ThreadedComposite: A Mechanism for Building Concurrent and Parallel Ptolemy II Models

Edward A. Lee

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-151
http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-151.html

December 7, 2008

Acknowledgement

This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET) and #0720841 (CSR-CPS)), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312), the Air Force Research Lab (AFRL), the State of California Micro Program, and the following companies: Agilent, Bosch, HSBC, Lockheed-Martin, National Instruments, and Toyota.

ThreadedComposite: A Mechanism for Building Concurrent and Parallel Ptolemy II Models *

Edward A. Lee
UC Berkeley
eal@eecs.berkeley.edu

December 7, 2008

Abstract

This paper describes the usage patterns of the ThreadedComposite actor in Ptolemy II. This actor enables the execution of opaque actors (atomic actors or composite actors with directors) in a separate thread, thus providing multithreading for models of computation that are not already multthreaded. It can be used to execute an actor in the background, to execute multiple actors in parallel (e.g. on a multicore machine), and to execute actors that block on I/O operations without blocking other actors.

1 Introduction

A program is said to be **concurrent** if different parts of the program *conceptually* execute simultaneously. A program is said to be **parallel** if different parts of the program *physically* execute simultaneously on distinct hardware (such as on multicore machines or on server farms). A parallel program must be concurrent, but a concurrent program need not be executed in parallel.

Ptolemy II has long included a **Kahn process networks** (PN) model of computation [16] and **rendevous** models of computation [1, 9]. Like many MoCs in Ptolemy II, these are intrinsically concurrent, but unlike most, they are implemented using multiple threads. On machines that execute threads on multiple processors, such as SMP or multicore machines, these MoCs provide a mechanism for constructing parallel programs.

*This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET) and #0720841 (CSR-CPS)), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312), the Air Force Research Lab (AFRL), the State of California Micro Program, and the following companies: Agilent, Bosch, HSBC, Lockheed-Martin, National Instruments, and Toyota.

This paper introduces a new Ptolemy II actor called ThreadedComposite that supports multithreaded computation with MoCs that are not implemented using threads, such as **discrete events** (DE), various **dataflow** MoCs, and **synchronous/reactive** (SR).

Programs are typically executed in parallel to improve performance. Hence, one may develop a concurrent program in order to be able to exploit parallel hardware. However, there are other reasons for developing concurrent programs. One reason is I/O. Portions of a program may stall to wait for inputs from the environment, for example, or to wait for the right time produce some output. Another reason is to create the illusion of parallel hardware. Time-sharing operating systems, for example, were originally created to give each user the illusion of having their own machine. Of course, these two reasons are intertwined, since interaction with such users is about I/O and its timing.

Techniques for developing concurrent programs divide into two families, **message passing** and **threads** [10]. Threads are sequential procedures that share memory. Java directly supports threads. See Lea [11] for an excellent "how to" guide to using threads in Java. Recently, Java acquired an extensive library of concurrent data structures and mechanisms based on threads [12]. In the message passing style, rather than sharing memory, sequential procedures invoke library mechanisms for sending messages to one another. A popular such library is MPI [19].

A program is said to be *determinate* if given the same inputs, it produces the same outputs on every execution. Sometimes programmers deliberately construct nondeterminate programs. Most of the time, however, even with concurrent programs, programmers want repeatable behavior. This is particularly true if one considers the timing of the inputs as part of the input. Given the same inputs arriving at the same times, the outputs of the program should be the same.

It is notoriously difficult to maintain determinacy with threads [20, 15, 8]. The essence of the problem is that multithreaded programs have vast numbers of possible interleavings, and distinct interleavings can result in observably different behavior. Unfortunately, maintaining determinacy with message-passing libraries like MPI is also difficult. The richness of the library makes it too easy for programmers to inadvertently introduce nondeterminate mechanisms.

There has been considerable debate in the literature between message passing schemes (or their variations referred to as event-based mechanisms) and threads. Some argue that the message passing schemes are a bad idea [2]. Some argue the contrary [21, 24]. In this paper, we will show a very practical and easy to use event-based scheme. We use threads as part of the underlying implementation, but threads are not part of the programmer's model. The reader can then form a judgement about this debate.

Ptolemy II includes a number of directors that have a message-passing style, but are far more disciplined than message-passing libraries like MPI. For example, the DE director implements a discrete-even model of computation [23, 5, 3], where sequential procedures (actors written in Java) communicate via time-stamped messages. The job of the director is to ensure that each actor reacts to input events in time-stamp order. DE models in Ptolemy II are determinate [22, 13, 18] and concurrent, but the DEDirector realizes this concurrency in a very limited way. It fires actors one at time, allowing each firing to complete before invoking the next one. Thus, although the model is

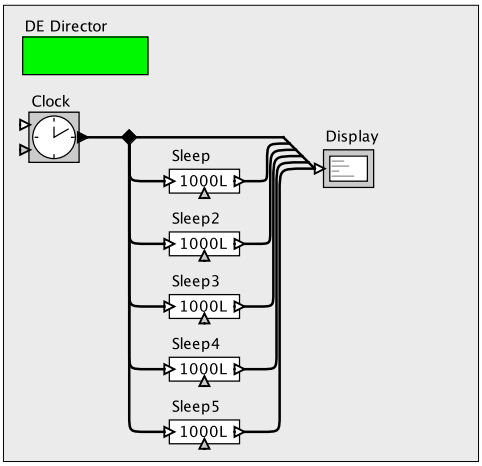

Figure 1: A discrete-event model that takes five seconds to process each Clock event.

concurrent, it will not effectively exploit parallel hardware, nor can it effectively handle actors whose fire methods block on I/O or block to achieve some real-time behavior. The firing of such actors will block other actors.

