Chapter 7 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.

7

DDE Domain

Author: *John S. Davis II*

7.1 Introduction

The distributed discrete-event (DDE) model of computation incorporates a distributed notion of time into a dataflow style of computation. Time progresses in a DDE model when the actors in the model execute and communicate. Actors in a DDE model communicate by sending messages through bounded, FIFO channels. Time in a DDE model is distributed and localized, and the actors of a DDE model each maintain their own local notion of the current time. Local time information is shared between two connected actors whenever a communication between said actors occurs. Conversely, communication between two connected actors can occur only when constraints on the relative local time information of the actors are adhered to.

The DDE domain is based on distributed discrete-event processing and leverages a wealth of research devoted to this topic. Some tutorial publications on this topic are [28][41][62][106]. The DDE domain implements a specific variant of distributed discrete event systems (DDES) as expounded by Chandy and Misra [28]. The domain serves as a framework for studying DDES with two special emphases. First we consider DDES from a dataflow perspective; we view DDE as an implementation of the Kahn dataflow model [64] with distributed time added on top. Second we study DDES not with the goal of improving execution speed (as has been the case traditionally). Instead we study DDES to learn its usefulness in modeling and designing systems that are timed and distributed.

7.2 Using DDE

The DDE domain is typed so that actors used in a model must be derived from TypedAtomicActor. The DDE domain is designed to use both DDE specific actors as well as domain-polymorphic actors. DDE specific actors take advantage of DDEActor and DDEIOPort which are designed to provide convenient support for specifying time when producing and consuming tokens. The DDE domain also has special restrictions on how feedback is specified in models.

7.2.1 DDEActor

The DDE model of computation makes one very strong assumption about the execution of an actor: *all input ports of an actor operating in a DDE model must be regularly polled to determine which input channel has the oldest pending event*. Any actor that adheres to this assumption can operate in a DDE model. Thus, many polymorphic actors found in ptolemy/actor/[lib, gui] are suitable for operation in DDE models. For convenience, DDEActor was developed to simplify the construction of actors that have DDE semantics. DDEActor has two key methods as follows:

getNextToken(). This method polls each input port of an actor and returns the (non-Null) token that represents the oldest event. This method blocks accordingly as outlined in section 7.3.1 (Communicating Time).

getLastPort(). This method returns the input IOPort from which the last (non-Null) token was consumed. This method presumes that getNextToken() is being used for token consumption.

7.2.2 DDEIOPort

DDEIOPort extends TypedIOPort with parameters for specifying time stamp values of tokens that are being sent to neighboring actors. Since DDEIOPort extends TypedIOPort, use of DDEIOPorts will not violate the type resolution protocol. DDEIOPort is not necessary to facilitate communication between actors executing in a DDE model; standard TypedIOPorts are sufficient in most communication. DDEIOPorts become useful when the time stamp to be associated with an outgoing token is greater than the current time of the sending actor. Hence, DDEIOPorts are only useful in conjunction with delay actors (see “Enabling Communication: Advancing Time” on page 7-103, for a definition of delay actor). Most polymorphic actors available for Ptolemy II are not delay actors.

7.2.3 Feedback Topologies

In order to execute models with feedback cycles that will not deadlock, FeedBackDelay actors must be used. FeedBackDelay is found in the DDE kernel package. FeedBackDelay actors do not perform computation, but instead increment the time stamps of tokens that flow through them by a specified delay. The delay value of a FeedBackDelay actor must be chosen to be less than the delta time of the feedback cycle in which the FeedBackDelay actor is contained. Elaborate delay values can be specified by overriding the getDelay() method in subclasses of FeedBackDelay. An example can be found in ptolemy/domains/dde/demo/LocalZeno/ZenoDelay.java.

A difficulty found in feedback cycles occurs during the initialization of a model’s execution. In figure 7.1 we see that even if Actor B is a FeedBackDelay actor, the system will deadlock if the first event is created by *A* since *C* will block on an event from *B*. To alleviate this problem a special time stamp value has been reserved: PrioritizedTimedQueue.IGNORE. When an actor encounters an event with a time stamp of IGNORE (an *ignore event*), the actor will ignore the event and the input channel

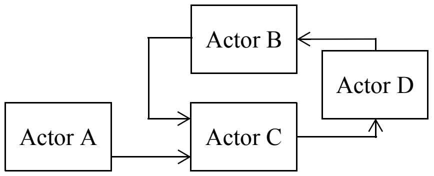

FIGURE 7.1. Initializing feedback topologies.

it is associated with. The actor then considers the other input channels in determining the next available event. After a non-ignore event is encountered and consumed by the actor, all ignore events will be cleared from the receivers. If all of an actor's input channels contain ignore events, then the actor will clear all ignore events and then proceed with normal operation.

