element, namely σ). For a source actor with firing function f, construct **s'** = F(σ) as follows:

```
// The procedure fills s' with the resulting output events.
// Initialize s' to an empty tuple.
s' = Λn
// We arbitrarily choose that the first firing, thus the first
// output event, is at time zero.
τ = 0
// Source processes produce events indefinitely, hence infinite loop.
while (true) {
   // s'' stores the output events resulting from the firing at
   // time τ.
   // The firing function f is updated to the continuation function.
   let (s'', f) = f(πI(s'(τ)), τ)
   // Obtain the smallest time stamp in the output events s'' because
   // it is assumed that sources are implemented using feedback
   // loop such as in figure 17.
```

```
        let τ = min(T(s''))
        // Append the output events s'' to the result s'.
        let s' = s' ∪ s''
    }                                                              (29)
```

6.2.7.2. Operational Semantics for a Network of Discrete-Event Processes.

Finally, we can give the operational semantics for a network of discrete event processes, such as the one in figure 16. The signals in the network are numbered according to the topological sort of input ports as described in previous sections. The ranks of the input ports corresponds to the number assigned to the signals. Signals that are "forked" (or split) to different actors (e.g. s_4 and s_7 in figure 16) are given different names. (Note that this is equivalent to creating extra output ports on the source process, as shown in the bottom topology of figure 22.) Signals that are "joined" (or merged) are not allowed and therefore do not appear in figure 16. (A way to work around this merged-signal restriction is shown in figure 21.) These restriction are mainly to accomodate that the topological sort is done on the input ports rather than connections. The goal is such that each input port is connected to a signal-with a single and unique name, and moreover, the name indicates the rank of the input port.

Suppose there are N signals and M actors with firing functions f_1, f_2, ..., f_M, with non-empty input index sets I_1, ..., I_M, and output index sets O_1, ..., O_M. Let $\mathbf{s} \in S^N$ denotes the events initially present. In the Ptolemy II implementation this is achieved by queueing events during the initialization phase. The operational semantic is shown in procedure (30).

```
// The signal s serves as the global event queue for this procedure.
// The iteration continues until there are no more events in the
// global event queue.
while (s ≠ Λn) {
  // The variable τ denotes the current time. The following statement
  // reflects the current time advancement done by the director.
  let τ = min(T(Mb(s)))

  // This is the start of an iteration
  // An iteration is the processing of all events in the global
  // event queue at time τ.
  while (s(τ) ≠ Λn) {
    // j is the index of the signal with lowest rank that contains
    // input events at time τ.
    let j = min{k ∈ {1, 2, ..., N} : sk(τ) ≠ λ }

    // i is the index of the process that takes the signal Sj as
    // one of its input signals.
    let i ∈ {1, 2, ..., M} : j ∈ Ii

    // πIi(s(τ)) is the events at time τ that are destined for
    // actor i.
    // Actor i is fired at time τ and its firing function is
    // updated.
    // The output events produced are stored in s'' ∈ SOi
    let (s'', fi) = fi(πIi(s(τ)), τ)
```

```
        // The function Select_{N,Ii} : S^N → S^N returns an N-tuple signal,
        // s_ret, similar to the argument, except that (s_ret)_k = λ for
        // k ∉ I_i.
        // The function Expand_{Oi,N} : S^|Oi| → S^N returns an N-tuple signal
        // with the matching components equal to those of the argument
        // and the the rest of the components is λ.

        // Select_{N,Ii}(s(τ)) denotes the events processed by actor i, while
        // Expand_{Oi,N}(s'') denotes the events produced by actor i
        // Update the global event queue appropriately.
        let s = s - Select_{N,Ii}(s(τ)) ∪ Expand_{Oi,N}(s'')
      }
      // The end of one iteration, because there is no more events
      // in the global event queue at time τ.
  }                                                                    (30)
```

where

$$\text{Select}_{N,I}(\mathbf{s}) = [p_1, p_2, ..., p_N] \in S^N \tag{31}$$

with

$$\begin{aligned} p_k &= s_k \text{ for } k \in I \\ &= \lambda \text{ for } k \notin I \end{aligned}$$

and where

$$\text{Expand}_{O,N}(\mathbf{s} = [s_{o1}, s_{o2}, ..., s_{o|O|}]) = [p_1, p_2, ..., p_N] \in S^N \tag{32}$$

with

$$\begin{aligned} O &= [o_1, o_2, ..., o_{|O|}], \text{ where } |O| \text{ denotes the cardinality of } O \\ pk &= \lambda \text{ for } k \notin O \\ &= s_{oh} \text{ for } k = o_h \end{aligned}$$

7.Bibliography

[1] The Ptolemy II project, http://ptolemy.eecs.berkeley.edu/ptolemyII.

[2] E.A. Lee, Overview of The Ptolemy Project, UCB/ERL Memorandum M98/72.

[3] Davis J, et al, Heterogeneous Concurrent Modeling and Design in Java, Memorandum UCB/ERL M98/72, EECS, University of California, Berkeley, CA, USA 94720, November 23 1998.

[4] J.T. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt, “Ptolemy: A Framework for Simulating and Prototyping heterogeneous Systems, “ Int. Journal of Computer Simulation, special issue on “Simulation Software Development, “ vol. 4. pp. 155-182, April, 1994.

[5] Randy Brown, “CalendarQueue: A Fast Priority Queue Implementation for the Simulating Event Set Problem”, Communications of the ACM, October 1988, volume 31, Number 10.

[6] E. A. Lee and A. Sangiovanni-Vincentelli, “A Framework for Comparing Models of Computation”, March 12, 1998. (Revised from ERL Memorandum UCB/ERL M97/22, University of California, Berkeley, CA 94270, January 30, 1997)

[7] Edward A. Lee, “Modeling Concurrent Real-Time Processes Using Discrete Events”, UCB/ERL Memorandum M98/7, March 4th 1998

[8] Jerry Banks, et al, “Discrete-Event System Simulation”, Second Edition, Prentice-Hall, Inc, 1996

[9] Randy Brown, “CalendarQueue: A Fast Priority Queue Implementation for The Simulation Event Set Problem”, Communications of the ACM, October 1998, Volume 31, Number 10.

[10] Jie Liu, “Continuous Time and Mixed-Signal Simulation in Ptolemy II”, MS Report, UCB/ERL Memorandum M98/74, Dept. of EECS, University of California, Berkeley, CA 94720, December 1998.