Parallel execution of DE models is a notoriously difficult problem [6, 25]. The ThreadedComposite actor introduced here stops short of providing fully parallel execution of DE. Instead, it provides a simple mechanism for controlling concurrency where it is needed. If a particular actor blocks on I/O, for example, the ThreadedComposite actor provides a way to prevent that actor from blocking other computations. It also provides a way to exploit multicore architectures to get faster execution of a model. All of this is accomplished while maintaining the determinate semantics of DE.

This paper assumes familiarity with the actor-oriented models of Ptolemy II and actor semantics [14, 4].

2 Concurrent Execution

We begin with a simple example showing concurrent execution of an actor that blocks on I/O. To emulate this, we use the Sleep actor, which in its fire() method calls the Java Thread.sleep() method. The thread that calls fire() will stall for an amount of time given by the *sleepTime* parameter of the Sleep actor (this parameter has type long and specifies the number of milliseconds to sleep).

Consider the model in figure 1, where each Sleep actor has been assigned a *sleepTime* value of 1000L, specifying a sleep time of one second. This is a discrete-event (DE) model. The Clock actor generates events with time-stamps 0, 1, 2, etc. This triggers execution of five Sleep actors. Since each Sleep actor sleeps for one second, the total execution time per Clock event is five seconds (or slightly more, since the Clock and Display actors also take time to execute).

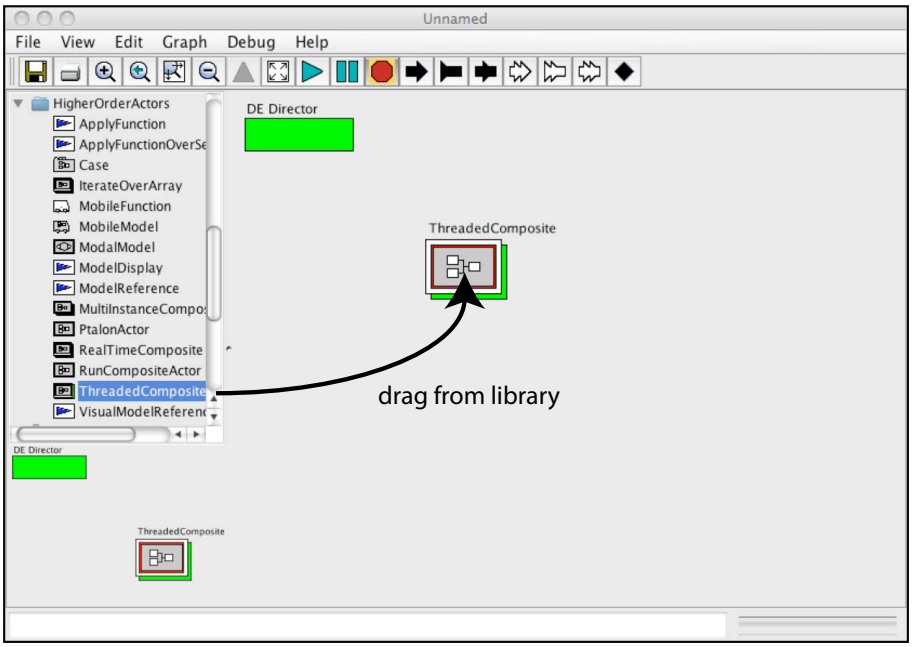

Figure 2: The ThreadedComposite actor dragged in from the library.

The Sleep actors, however, are not doing anything useful during their sleep time. They are blocked on I/O. Indeed, the way that Java.sleep() works is that, via the operating system, it sets up a timer interrupt that will reawaken the current thread one second later, and then suspends execution of the thread. The operating system is able to meanwhile execute other threads, but there are no other threads associated with this model, so the entire model is blocked while each Sleep actor fires.

If instead we put the Sleep actors inside instances of ThreadedComposite, then the fire() method of the Sleep actor will be called in a separate thread from that of the DE director. We refer to the **inside thread** and the **director thread** to distinguish these two threads.

To use the ThreadedComposite actor, drag it into a model from the HigherOrderActors library, as shown in figure 2. ThreadedComposite is a **higher-order actor** in that it is an actor parameterized by another actor [16]. We will first illustrate the use of this actor with a single atomic actor inside. In particular, you can find the Sleep actor in the RealTime library and drag it onto the instance of ThreadedComposite that you just created, as shown in figure 3. Notice that the ThreadedComposite acquires the ports and icon of the actor that we just dragged onto it. It also acquires the parameters of that actor, as you can verify by double clicking on it.