The initialize method of FeedBackDelay produces an ignore event. Thus, in figure 7.1, if *B* is a FeedBackDelay actor, the ignore event it produces will be sent to *C*'s upper input channel allowing *C* to consume the first event from *A*. The production of null tokens and feedback delays will then be sufficient to continue execution from that point on. Note that the production of an ignore event by a FeedBackDelay actor serves as a major distinction between it and all other actors. *If a delay is desired simply to represent the computational delay of a given model, a FeedBackDelay actor should not be used*.

The intricate operation of ignore events requires special consideration when determining the position of a FeedBackDelay actor in a feedback cycle. A FeedBackDelay actor should be placed so that the ignore event it produces will be ignored in deference to the first real event that enters a feedback cycle. Thus, choosing actor *D* as a FeedBackDelay actor in figure 7.1 would not be useful given that the first real event entering the cycle is created by *A*.

7.3 Properties of the DDE domain

Operationally, the semantics of the DDE domain can be separated into two functionalities. The first functionality relates to how time advances during the communication of data and how communication proceeds via blocking reads and writes. The second functionality considers how a DDE model prevents deadlock due to local time dependencies. The technique for preventing deadlock involves the communication of *null messages* that consist solely of local time information.

7.3.1 Enabling Communication: Advancing Time

Communicating Tokens. A DDE model consists of a network of sequential actors that are connected via unidirectional, bounded, FIFO queues. Tokens are sent from a sending actor to a receiving actor by placing a token in the appropriate queue where the token is stored until the receiving actor consumes it. As in the process networks domain, the execution of each actor is controlled by a process. If a process attempts to read a token from a queue that is empty, then the process will block until a token becomes available on the channel. If a process attempts to write a token to a queue that is full, then the process will block until space becomes available for more tokens in that queue. Note that this blocking read/write paradigm is equivalent to the operational semantics found in non-timed process networks (PN) as implemented in Ptolemy II (see the PN Domain chapter).

If all processes in a DDE model simultaneously block, then the model deadlocks. If a deadlock is due to processes that are either waiting to read from an empty queue, *read blocks*, or waiting to write to a full queue, *write blocks*, then we say that the model has experienced *non-timed deadlock*. Non-timed deadlock is equivalent to the notion of deadlock found in bounded process networks scheduling problems as outlined by Parks [117]. If a non-timed deadlock is due to a model that consists solely of processes that are read blocked, then we say that a *real deadlock* has occurred and the model is terminated. If a non-timed deadlock is due to a model that consists of at least one process that is write blocked, then the capacity of the full queues are increased until deadlock no longer exists. Such deadlocks are called *artificial deadlock*, and the policy of increasing the capacity of full queues as shown by Parks can guarantee the execution of a model in bounded memory whenever possible.

Communicating Time. Each actor in a DDE model maintains a local notion of time. Any non-negative real number may serve as a valid value of time. As tokens are communicated between actors, time stamps are associated with each token. Whenever an actor consumes a token, the actor's *current time* is set to be equal to that of the consumed token's time stamp. The time stamp value applied to outgoing tokens of an actor is equivalent to that actor's *output time*. For actors that model a process in which there is delay between incoming time stamps and corresponding outgoing time stamps, then the output time is always greater than the current time; otherwise, the output time is equal to the current time. We refer to actors of the former case as *delay actors*.

For a given queue containing time stamped tokens, the time stamp of the first token currently contained by the queue is referred to as the *receiver time* of the queue. If a queue is empty, its receiver time is the value of the time stamp associated with the last token to flow through the queue, or 0.0 if no tokens have traveled through the queue. An actor may consume a token from an input queue given that the queue has a token available and the receiver time of the queue is less than the receiver times of all other input queues of the actor. If the queue with the smallest receiver time is empty, then the actor blocks until this queue receives a token, at which time the actor considers the updated receiver time in selecting a queue to read from. The *last time* of a queue is the time stamp of the last token to be placed in the queue. If no tokens have been placed in the queue, then the last time is 0.0

