PTIDES: A Programming Model for Distributed Real-Time Embedded Systems

Patricia Derler
Thomas Huining Feng
Edward A. Lee
Slobodan Matic
Hiren D. Patel
Yang Zhao
Jia Zou

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-72
http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-72.html

May 28, 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), #0647591 (CSR-SGER), 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.

PTIDES: A Programming Model for Distributed Real-Time Embedded Systems *

Patricia Derler
University of Salzburg

Thomas Huining Feng
UC Berkeley

Edward A. Lee
UC Berkeley

Slobodan Matic
UC Berkeley

Hiren D. Patel
UC Berkeley

Yang Zhao
Google

Jia Zou
UC Berkeley

ABSTRACT

We describe a programming model called *PTIDES* (Programming Temporally Integrated Distributed Embedded Systems), that extends the discrete-event model of computation with a carefully chosen relationship between real time and model time. PTIDES provides a framework for exploring a family of execution strategies for distributed embedded systems. Our objective in this paper is to present an execution strategy that 1) allows independent events to be processed out of time stamp order, 2) uses clock synchronization as a replacement for null message communication across distributed platforms, 3) defines a notion of when events are *safe to process* and 4) presents an implementation of a PTIDES model. This work puts forward an execution strategy that is aggressive in concurrent execution of events.

1. INTRODUCTION

Current programming practices for distributed real-time embedded systems often employ commercial-off-the-shelf real-time operating systems (RTOSs) and real-time object request brokers as utilities for implementing the system. Programmers also use languages such as C with concurrency expressed by threads. RTOSs and threads, however, provide only weak guarantees that the system will meet real-time constraints. Nor do they guarantee that the behavior of the system is deterministic. A consequence is that the only way to achieve confidence in the implementation is through extensive testing. Such testing validates that the functionality and real-time requirements of the system are met for the tested scenarios. However this technique is inherently flawed, because no assurance can be given about the behavior of the entire system. We identify the source of problem for such techniques as the lack of a timed semantic foundation combined with the inherent nondeterminism in threads [10].

These problems can be addressed by using a distributed discrete-event (DE) model of computation (MoC). Though normally used for simulation (of hardware, networks, and systems of systems, for example), by carefully binding real time with model time at sensors, actuators, and network interfaces, DE can be used for distributed embedded systems [22]. The advantage of using DE as a semantic foundation is that it is simple, time-aware, deterministic, and natural as a specification language for many applications.

In the DE MoC, components send time-stamped events to one another [21]. The time stamps may be integers (as is typical with hardware description languages such as Verilog and VHDL), floating point numbers (as is typical in network simulators), or members of any totally ordered set.

Distributed DE simulation is an old topic [6]. The focus has been on accelerating simulation by exploiting parallel computing resources. A brute-force technique for distributed DE execution uses a single global event queue that sorts events by time stamp. This technique, however, is only suitable for extremely coarse grained computations, and it provides a vulnerable single point of failure. For these reasons, the community has developed distributed schedulers that can react to time-stamped events concurrently. So-called "conservative" techniques process time-stamped events only when it is known to be safe to do so [3, 19]. It is safe to process a time-stamped event if we can be sure that at no time later in the execution will an event with an earlier time stamp appear that should have been processed first. So-called "optimistic" techniques [8] speculatively process events even when there is no such assurance, and roll back if necessary.

*The authors can be reached at {tfeng,eal,jiazou, matic, hiren}@eecs.berkeley.edu, yzhao@google.com, and Patricia.Derler@cs.uni-salzburg.at. 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), #0647591 (CSR-SGER), 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.

For distributed embedded systems, the potential for roll back is limited by actuators (which cannot be rolled back once they have had an effect on the physical world) [5]. Established conservative techniques, however, also prove inadequate. In the classic Chandy and Misra technique [3, 19], each compute platform in a distributed simulator sends messages even when there are no events to convey in order to provide lower bounds on the time stamps of future messages. This technique carries an unacceptably high price in our context. In particular, messages need to be frequent enough to prevent violating real-time constraints due to waiting for such messages. Messages that only carry time stamp information and no data are called "null messages." These messages increase networking overhead and also reduce the available precision of real-time constraints. Moreover, the technique is not robust; failure of single component results in no more such messages, thus blocking progress in other components. Our work is related to several efforts to reduce the number of null messages, such as [20], but makes much heavier use of static analysis.