The model in figure 4 is similar to the one above with five instances of Sleep. Its behavior as a DE model is nearly identical, but the five Sleep actors fire concurrently in separate threads. When those threads block, the DE director thread is not blocked, so all

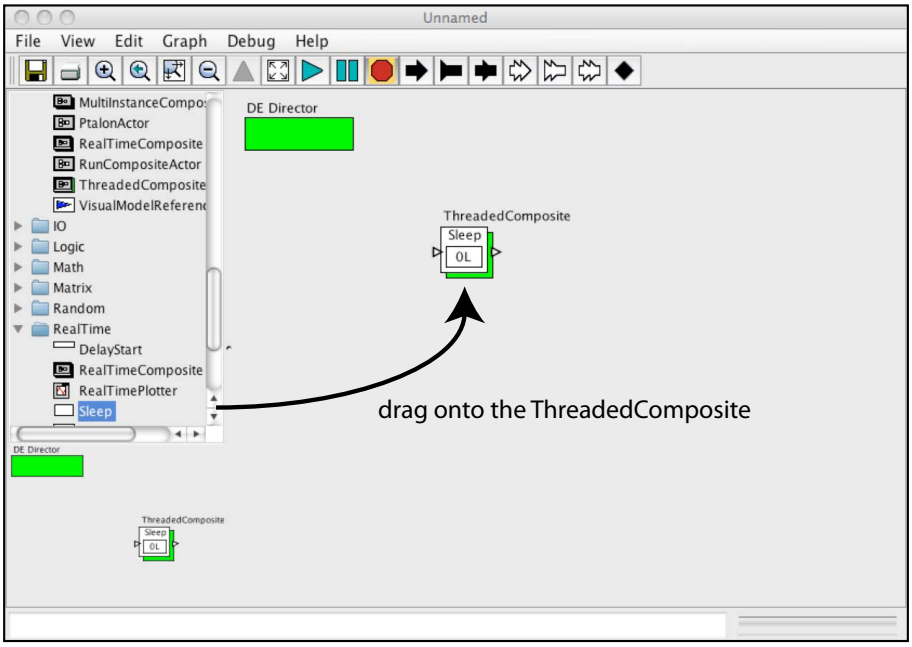

Figure 3: The ThreadedComposite acquires the icon and parameters of actors you drag into it.

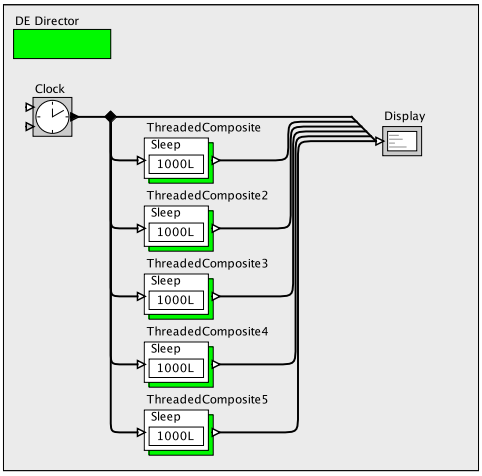

Figure 4: A model similar to the one in figure 1, but using instances of ThreadedComposite to fire the five Sleep actors concurrently.

five actors sleep simultaneously. The execution time now is approximately one second per Clock event rather than five seconds per Clock event. As a DE model, however, the behavior is almost identical. The same sequence of time-stamped events is delivered to the Display actor. The (slight) difference in semantics is that the ThreadedComposite actor behaves like a TimedDelay actor, as discussed next.

3 ThreadedComposite and Model Time Delay

The Sleep actor in DE behaves semantically like a wire, albeit a slow one. The output event is identical to the input event. It has the same value and time stamp. If you replace each Sleep actor in figure 1 with a wire straight through, the behavior will be identical except that the execution will be much faster. When you put the Sleep actor inside of an instance of ThreadedComposite, the effect is almost, but not quite the same.

To achieve concurrency, ThreadedComposite introduces **model time delay** from its inputs to its outputs. This delay enables the actor to produce its outputs in a subsequent firing, after the firing where it consumes the inputs that trigger those outputs. This model time delay is not the same as execution time delay (or real time delay). In our example, the model time delay is zero (or more precisely, one microstep, as explained below), whereas the computation time is a full second.

In particular, the ThreadedComposite has a parameter *delay* that specifies the model time delay from an input to the output. If an input event with time stamp τ triggers a firing of the inside actor, and that inside actor produces an output on that firing, then the output will have time stamp $\tau + delay$. Hence, a ThreadedComposite with Sleep actor inside is semantically equivalent to a cascade of a Sleep and TimedDelay, which is semantically equivalent to a TimedDelay alone.

By default, the value of *delay* is zero, which means that the time stamp of the output will equal that of the input. But there is a **microstep** delay that you can think of as an infinitesimal delay.