Figure 7.2 shows three actors, each with three input queues. Actor *A* has two tokens available on the top queue, no tokens available on the middle queue and one token available on the bottom queue. The receiver times of the top, middle and bottom queue are respectively, 17.0, 12.0 and 15.0. Since the queue with the minimum receiver time (the middle queue) is empty, *A* blocks on this queue before it proceeds. In the case of actor *B*, the minimum receiver time belongs to the bottom queue. Thus, *B* proceeds by consuming the token found on the bottom queue. After consuming this token, *B* compares all of its receiver times to determine which token it can consume next. Actor *C* is an example of an actor that contains multiple input queues with identical receiver times. To accommodate this situation, each actor assigns a unique priority to each input queue. An actor can consume a token from a queue if no other queue has a lower receiver time and if all queues that have an identical receiver time also have a lower priority.

Each receiver has a *completion time* that is set during the initialization of a model. The completion time of the receiver specifies the time after which the receiver will no longer operate. If the time stamp of the oldest token in a receiver exceeds the completion time, then that receiver will become *inactive*.

7.3.2 Maintaining Communication: Null Tokens

Deadlocks can occur in a DDE model in a form that differs from the deadlocks described in the previous section. This alternative form of deadlock occurs when an actor read blocks on an input port even though it contains other ports with tokens. The topology of a DDE model can lead to deadlock as read blocked actors wait on each other for time stamped tokens that will never appear. Figure 7.3 illus-

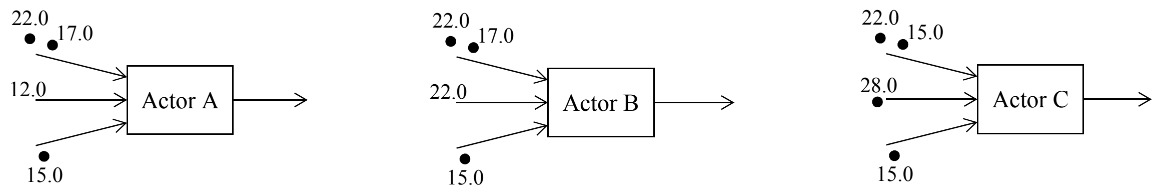

FIGURE 7.2. DDE actors and local time.

trates this problem. In this topology, consider a situation in which actor *A* only creates tokens on its lower output queue. This will lead to tokens being created on actor *C*'s output queue but no tokens will be created on *B*'s output queue (since *B* has no tokens to consume). This situation results in *D* read blocking indefinitely on its upper input queue even though it is clear that no tokens will ever flow through this queue. The result: *timed deadlock!* The situation shown in figure 7.3 is only one example of timed deadlock. In fact there are two types of timed deadlock: *feedforward* and *feedback*.

Figure 7.3 is an example of feedforward deadlock. Feedforward deadlock occurs when a set of connected actors are deadlocked such that all actors in the set are read blocked and at least one of the actors in the set is read blocked on an input queue that has a receiver time that is less than the local clock of the input queue's source actor. In the example shown above, the upper input queue of *B* has a receiver time of 0.0 even though the local clock of *A* has advanced to 8.0.

Feedback deadlock occurs when a set of cyclically connected actors are deadlocked such that all actors in the set are read blocked and at least one actor in the set, say actor *X*, is read blocked on an input queue that can read tokens which are directly or indirectly a result of output from that same actor (actor *X*). Figure 7.4 is an example of feedback timed deadlock. Note that *B* can not produce an output based on the consumption of the token timestamped at 5.0 because it must wait for a token on the upper input that depends on the output of *B*!

Preventing Feedforward Timed Deadlock. To address feedforward timed deadlock, *null tokens* are employed. A null token provides an actor with a means of communicating time advancement even though data (*real* tokens) are not being transmitted. Whenever an actor consumes a token, it places a null token on each of its output queues such that the time stamp of the null token is equal to the current time of the actor. Thus, if actor *A* of figure 7.3, produced a token on its lower output queue at time 5.0, it would also produce a null token on its upper output queue at time 5.0.