The key idea of Zhao, Liu and Lee in [22] is to leverage static analysis of DE models to achieve distributed DE scheduling that is conservative but does not require null messages. The static analysis enables independent events to be processed out of time stamp order. For events where there are dependencies, the technique goes a step further by requiring clocks on the distributed computational platforms to be synchronized with bounded error. In this case, the mere passage of time obviates the need for null messages.

This paper reviews the PTIDES programming model (pronounced "tides," where the "P" is silent, as in "Ptolemy"), an acronym for *programming temporally integrated distributed embedded systems*. PTIDES brings forth the advantages that it 1) builds on top of a strong timed semantic foundation, 2) provides a mathematical framework for presenting strategies that explore concurrency of implementations, 3) allows deterministic schedulability analysis, and 4) eases specification of real-time constraints. In reviewing PTIDES, we revisit the static analysis techniques of [22] and define a family of execution strategies based on it. However, our focus here is to present an execution strategy that assumes that platforms have clocks for synchronization and attempts to aggressively process events on the computing platforms. We also establish a relationship between model time and real time at sensors, actuators and network interfaces. These relationships, clock synchronization protocols and the static analysis are used to: 1) allow independent events to be processed out of time stamp order, 2) replace the need to send null messages across distributed computing platforms as done in [19], 3) determine when events are *safe to process*, and 4) present a sketch for an implementation of a PTIDES model. These are our main contribution in this paper.

2. MODEL TIME AND REAL TIME

We specify DE models using the actor-oriented [11] approach. In this case, actors are concurrent components that exchange time-stamped events via input and output ports. The input ports receive time-stamped messages from other actors, and the output ports send time-stamped messages to other actors. Actors react to input messages by "firing," by which we mean performing a finite computation and possibly sending output messages. An actor may also send a time-stamped message to itself, effectively requesting a future firing.

The "time" in time stamps is *model time*, not *wall clock time*, which is also referred to as *real time* or *physical time* in this paper. DE semantics is agnostic about when in real time time-stamped events are processed. All that matters is that each actor process input events in time-stamp order. That is, if it fires in response to an input event with time stamp τ, it should not later fire in response to an input event with time stamp less than τ.

The semantics of DE models is studied in [12, 17, 16, 14]. In particular, the structure of model time is important for dealing correctly with simultaneous events and feedback systems. For the purposes of this paper, we only care that there are policies for dealing predictably with multiple events with identical time stamps. To be concrete, we will assume that time stamps are elements of the set $\mathbb{R}^+ \cup \{\infty\}$. In full generality, however, our techniques work for any set of time stamps that is totally ordered, has a top and a bottom, and has a closed addition operator.

Since we are focused on distributed embedded systems rather than distributed simulation, some of the actors are wrappers for sensors and actuators. Sensors and actuators interact with the physical world, and we can assume that in the physical world, there is also a notion of time. To distinguish it from model time, we refer to *real time*. Real time has its own subtleties, of course, even without getting into relativistic effects or quantum entanglement. In a classical Newtonian notion of time, we can imagine an oracle that oversees the execution of a distributed system and maintains a single coherent global notion of real time. With such a notion, we can talk about components in the system performing actions "simultaneously." However, such an oracle remains fictional. In a distributed system, there is no (known) mechanism by which all components can precisely coordinate their notions of real time.

For the purposes of this paper, we assume a classical Newtonian notion of physical time, and assume that each compute platform in a distributed system maintains a clock that measures the passage of physical time. These clocks are not perfect, so each platform has a distinct local notion of physical time. We assume further than were there a Newtonian oracle that could simultaneously compare the notions of real time on the distinct platforms, that we could find a bound on the discrepancies between clocks. That is, at any global instant, any two clocks in the system agree on the notion of real time up to some bounded error.

Such synchronized clocks turn out to be quite practical [9]. We have had available for some time generic clock synchronization protocols like NTP [18]. Recently, however, techniques have been developed that deliver astonishing precision, such as IEEE 1588 [7]. Hardware interfaces for Ethernet have recently become available that advertise a precision of 8ns over a local area network. Such precise clock synchronization offers truly game-changing opportunities for distributed embedded software.