Specifically, the discrete-event domain in Ptolemy II (as with some other timed domains like Continuous) has a **super dense** model of time [17]. This means that a signal from one actor to another can contain multiple events with the same time stamp. These events are "simultaneous," but nonetheless have a well-defined sequential ordering determined by the order in which they are produced.

If *delay* is 0.0, then the fire() method of the ThreadedComposite actor produces on its output port the output result from the *previous iteration* of the inside actor with the same time stamp, if there was one. If there wasn't such a previous iteration, then it produces no output, effectively asserting that the output is absent. The postfire() method consumes and records any inputs for use by the next firing of the inside actor. If there are such inputs, it also requests a refiring at the current time. This refire request triggers the next iteration (at the same time stamp), on which the output is produced.

For the example in figure 4, the execution proceeds as follows. The Clock is the first actor to fire, producing an event with time stamp 0. This event enables the Display actor and all five instances of ThreadedComposite. These can fire in any order because the instances of ThreadedComposite will not produce any output in that firing. Instead, when their postfire() method is invoked, they will record the input in a queue for use by

the inside thread, which fires the inside actor. They will also request a refiring at time 0.0 (the current time). When this refiring occurs, the ThreadedComposite will stall the director thread until the inside thread has completed the firing of the inside actor.

All instances of ThreadedComposite will be postfired before any of these requested refirings occur. Thus, all five inside threads will be presented with inputs before the director thread stalls waiting for results from any one of those threads. This is how concurrent execution of all five actors is achieved.

For the model in figure 4, if the *delay* parameter of each ThreadedComposite is set to 0.0, then the only observable difference in behavior from that of figure 1 (aside from execution time) is that the Display actor will fire twice instead of once for each Clock event. On the first of these firings, it will have only one event on the topmost input, which comes directly from the Clock. The other input channels will all be absent. On the second firing, which occurs with the same time stamp, the topmost input channel will be absent and the other five channels will have events from the ThreadedComposite. This difference is observable on the displayed output if the *suppressBlankLines* parameter of the Display actor set to false, because in this case the Display actor will produce a blank line for each absent event.

4 Background Execution

In the previous example, ThreadedComposite is used to dispatch several blocking I/O operations simultaneously. Another use for the ThreadedComposite actor is to execute an actor or a submodel in the background while a model continues to execute for some time. Typically this means that the *delay* parameter of the ThreadedComposite will be given a value greater than zero. If the value is d, then this effectively gives a DE director (or any other timed director) permission to continue to execute the model up to model time $\tau + d$ after delivering an event with time stamp τ to the ThreadedComposite. In effect, the ThreadedComposite guarantees that it will not produce an event with a time stamp smaller than $\tau + d$. In fact, any output it produces in response to the input event at time τ will have time stamp $\tau + d$.

Consider the example in figure 5. In that model, the ThreadedComposite has delay 2.0 and contains a Sleep actor with a sleep time of 1.5 seconds. Executing this model produces exactly the same result as if the ThreadedComposite were replaced by a TimedDelay with delay 2.0.

The ThreadedComposite takes a full 1.5 seconds to execute, but that execution occurs in the background and does not block firings of the other actors, except as necessary to preserve timed DE semantics. Thus, the lower actors react to events at times 0, 0.25, 1, and 1.25 before the ThreadedComposite has completed its execution. The ThreadedComposite, in effect, executes the Sleep actor in the background without compromising the determinacy of DE semantics.

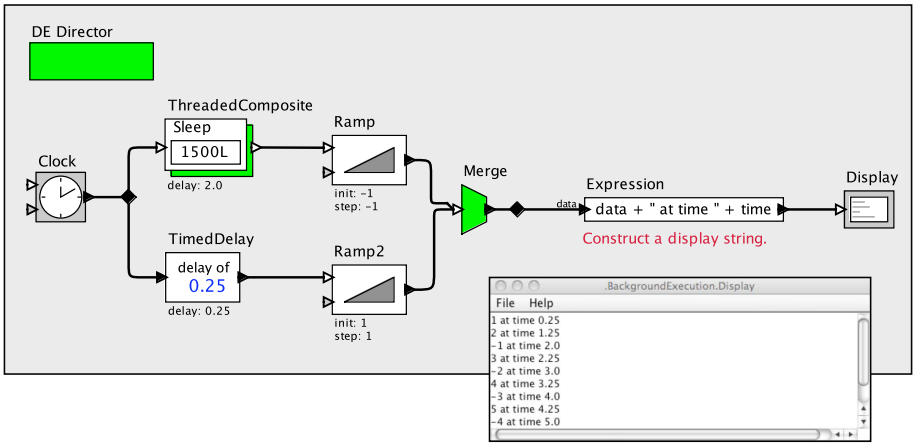

Figure 5: This model uses ThreadedComposite with a non-zero delay to allow the director to process events with future time stamps while the ThreadedComposite reacts in the background to a given event.

```
public void fire() throws IllegalActionException {
  // Record the start time.
  long start = System.currentTimeMillis();

  // Read and discard the input, if there is one.
  if (input.hasToken(0)) {input.get(0); }

  // Perform a fixed (useless) computation.
  int dummy = 0;
  for (long i = 0; i < count; i++) {
    dummy++;
  }
  // Produce on the output the actual time consumed.
  Token result = new LongToken(System.currentTimeMillis() - start);
  output.send(0, result);
}
```

Figure 6: The fire() method of an actor that consumes cycles on the processor to emulate a computational load. This is a simplified version of the Ptolemy II ExecutionTime actor.