If an actor encounters a null token on one of its input queues, then the actor does the following. First it consumes the tokens of all other input queues it contains given that the other input queues have receiver times that are less than or equal to the time stamp of the null token. Next the actor removes the null token from the input queue and sets its current time to equal the time stamp of the null token. The actor then places null tokens time stamped to the current time on all output queues that have a last time that is less then the actor's current time. As an example, if *B* in figure 7.3 consumes a null token on its input with a time stamp of 5.0 then it would also produce a null token on its output with a time stamp

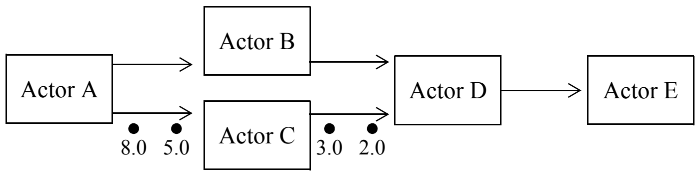

FIGURE 7.3. Timed deadlock (feedforward).

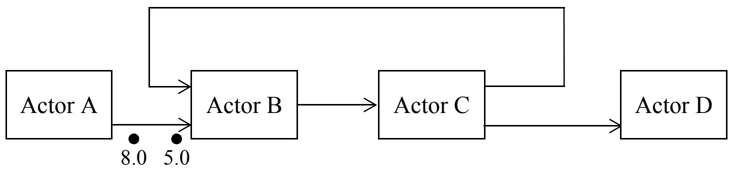

FIGURE 7.4. Timed deadlock (feedback).

of 5.0.

The result of using null tokens is that time information is evenly propagated through a model's topology. The beauty of null tokens is that they inform actors of inactivity in other components of a model without requiring centralized dissemination of this information. Given the use of null tokens, feedforward timed deadlock is prevented in the execution of DDE models. It is important to recognize that null tokens are used solely for the purpose of avoiding deadlocks. Null tokens do not represent any actual components of the physical system being modeled. Furthermore, the production of a null token that is the direct result of the consumption of a null token is not considered computation from the standpoint of the system being modeled. The idea of null tokens was first espoused by Chandy and Misra [28].

Preventing Feedback Timed Deadlock. We address feedback timed deadlock as follows. All feedback loops are required to have a cumulative time stamp increment that is greater than zero. In other words, feedback loops are required to contain delay actors. Peacock, Wong and Manning [118] have shown that a necessary condition for feedback timed deadlock is that a feedback loop must contain no delay actors. The delay value (delay = output time - current time) of a delay actor must be chosen wisely; it must be less then the smallest delta time of all other actors contained in the same feedback loop. *Delta time* is the difference between the time stamps of a token that is consumed by an actor and the corresponding token that is produced in direct response. If a system being modeled has characteristics that prevent a fixed, positive lower bound on delta time from being specified, then our approach can not solve feedback timed deadlock. Such a situation is referred to as a *Zeno condition*. An application involving an approximated Zeno condition is discussed in section 7.5 below.

The DDE software architecture provides one delay actor for use in preventing feedback timed deadlock: *FeedBackDelay*. See "Feedback Topologies" on page 7-102 for further details about this actor.

7.3.3 Alternative Distributed Discrete Event Methods

The field of distributed discrete event simulation, also referred to as parallel discrete event simulation (PDES), has been an active area of research since the late 1970's [28][41][62][106][118]. Recently there has been a resurgence of activity [8][9][16]. This is due in part to the wide availability of distributed frameworks for hosting simulations and the application of parallel simulation techniques to non-research oriented domains. For example, several WWW search engines are based on network of workstation technology.

The field of distributed discrete event simulation can be cast into two camps that are distinguished by the blocking read approach taken by the actors. One camp was introduced by Chandy and Misra [28][41][106][118] and is known as *conservative* blocking. The second camp was introduced by David Jefferson through the Jet Propulsion Laboratory Time Warp system and is referred to as the *optimistic* approach [62][41]. In certain problems, the optimistic approach executes faster than the conservative approach, nevertheless, the gains in speed result in significant increases in program memory. The conservative approach does not perform faster than the optimistic approach but it executes efficiently for all classes of discrete event systems. Given the modeling semantics emphasis of Ptolemy II, performance (speed) is not considered a premium. Furthermore, Ptolemy II's embedded systems emphasis suggests that memory constraints are likely to be strict. For these reasons, the implementation found in the DDE domain follows the conservative approach.

7.4 The DDE Software Architecture

For a model to have DDE semantics, it must have a DDEDirector controlling it. This ensures that the receivers in the ports are DDEReceivers. Each actor in a DDE model is under the control of a DDEThread. DDEThreads contain a TimeKeeper that manages the local notion of time that is associated with the DDEThread's actor.