We assume that model time and real time are disjoint, but

that they can be compared. That is, we assume that model time is in fact a representation of real time, even though time-stamped events can occur at arbitrary physical times. In our DE models, an actor that wraps a sensor, however, cannot produce time-stamped events at arbitrary times. In particular, it will produce a time-stamped output only after physical time (the local notion of physical time) equals or exceeds the value of the time stamp. That is, the time stamp represents the physical time at which the sensor reading is taken, and hence it cannot appear at a physical time earlier than the value of the time stamp.

An actor that wraps an actuator has a complementary constraint. A time-stamped input to such an actor will be interpreted as a command to produce a physical effect at (local) physical time equal to the time stamp. Consequently, the model-time time stamp is a physical-time deadline for delivery of an event to an actuator.

We will also impose timing constraints at network interfaces. A network connection carries time-stamped events from one compute platform to another. We will abstract such an interface as an actor with one input port and one output port. Events presented at the input port will appear unchanged at the output port (in particular, they will have the same time stamp). Similar to actuators, however, we will require that an event with time stamp τ be delivered to the input of the network interface before real time exceeds τ. Moreover, we will assume a bounded network delay, so that the real time that elapses between delivery of such an event to the network interface and appearance of the event at the output of the network interface is bounded. Taken together, these constraints guarantee an upper bound, relative to the time stamp, on the real time at which a time-stamped event is delivered by the network to its destination. A bounded network delay is realizable in real-time networks such as TTA or FlexRay, or by over-provisioning in other networks.

At actors that are neither sensors or actuators, there is no relationship between real and model time. At these actors, input events must be processed in model-time order, but such processing can occur at any real time (earlier or later than the time stamp).

3. THE PTIDES EXECUTION STRATEGY

We leverage static dependency information between actors to develop an execution strategy for discrete-event models. This strategy is general in the sense that it allows for different implementations targeting a variety of computer architectures. An implementation and a time-synchronized architecture are discussed in the next section making use of this strategy.

3.1 Model Structure and Events

We assume a port to be either an input port or an output port. This is without loss of generality, because a port that is an input port and an output port at the same time can be considered as two distinct ports. We further assume ports to be interconnected by a fixed and static network, where each input port is connected to at most one output port. For communication networks that are allowed to change dynamically, it is possible to recompute the dependency information for the changed parts on the fly, which is beyond the scope of this paper.

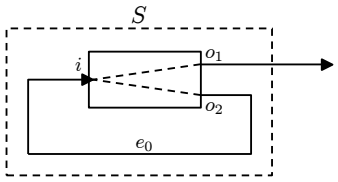

Figure 2: A source actor triggered by an initial event

Our execution strategy for distributed discrete-event systems can be viewed as a generalization of prior work in this field. Specifically, it relaxes an assumption made by Chandy and Misra in [3] and [19]. We do not require that events sent and received on any connection be ordered by time stamps. Hence, we allow an input port to receive events in arbitrary order.

A further generalization that we make is to allow zero or more *initial events* to be provided to each input port at the start of an execution. They can be generated in the initialization phase of the execution. Their time stamps are unrestricted. Actors process initial events and other events generated during the execution in time stamp order. A special use of initial events is seen at *source actors*, which spontaneously output events whose time stamps have no relationship to real time. In the literature, a source actor usually has no input port but one output port. We instead consider it as an actor with one input port that acquires an initial event e_0 with time stamp 0, as shown in Figure 2. The initial event effectively triggers the first output to both of the source actor's output ports. (The dashed lines in the figure denote dependencies, which we will define next.)

The core of our execution strategy is a way to decide whether it is safe to process an input event. An event is *safe to process* if no other input event for the same actor with a smaller time stamp can affect an output signal that is affected by that event.

3.2 Dependencies

We represent the input-output dependency between ports with *minimum model-time delay*, which is computed statically. It is formulated as a causality interface [15] using min-plus algebra [1]. It constitutes an interface definition [4] for the actors.

A model in our formal representation consists of a set of actors, represented by $\mathcal{A}$. Any actor $\alpha \in \mathcal{A}$ has a set of input ports I_α and a set of output ports O_α. The set of all input ports is $I = \bigcup_{\alpha \in \mathcal{A}} I_\alpha$. The set of all output ports is $O = \bigcup_{\alpha \in \mathcal{A}} O_\alpha$. The set of all ports is $P = I \cup O$.