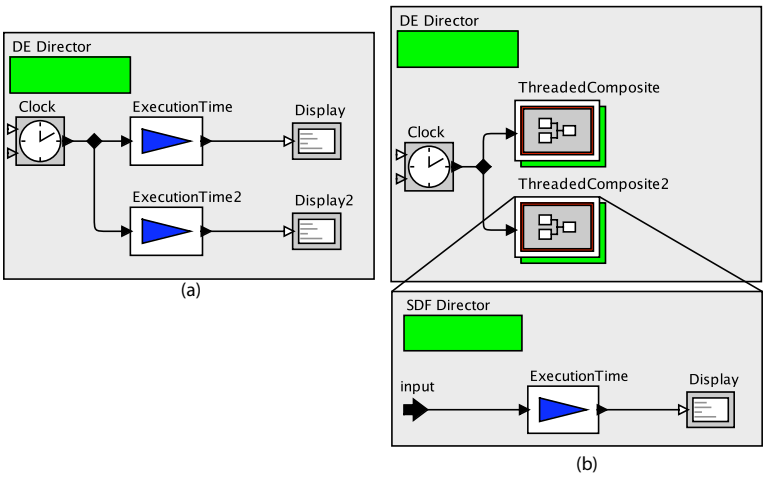

Figure 7: Two models that use the ExecutionTime actor. The model on the left executes the two actors in the same thread, and thus does not exploit a multicore machine. The one on the right has the ExecutionTime actors inside instances of ThreadedComposite. On a two-core machine, it executes approximately twice as fast as the model on the left.

5 Parallel Execution

ThreadedComposite can also be used to achieve parallel execution on multicore machines. This is useful if you have actors that consume a large number of CPU cycles. The ExecutionTime actor can be used to illustrate this effect. Its fire() method (simplified) is shown in figure 6. Like the Sleep actor, this actor's role is to consume time, but unlike the Sleep actor, it uses CPU cycles doing a (useless) fixed computation during that time.

In figure 7, we show two models using the ExecutionTime actor. The model on the left fires these actors directly in the director thread. The one on the right wraps them in instances of ThreadedComposite, so their execution occurs in separate threads. On a dual-core MacBook Pro, for example, the model on the right executes approximately twice as fast as the model on the left.

The model on the right in the figure also illustrates that ThreadedComposite can be used with composite actors as well as with atomic actors. Instead of dragging an actor onto the ThreadedComposite as in figure 2, just right click on the instance of ThreadedComposite and select Open Actor. Then populate the actor with a director, ports, and a submodel consisting of any number of other actors. The submodel will be executed in a separate thread. In this example, we have also shown the use of a distinct director, in this case the synchronous dataflow (SDF) director.

Note that the process networks (PN) director, since it executes every actor in a separate thread, can also be used to get the same effect, exploiting multiple cores. The

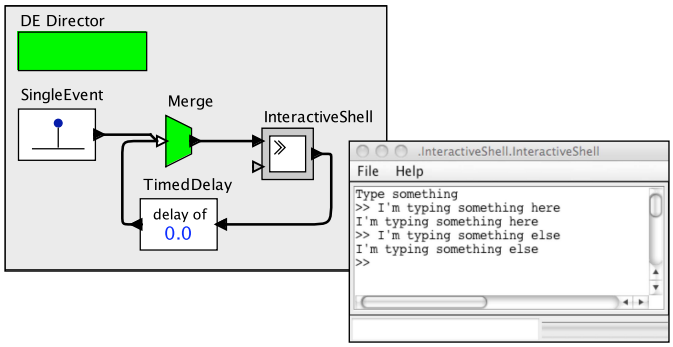

Figure 8: A model that opens a window on the screen into which a user types commands. This model simply echos back what the user typed.

ThreadedComposite actor makes this possible with directors that are not themselves multithreaded, such as the DE director.

6 Sporadic Behaviors

All of the examples we have considered so far use actors with rather simple execution patterns. For each input event, they produce an output event. Moreover, the model time of the output event has a fixed relationship to the input event. Not all applications of ThreadedComposite have such simple behaviors.

Consider the InteractiveShell actor in Ptolemy II. This actor opens a window on the screen into which a user may type commands. When the actor fires, it reads a string from the input port and displays it in the window. It then displays a prompt (which is a string specified by a parameter). The user then types something, and when the user hits the Return or Enter key, the command is sent as a string to the output port, and the fire() method returns. This actor blocks execution of the model until the user enters a command.