7.4.1 Local Time Management

The UML diagram of the local time management system of the DDE domain is shown in figure 7.5 and consists of PrioritizedTimedQueue, DDEReceiver, DDEThread and TimeKeeper. Since time is localized, the DDEDirector does not have a direct role in this process. Note that DDEReceiver is derived from PrioritizedTimedQueue. The primary purpose of PrioritizedTimedQueue is to keep track of a receiver's local time information. DDEReceiver adds blocking read/write functionality to PrioritizedTimedQueue.

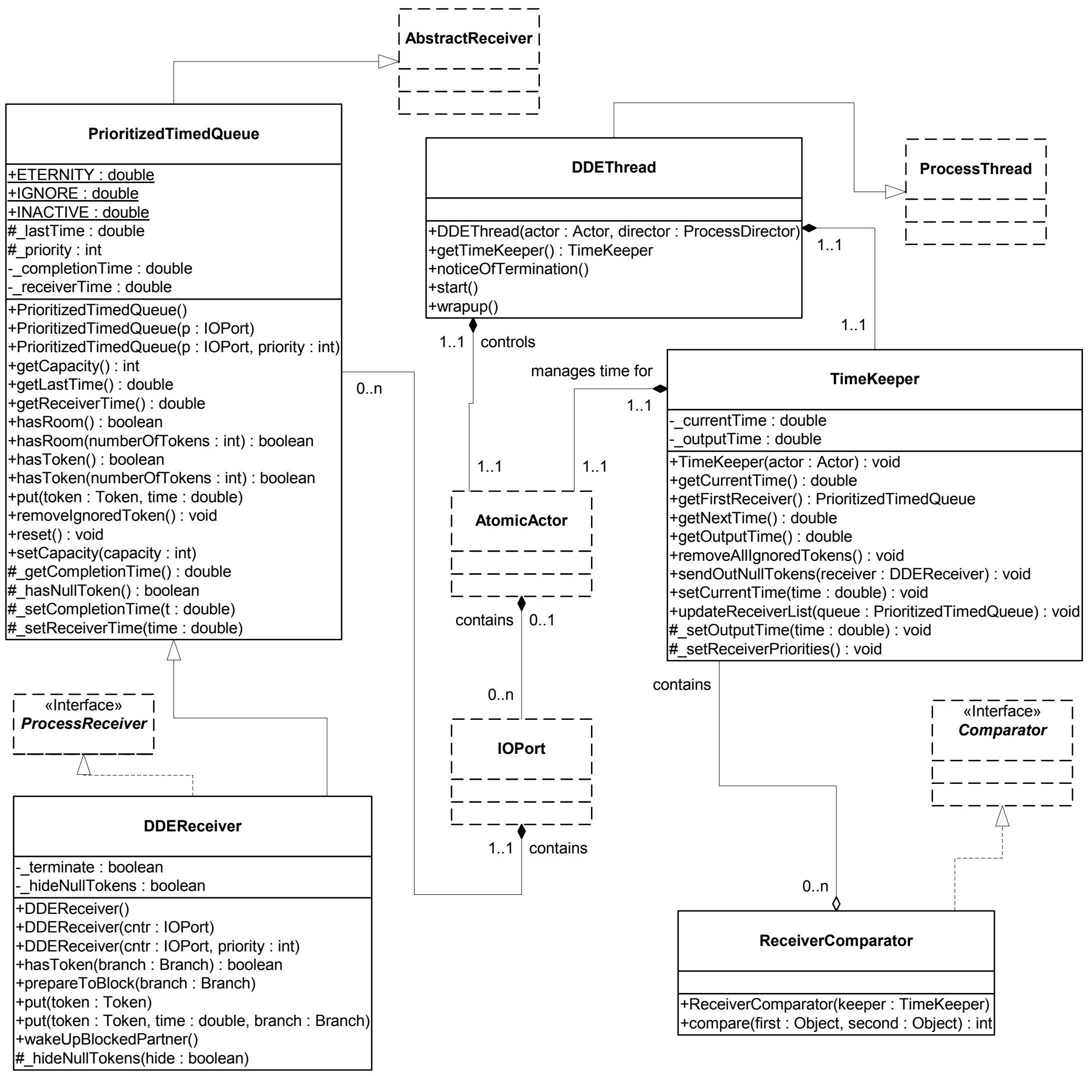
AbstractReceiver
PrioritizedTimedQueue
+ETERNITY : double
+IGNORE : double
+INACTIVE : double
#_lastTime : double
#_priority : int
-_completionTime : double
-_receiverTime : double
+PrioritizedTimedQueue()
+PrioritizedTimedQueue(p : IOPort)
+PrioritizedTimedQueue(p : IOPort, priority : int)
+getCapacity() : int
+getLastTime() : double
+getReceiverTime() : double
+hasRoom() : boolean
+hasRoom(numberOfTokens : int) : boolean
+hasToken() : boolean
+hasToken(numberOfTokens : int) : boolean
+put(token : Token, time : double)
+removeIgnoredToken() : void
+reset() : void
+setCapacity(capacity : int)
#_getCompletionTime() : double
#_hasNullToken() : boolean
#_setCompletionTime(t : double)
#_setReceiverTime(time : double)
DDEThread
+DDEThread(actor : Actor, director : ProcessDirector)
+getTimeKeeper() : TimeKeeper
+noticeOfTermination()
+start()
+wrapup()
ProcessThread
1..1
1..1
1..1
controls
manages time for
1..1
0..n
TimeKeeper
-_currentTime : double
-_outputTime : double
+TimeKeeper(actor : Actor) : void
+getCurrentTime() : double
+getFirstReceiver() : PrioritizedTimedQueue
+getNextTime() : double
+getOutputTime() : double
+removeAllIgnoredTokens() : void
+sendOutNullTokens(receiver : DDEReceiver) : void
+setCurrentTime(time : double) : void
+updateReceiverList(queue : PrioritizedTimedQueue) : void
#_setOutputTime(time : double) : void
#_setReceiverPriorities() : void
1..1
1..1
AtomicActor
contains
0..1
0..n
IOPort
1..1
contains
contains
«Interface»
ProcessReceiver
«Interface»
Comparator
0..n
DDEReceiver
-_terminate : boolean
-_hideNullTokens : boolean
+DDEReceiver()
+DDEReceiver(cntr : IOPort)
+DDEReceiver(cntr : IOPort, priority : int)
+hasToken(branch : Branch) : boolean
+prepareToBlock(branch : Branch)
+put(token : Token)
+put(token : Token, time : double, branch : Branch)
+wakeUpBlockedPartner()
#_hideNullTokens(hide : boolean)
ReceiverComparator
+ReceiverComparator(keeper : TimeKeeper)
+compare(first : Object, second : Object) : int