We require a function $\delta_0 \colon P \times P \to \mathbb{R}^+ \cup \{\infty\}$ to be provided *a priori*, where $\mathbb{R}^+$ is the set of non-negative real numbers. By requiring the return values be non-negative, we explicitly assume the actors we are dealing with to be *causal*, in the sense that their output events are no earlier in model time

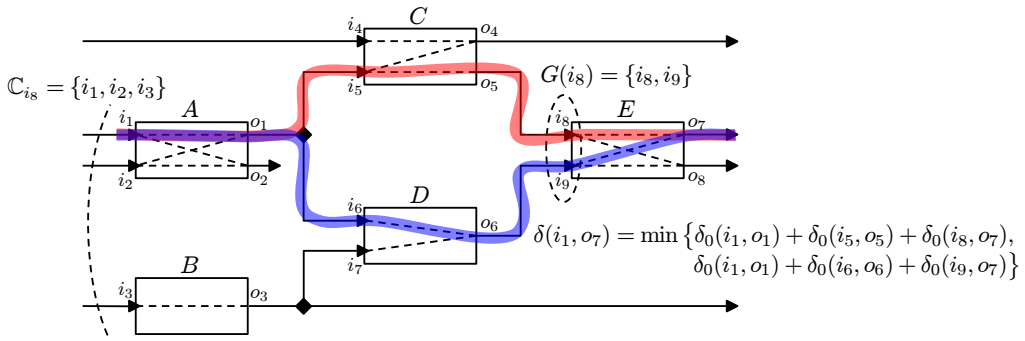

Figure 1: Example with minimum model-time delay, relevant dependency and dependency cut

than the input events that cause them. [13]

δ_0 is defined as follows.

1. If p_1 is an output port, p_2 is an input port, and p_1 is connected to p_2, then $\delta_0(p_1, p_2) = 0$.
2. If $p_1 \in I_\alpha$ and $p_2 \in O_\alpha$ for some $\alpha \in \mathcal{A}$, then $\delta_0(p_1, p_2)$ is provided by the designer of actor α to characterize the dependency between input port p_1 and output port p_2. Alternatively, it may be inferred from a hierarchical definition of α using the methods of [15]. In either case, if $\delta_0(p_1, p_2) = \tau_0$ (where $\tau_0 \in \mathbb{R}^+$), the actor guarantees that an input event at p_1 with time stamp τ has no effect on any event(s) at p_2 with time stamp less than $\tau + \tau_0$.
3. For all other ports p_1 and p_2, $\delta_0(p_1, p_2) = \infty$.

For example, for a *Delay* actor with input port p_1, output port p_2 and a constant model-time delay τ_D between them ($\tau_D \geq 0$), $\delta_0(p_1, p_2) = \tau_D$. For a *VariableDelay* actor, whose delay can be changed at run-time but is always non-negative, $\delta_0(p_1, p_2) = 0$. If the events at an input port p_1 never affect those at an output port p_2, then $\delta_0(p_1, p_2) = \infty$.

Some actors have internal state that is affected by input events. We assume the state s of actor α is modeled by an output port $o_s \in O_\alpha$. Most commonly, for any input port $i \in I_\alpha$, we will have $\delta_0(i, o_s) = 0$, indicating that the state is immediately affected by the input. For some actors, $\delta_0(i, o_s)$ may be infinite, indicating that events at i have no effect on the state. It may also be a positive real number τ_s, indicating that the effect on the state of an input with time stamp τ is not observable at outputs with time stamp less than $\tau + \tau_s$. By modeling state as special output ports, our results developed for non-stateful actors are trivially applicable to stateful actors as well.

We will use the model in Figure 1 as a running example to clarify the definitions in this section. In that figure, rectangles represent actors and filled triangles pointing into actors represent input ports of those actors. Output ports are denoted with outgoing arrows. The input ports are labeled i_1 through i_9 and the output ports are labeled o_1 through o_8.