A simple model using this actor is shown in figure 8. Here, a SingleEvent actor provides an initial event that starts execution of the feedback loop. Its output is the string "Type something." When the user types something, that string is fed back through the TimedDelay actor and through the Merge actor back to the InteractiveShell, which displays what was typed on the next line.

The TimedDelay actor in this model is needed to break the circular causality that is implied by the feedback loop. In this example, the *delay* parameter of the TimedDelay is set to 0.0, so the time-stamp of the output is the same as the time stamp of the input. However, the output occurs one microstep later in super-dense time, as explained above. Thus, the outputs of the Merge actor, though having the same time stamp, are logically ordered. In this model, time does not advance past 0.0. If you change the *delay* parameter of the TimedDelay actor to something larger, then model time will advance in fixed increments.

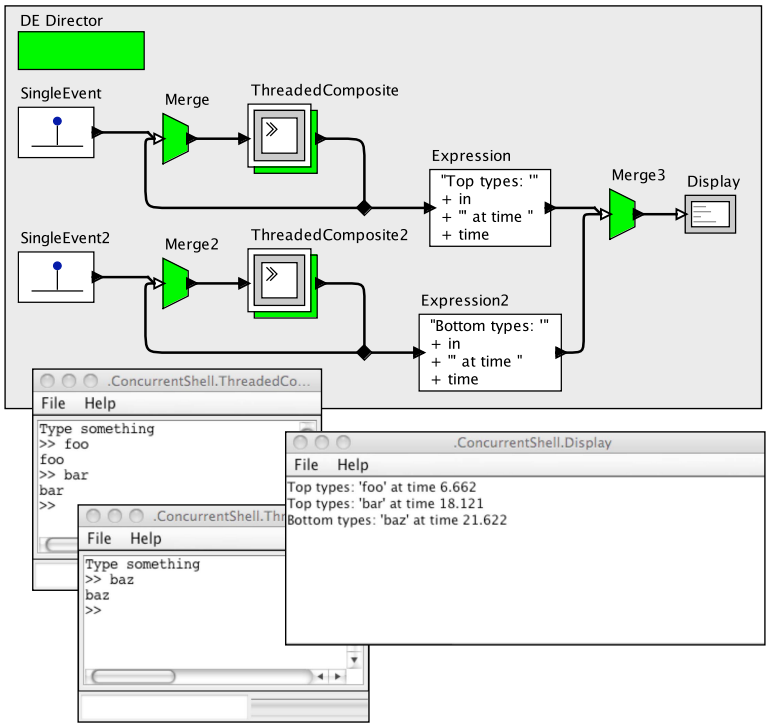

Figure 9: A model that opens two windows on the screen into which users type commands. This model merges what the users type and displays them together in a third window. If the *delay* parameter of the ThreadedComposite actors is set to `UNDEFINED`, then the users can type in any order, and their text will be merged in the order in which they type.

Now suppose that we would like to two instances of InteractiveShell in the same model. Since each instance blocks execution of the model, this would be problematic. The user would have to type into one first, then the other, in whatever order the DE scheduler choses to fire the instances of InteractiveShell. This is probably not what we want.

Consider the model shown in figure 9. Here, we have two instances of InteractiveShell, each wrapped in an instance of ThreadedComposite. With the default parameters, however, this still does not do what we want. In order to allow users to type in arbitrary order, we need to set the *delay* parameter of the instances of ThreadedComposite to `UNDEFINED`.

Recall that with ThreadedComposite, the instances of InteractiveShell execute in separate threads called the inside thread. If *delay* has value `UNDEFINED`, then output events from ThreadedComposite are produced at the current model time when the inside thread happens to produce those events. However, in the model we have given,

there is no mechanism for time to advance, so model time will remain at 0.0. This won't quite work. We could add, for example, a Clock actor to the model, and then time will advance.

More interestingly, we can set the *synchronizeToRealTime* parameter of the ThreadedComposite actors to `true`. This has the effect that setting the time stamp of any outputs from the ThreadedComposite to the greater of real time (measured in seconds from the start of execution) and the model time. Since there is no mechanism for model time to advance, in this example, the time stamps will be the real time.

In addition, for this model to work, we need to set the *stopWhenQueueIsEmpty* parameter of DE Director to `false`. While the model is waiting for users to type something, there are no pending events on the event queue, and by default, the DE Director terminates the execution of a model when this occurs. The *stopWhenQueueIsEmpty* parameter prevents this termination.

The figure shows a sample execution with the two interactive shell windows and the display of the merged results. Note that the users can type in any order, and that the time stamps represent the time at which the user hit the Enter or Return. The Expression actors are used to format the text that is displayed.

This example illustrates the use of ThreadedComposite to generate sporadic events (in this case user inputs) without blocking the model to wait for those events. The same mechanism can be used to inject into a model sensor events, network packet arrivals, or any pushed data.