FIGURE 7.5. Key classes for managing time locally.

When a DDEDirector is initialized, it instantiates a DDEThread for each actor that the director manages. DDEThread is derived from ProcessThread. The ProcessThread class provides functionality that is common to all of the process domains (e.g., CSP, DDE and PN). The directors of all process domains (including DDE) assign a single actor to each ProcessThread. ProcessThreads take responsibility of their assigned actor's execution by invoking the iteration methods of the actor. The iteration methods are prefire(), fire() and postfire(); ProcessThreads also invoke wrapup() on the actors they control.

DDEThread extends the functionality of ProcessThread. Upon instantiation, a DDEThread creates a TimeKeeper object and assigns this object to the actor that it controls. The TimeKeeper gets access to each of the DDEReceivers that the actor contains. Each of the receivers can access the TimeKeeper and through the TimeKeeper the receivers can then determine their relative receiver times. With this information, the receivers are fully equipped to apply the appropriate blocking rules as they get and put time stamped tokens.

DDEReceivers use a dynamic approach to accessing the DDEThread and TimeKeeper. To ensure domain polymorphism, actors (DDE or otherwise) do not have static references to the TimeKeeper and DDEThread that they are controlled by. To ensure simplified mutability support, DDEReceivers do not have a static reference to TimeKeepers. Access to the local time management facilities is accomplished via the Java Thread.currentThread() method. Using this method, a DDEReceiver dynamically accesses the thread responsible for invoking it. Presumably the calling thread is a DDEThread and appropriate steps are taken if it is not. Once the DDEThread is accessed, the corresponding TimeKeeper can be accessed as well. The DDE domain uses this approach extensively in DDEReceiver.put(Token) and DDEReceiver.get().

7.4.2 Detecting Deadlock

The other kernel classes of the DDE domain are shown in figure 7.6. The purpose of the DDEDirector is to detect and (if possible) resolve timed and/or non-timed deadlock of the model it controls. Whenever a receiver blocks, it informs the director. The director keeps track of the number of active processes, and the number of processes that are either blocked on a read or write. Artificial deadlocks are resolved by increasing the queue capacity of write-blocked receivers.