A dashed line in an actor represents predefined non-infinity dependency between the connected input port and output port. (The concrete value is omitted to reduce clutter.) For example, the dashed line between i_1 and o_1 in actor A implies that $\delta_0(i_1, o_1)$ is statically known to be a number in $\mathbb{R}^+$.

A *path* from port p_1 to p_n is a sequence of ports $[p_1, p_2, \cdots, p_n]$ for some $n > 0$. A *subpath* is a sequence of consecutive ports in a path. (It is also called a *substring* in the literature.) In Figure 1, $[i_1, o_1, i_5, o_5, i_8, o_7]$ is a path, and $[i_5, o_5, i_8]$ is a subpath.

We define $\delta_P(\rho)$ for path $\rho = [p_1, p_2, \cdots, p_n]$ as the model-time delay on the path as follows. If $n = 1$, then $\delta_P(\rho) = 0$. Otherwise,

$$\delta_P(\rho) = \sum_{k=1}^{n-1} \delta_0(p_k, p_{k+1}).$$

To continue with the previous example in Figure 1, $\delta_P([i_1, o_1, i_5, o_5, i_8, o_7]) = \delta_0(i_1, o_1) + \delta_0(i_5, o_5) + \delta_0(i_8, o_7)$, where we observe that $\delta_0(o_1, i_5) = \delta(o_5, i_8) = 0$.

Now we are ready to define the minimum model-time delay for arbitrary pairs of ports with function $\delta : P \times P \to \mathbb{R}^+ \cup \{\infty\}$. For any $p_x, p_y \in P$, $\delta(p_x, p_y)$ is defined as follows,

$$\delta(p_x, p_y) = \min\{\delta_P(\rho) \mid \rho \text{ is a path from } p_x \text{ to } p_y\}$$

That is, $\delta(p_x, p_y)$ is the smallest model-time delay on any path from p_x to p_y. In Figure 1, there are only two paths from i_1 to o_7 that may not yield ∞, so the minimum model-time delay from i_1 to o_7 is:

$$\begin{aligned}\delta(i_1, o_7) &= \min\{\delta_P([i_1, o_1, i_5, o_5, i_8, o_7]),\\ &\qquad \delta_P([i_1, o_1, i_6, o_6, i_9, o_7])\}\\ &= \min\{\delta_0(i_1, o_1) + \delta_0(i_5, o_5) + \delta_0(i_8, o_7),\\ &\qquad \delta_0(i_1, o_1) + \delta_0(i_6, o_6) + \delta_0(i_9, o_7)\}\end{aligned}$$

The minimum model-time delay function δ can be computed in a static analysis before execution. This reduces the workload of the run-time DE scheduler.

3.3 General Execution Strategy

In this section, we discuss our execution strategy for distributed discrete-event systems. It is general enough to serve

as the basis of a variety of concrete implementations of execution policies. For the ease of this discussion, we make an additional assumptions that, conceptually, an input queue is maintained for each input port. An actor removes events from its input queues only when those events are processed and the generated output events (if any) are delivered to the input queues of the receiving ports. This assumption does not affect the applicability of our general execution strategy. Optimization is possible by employing less input queues, as is done in an implementation described in the next section.

For an actor α to decide whether it is safe to process event e at input port $i \in I_\alpha$, it is not enough to have only the minimum model-time delay from other ports to i. This is because we also need to consider events received at α's other input ports, if any. If events at those other input ports affect events at the same output port (which could be the state of the actor), then e and those events must be processed in the order of their time stamps.

This leads us to extend the notion of dependency by considering input ports of an actor that affect the same output port. We define function $G : I \to 2^I$ to return a complete *group of ports* of the same actor that we need to consider before processing an event at a given port. For any $i \in I_\alpha$,

$$G(i) = \left\{ i' \mid i' \in I_\alpha \wedge \exists o \in O_\alpha.\ \left(\delta_0(i,o) < \infty \wedge \delta_0(i',o) < \infty\right) \right\}$$

In particular, i itself is a member of $G(i)$ if events at it affect any output port of α. $G(i) = \emptyset$ if and only if $\forall o \in O_\alpha,\ \delta_0(i,o) = \infty$. In Figure 1, $G(i_8) = \{i_8, i_9\}$.

A set $\mathbb{C}_i \subseteq I$ is called a *dependency cut* for input port $i \in I$ if it is a minimal set of input ports that satisfies the following condition.