In summary, to make this work, we have set the parameters of the instances of ThreadedComposite as follows:

delay: `UNDEFINED`

synchronizeToRealTime: `true`

and the DE Director as follows:

stopWhenQueueIsEmpty: `false`

7 Real-Time Behaviors

We have seen that in ThreadedComposite, if *synchronizeToRealTime* is true, then output events are assigned a time stamp that is the larger of the current model time and the current real time (in seconds since the start of execution). In addition, when a time-stamped input event is delivered to an input of the ThreadedComposite, if the time stamp value is larger than real time, then the inside thread will stall until real time matches or exceeds the value of the time stamp (again measured in seconds since the start of execution of the inside thread). Thus, ThreadedComposite can be used to delay execution of actor until some real time.

In the example of figure 9, the input events of the InteractiveShell actors always have time stamps less than real time because they are generated either at the start by the SingleEvent actor (with time stamp 0.0) or by the outputs fed back from the InteractiveShell, which assigns time stamps that match real time. Hence, in that example, the stall of the inside thread will not occur.

If we modify the model, however, putting a TimedDelay actor in each feedback loop with the *delay* parameter set to, say, 10.0, then users will be constrained to enter

data no more than once every 10 seconds.

Note that the DE Director also has a *synchronizeToRealTime* parameter. That parameter delays execution of *all* actors until real time matches the time stamp of their inputs. Using ThreadedComposite, we can build models with more targeted real-time properties.

8 Using ThreadedComposite with Other Directors

All of the examples above use ThreadedComposite with the DE director. This is natural because of the *delay* parameter and the correspondence with the TimedDelay actor. With some care, however, ThreadedComposite can also be used with some other directors.

First, note that it would make no sense to use ThreadedComposite with PN or Rendezvous, since these directors already execute all actors in their own threads.

The synchronous/reactive (SR) director can be used with ThreadedComposite, with one constraint. The *delay* parameter of ThreadedComposite must match the *period* parameter of the SR director (or it can an exact multiple of the *period*). By default, both parameters have value 0.0, so the default values match. Note that with this default, any outputs produced by the ThreadedComposite will appear one clock tick later than the inputs that trigger them, so the ThreadedComposite behaves like a Pre actor. If the *period* parameter is 1.0, say, and the *delay* is 2.0, then outputs will appear two ticks later than the input that triggered them. This mechanism can be used to achieve **pipeline parallelism** in SR. If the *delay* parameter of ThreadedComposite is given the value `UNDEFINED`, then the output will appear a nondeterminate number of ticks later.

The synchronous dataflow (SDF) director can sometimes be used with ThreadedComposite, but the combination is a bit odd. To get multithreaded execution with SDF, it is usually better to use a PN director instead. Nonetheless, it is sometimes possible to use ThreadedComposite with SDF, and it can be occasionally be useful because PN does not compose well with other domains [7].

First, be aware that, just as with SR, the first firing of ThreadedComposite will not produce any output. This violates the SDF assumption that the number of tokens produced and consumed on every port is constant throughout the execution of the model. Nonetheless, as long as downstream actors are robust enough to not throw exceptions if they are fired with no inputs, models can be made to work. Typically, actors check for input availability in prefire(), returning false if their required inputs are not present. If all downstream actors do that, then the model will execute without exception. Because the first firing produces no output, a more sensible dataflow director to use with ThreadedComposite is dynamic dataflow (DDF).

As with SR, using ThreadedComposite with SDF requires that the *delay* parameter match the *period* parameter of the SDF director (or be an exact multiple). Again, this can be used to achieve pipeline parallelism in SDF. As with SR, if *period* is `UNDEFINED`, then outputs will appear a nondeterminate number of iterations later than the inputs that trigger them.

As of this writing, the DDF director has no *period* parameter, and hence will not increment time. Thus, for ThreadedComposite to work with DDF, the *delay* parameter

must remain at 0.0 (or be *UNDEFINED*, if the nondeterminism is tolerable), unless DDF is being used inside some domain that itself increments time (such as DE).

ThreadedComposite can be used with CT or Continuous directors, just as with DE.

9 Port-Parameters

One last subtlety of the ThreadedComposite actor is that it cannot expose instances of ParameterPort without introducing nondeterminacy in the execution. A ParameterPort is an input port that sets the value of a parameter with the same name. Upon receiving a token at such a port, if the ThreadedComposite were to set a parameter visible by the inside thread, there is no assurance that the inside thread is not still executing an earlier iteration. Thus, it could appear to be sending a message backward in time, which would be bizarre. To prevent this error, the ThreadedComposite actor does not mirror such ports, and hence they appear on the outside only as parameters.

10 How it Works

The ThreadedComposite Actor is a container for another actor that executes that other actor in a separate thread (the inside thread). This actor starts that thread in its initialize() method, which is invoked by its executive director (the director in charge of firing the ThreadedComposite). The thread that invokes the action methods of the ThreadedComposite (initialize(), prefire(), fire(), postfire(), and wrapup()) is called the director thread.

ThreadedComposite is a subclass of MirrorComposite, which automatically creates input and output ports to match those of the inside actor. Input events provided at those input ports are provided as input events to the contained actor. Outputs provided by the contained actor become output events of this actor. If used in a timed domain, the time stamp of the output events depends on the *delay* parameter, as explained above.