Note the distinction between internal and external read blocks in DDEDirector's package friendly methods. The current release of DDE assumes that actors that execute according to a DDE model of computation are atomic rather than composite. In a future Ptolemy II release, composite actors will be facilitated in the DDE domain. At that time, it will be important to distinguish internal and external read blocks. Until then, only internal read blocks are in use.

7.4.3 Ending Execution

Execution of a model ends if either an unresolvable deadlock occurs, the director's completion time is exceeded by all of the actors it manages, or early termination is requested (e.g., by a user interface button). The director's completion time is set via the public *stopTime* parameter of DDEDirector. The completion time is passed on to each DDEReceiver. If a receiver's receiver time exceeds the completion time, then the receiver becomes inactive. If all receivers of an actor become inactive and the actor is not a source actor, then the actor will end execution and its wrapup() method will be called. In such a scenario, the actor is said to have terminated *normally.*

Early terminations and unresolvable deadlocks share a common mechanism for ending execution. Each DDEReceiver has a boolean `_terminate` flag. If the flag is set to true, then the receiver will throw a `TerminateProcessException` the next time any of its methods are invoked. TerminateProcessExceptions are part of the `ptolemy/actor/process` package and ProcessThreads know to end an actor's execution if this exception is caught. In the case of unresolvable deadlock, the `_terminate` flag of all blocked receivers is set to true. The receivers are then awakened from blocking and they each throw the exception.

7.5 Example DDE Applications

To illustrate distributed discrete event execution, we have developed an applet that features a feedback topology and incorporates polymorphic as well as DDE specific actors. The model, shown in figure 7.7, consists of a single source actor (ptolemy/actor/lib/Clock) and an upper and lower branch of four actors each. The upper and lower branches have identical topologies and are fed an identical stream of tokens from the Clock source with the exception that in the lower branch ZenoDelay replaces FeedBackDelay.

As with all feedback topologies in DDE (and DE) models, a positive time delay is necessary in feedback loops to prevent deadlock. If the time delay of a given loop is lower bounded by zero but can not be guaranteed to be greater than a fixed positive value, then a Zeno condition can occur in which time will not advance beyond a certain point even though the actors of the feedback loop continue to execute without deadlocking. ZenoDelay extends FeedBackDelay and is designed so that a Zeno condition will be encountered. When execution of the model begins, both FeedBackDelay and ZenoDelay