The inside thread blocks waiting for inputs or pure events (firings where all inputs are absent). Inputs are queued for use by the inside thread when the postfire() method of the ThreadedComposite is invoked by the director thread. Pure events are provided after fireAt(), fireAtCurrentTime(), or fireAtFirstValidTimeAfter() are called by either the inside thread or the director thread. When the time of those firing requests becomes current time, the container will (presumably) fire the ThreadedComposite actor, and this actor will provide a pure event to the inside thread, causing it to fire the contained actor.

When the inside thread completes an iteration (prefire(), fire(), postfire()) of the inside actor, any outputs that are produced by that iteration are collected and queued for use by the director thread. When the director thread fires the ThreadedComposite actor and current model time matches the time at which those outputs should be produced, the director thread retrieves these outputs from the queue and sends them via the output ports of the ThreadedComposite. If the inside thread hasn't completed the appointed iteration, then the director thread stalls until it has. If there are no output ports, or if the inside actor doesn't happen to produce any tokens on its output ports, the director

thread stalls anyway to wait for the completion of the iteration. This prevents model time in the director thread from getting ahead of model time seen by the inside thread by more than the value of *delay*.

When the wrapup() method of a ThreadedComposite is called, the inside thread is provided with signal to terminate rather than to process additional inputs. The inside thread will also exit if stop() is called on the ThreadedComposite, as occurs for example if the user pushes the Stop button in the GUI; however, in this case, which iterations are completed is nondeterminate (there may be inputs left unprocessed).

11 Conclusion

The ThreadedComposite actor provides a versatile mechanism for executing components of a model concurrently and/or in parallel with execution of other components. This mechanism can exploit multicore architecture to get faster executions, or it can be used to contain the extent to which I/O operations block execution of the model. All of this done while preserving the determinate semantics of discrete-event models.

References

[1] F. Arbab. Reo: A channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[2] R. v. Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *10th Workshop on Hot Topics in Operating Systems (HotOS IX)*, Lihue, Hawaii, 2003.

[3] C. G. Cassandras. *Discrete Event Systems, Modeling and Performance Analysis*. Irwin, 1993.

[4] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(2):127–144, 2003.

[5] G. S. Fishman. *Discrete-Event Simulation: Modeling, Programming, and Analysis*. Springer-Verlag, 2001.

[6] R. Fujimoto. *Parallel and Distributed Simulation Systems*. John Wiley and Sons, 2000.

[7] A. Goderis, C. Brooks, l. Altintas, and E. A. Lee. Composing different models of computation in Ptolemy II and Kepler. In *International Conference on Computational Science (ICCS)*, to appear, 2007.

[8] B. Hayes. Computing in a parallel universe. *American Scientist*, 95:476–480, 2007.

[9] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8), 1978.

[10] H. C. Lauer and R. M. Needham. On the duality of operating system structures. *SIGOPS Operating Systems Review*, 13(2):3–19, 1979.

[11] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, Reading MA, 1997.

[12] D. Lea. The java.util.concurrent synchronizer framework. *Science of Computer Programming*, 58(3):293–309, 2005.

[13] E. A. Lee. Modeling concurrent real-time processes using discrete events. *Annals of Software Engineering*, 7:25–45, 1999.

[14] E. A. Lee. Model-driven development - from object-oriented design to actor-oriented design. In *Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation (a.k.a. The Monterey Workshop)*, Chicago, September 24 2003.

[15] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[16] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[17] E. A. Lee and H. Zheng. Operational semantics of hybrid systems. In M. Morari and L. Thiele, editors, *Hybrid Systems: Computation and Control (HSCC)*, volume LNCS 3414, pages pp. 25–53, Zurich, Switzerland, March 9-11 2005. Springer-Verlag.

[18] E. A. Lee and H. Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *EMSOFT*, Salzburg, Austria, October 2007. ACM.

[19] Message Passing Interface Forum. MPI2: A message passing interface standard. *International Journal of High Performance Computing Applications*, 12(1-2):1–299, 1998.

[20] H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.

[21] R. van Renesse. Goal-oriented programming, or composition using events, or threads considered harmful. In *Eighth ACM SIGOPS European Workshop*, September 1998.

[22] R. K. Yates. Networks of real-time processes. In E. Best, editor, *Proc. of the 4th Int. Conf. on Concurrency Theory (CONCUR)*, volume LNCS 715. Springer-Verlag, 1993.

[23] B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation*. Academic Press, 2nd edition, 2000.

[24] N. Zeldovich, A. Yip, F. Dabek, R. T. Morris, D. Mazieres, and F. Kaashoek. Multiprocessor support for event-driven programs. In *USENIX Annual Technical Conference*, San Antonio, Texas, USA, June 9-14 2003.

[25] Y. Zhao, E. A. Lee, and J. Liu. A programming model for time-synchronized distributed real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Bellevue, WA, USA, April 3-6 2007. IEEE.