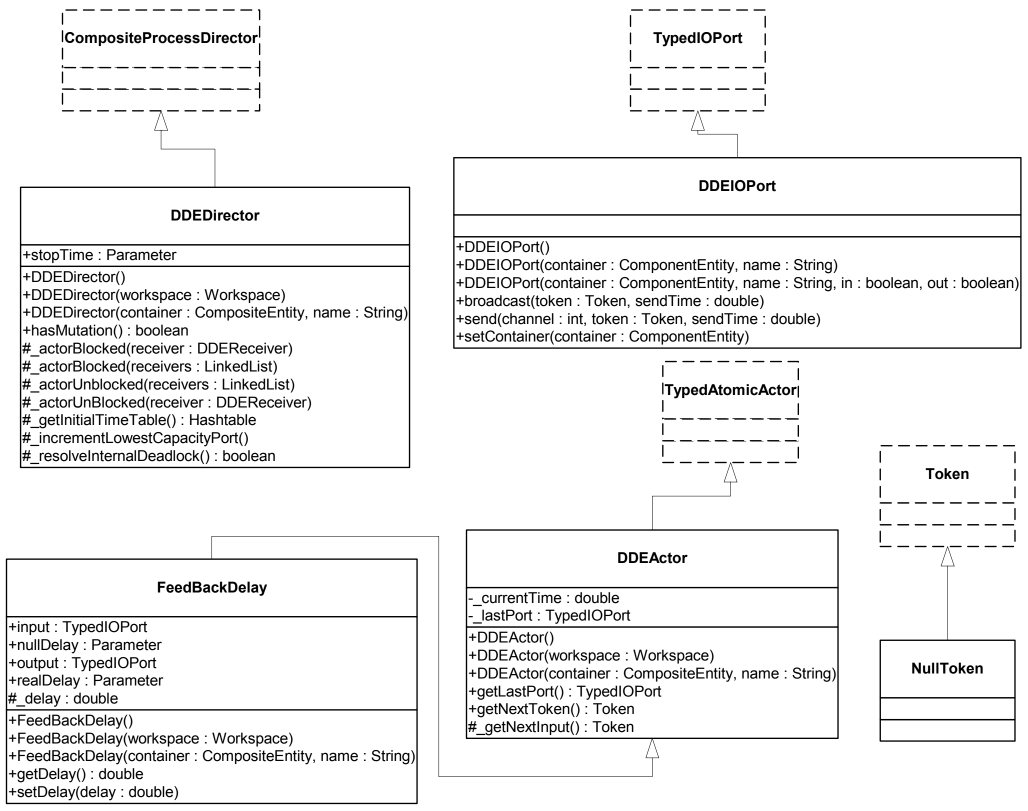
CompositeProcessDirector
TypedIOPort
DDEDirector
+stopTime : Parameter
+DDEDirector()
+DDEDirector(workspace : Workspace)
+DDEDirector(container : CompositeEntity, name : String)
+hasMutation() : boolean
#_actorBlocked(receiver : DDEReceiver)
#_actorBlocked(receivers : LinkedList)
#_actorUnblocked(receivers : LinkedList)
#_actorUnBlocked(receiver : DDEReceiver)
#_getInitialTimeTable() : Hashtable
#_incrementLowestCapacityPort()
#_resolveInternalDeadlock() : boolean
DDEIOPort
+DDEIOPort()
+DDEIOPort(container : ComponentEntity, name : String)
+DDEIOPort(container : ComponentEntity, name : String, in : boolean, out : boolean)
+broadcast(token : Token, sendTime : double)
+send(channel : int, token : Token, sendTime : double)
+setContainer(container : ComponentEntity)
TypedAtomicActor
Token
FeedBackDelay
+input : TypedIOPort
+nullDelay : Parameter
+output : TypedIOPort
+realDelay : Parameter
#_delay : double
+FeedBackDelay()
+FeedBackDelay(workspace : Workspace)
+FeedBackDelay(container : CompositeEntity, name : String)
+getDelay() : double
+setDelay(delay : double)
DDEActor
-_currentTime : double
-_lastPort : TypedIOPort
+DDEActor()
+DDEActor(workspace : Workspace)
+DDEActor(container : CompositeEntity, name : String)
+getLastPort() : TypedIOPort
+getNextToken() : Token
#_getNextInput() : Token
NullToken

FIGURE 7.6. Additional classes in the DDE kernel.

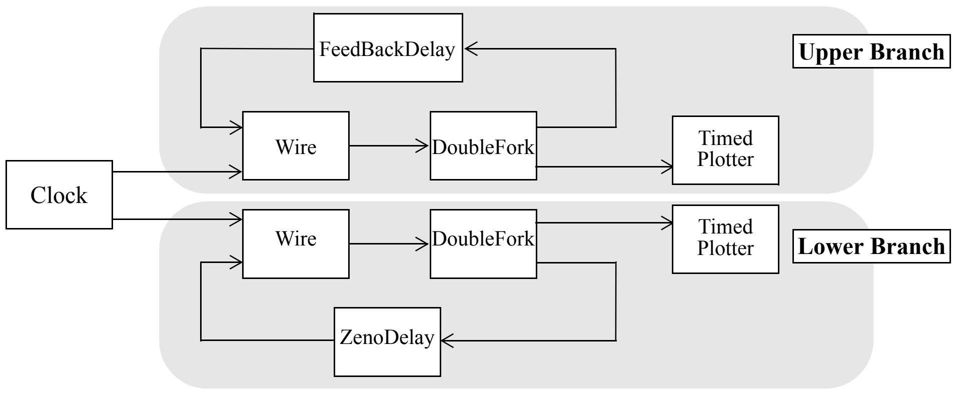

FIGURE 7.7. Localized Zeno condition topology.

are used to feed back null tokens into Wire so that the model does not deadlock. After local time exceeds a preset value, ZenoDelay reduces its delay so that the lower branch approximates a Zeno condition.

In centralized discrete event systems, Zeno conditions prevent progress in the entire model. This is true because the feedback cycle experiencing the Zeno condition prevents time from advancing in the entire model. In contrast, distributed discrete event systems localize Zeno conditions as much as is possible based on the topology of the system. Thus, a Zeno condition can exist in the lower branch and the upper branch will continue its execution unimpeded. Localizing Zeno conditions can be useful in large scale modeling in which a Zeno condition may not be discovered until a great deal of time has been invested in execution of the model. In such situations, partial data collection may proceed prior to correction of the delay error that resulted in the Zeno condition.