> *For any $i_y \in G(i)$ and any path ρ to i_y satisfying $\delta_P(\rho) < \infty$, there exist input port $i_x \in \mathbb{C}_i$ and path ρ' from i_x to i_y satisfying $\delta_P(\rho') < \infty$, such that either ρ is a subpath of ρ' or ρ' is a subpath of ρ.*

Intuitively, a dependency cut for i is a "complete" set of ports on which i depends. Completeness in this case means that for each port in $G(i)$, all ports it depends on will be accounted for in $\mathbb{C}_i$, either directly by being included or indirectly by having either upstream or downstream ports included.

Again using Figure 1 as an example, the dashed curve depicts one possible dependency cut for i_8, namely $\mathbb{C}_{i_8} = \{i_1, i_2, i_3\}$.

The dependency cut for a given input port is not unique. For input port i, $G(i)$ is obviously one of the possible dependency cuts.

A dependency cut can be used to determine whether it is safe to process an input event. The strategy is stated as follows:

> *Given a dependency cut $\mathbb{C}_i$ for input port i of actor α, an event e at i with time stamp τ is safe to process if for any $i_x \in \mathbb{C}_i$ and any $i_y \in G(i)$,*
>
> 1. *i_x has received all events with time stamps less than or equal to $\tau - \delta(i_x, i_y)$, and*
> 2. *for any $i_z \in I$ such that $\delta(i_x, i_z) < \infty$,*
>   (a) *if $i_z \in G(i)$, all events in i_z's input queue have time stamps greater than or equal to τ,*
>   (b) *if $i_z \notin G(i)$, all events in i_z's input queue have time stamps greater than $\tau - \delta(i_z, i_y)$*

Intuitively, these conditions ensure that actor α has received all events that can possibly invalidate the processing of e. The first condition ensures that no future events will be received at the ports in the dependency cut $\mathbb{C}_i$ that can possibly affect an event at the ports in $G(i)$. The second condition ensures that no event at the ports between any port in $\mathbb{C}_i$ and any port in $G(i)$ can possibly affect an event at the ports in $G(i)$. (Notice that if $\delta(i_z, i_y) = \infty$, then this condition is trivially satisfied.) The two conditions together serve as a guarantee that the ports in $G(i)$ have received all events with time stamp τ.

This principal, of course, can be satisfied by a classical DE scheduler, which uses a global event queue to sort events by time stamp. In this case, the oldest event (with the least time stamp) can always be processed. This assumes, of course, that all actors are causal, so events that are produced in reaction to processing an event always have a time stamp at least as great as that of the processed event.

However, this principle relaxes the policy considerably, clarifying that we only need to know whether an event is "oldest" among the events that can appear in a dependency cut. We do not need to know that it is globally oldest. Of course, the choice of dependency cuts will have a significant effect on how much this relaxes the scheduling. We discuss that in the next section.

4. PTIDES IMPLEMENTATION

The general execution policy requires knowing that events satisfying certain time-stamp conditions have been received. But it does not specify how this can be known. A classical DE scheduler used for simulation puts all events in a single sorted event queue, and hence can easily determine when these conditions have been satisfied. However, if the execution is distributed, then a single event queue is not practical. Separate event queues must be maintained on each execution platform, and time-stamped events can arrive unpredictably over the network. If the model includes sensors components that can produce events at arbitrary times, then a similar problem occurs. The events on a sorted event queue do not automatically provide information about what events might appear later.

In this section, we specialize the general execution policy to handle these situations. We use the notion of *real-time ports* [22], which are ports where time stamps have a particular defined relationship to real time. These relationships are discussed intuitively above in section 2, where actors that

wrap sensors, actuators, and network connections have such real-time ports.

4.1 Real Time Ports

As in section 3.3, we assume that each input port maintains a queue of as-yet unprocessed events. An input port that is a real-time port has the constraint that at any real time t, for each event e in the queue,

$$t \leq \tau, \tag{1}$$

where τ is the time stamp of event e. The input ports of actuator actors and network interface actors (see Figure 4 below) are normally such real-time ports.

This constraint imposes a real-time deadline on delivery of each event to the queue, because if the event is delivered at a real time $t > \tau$, then upon delivery, there will be an event in the event queue that violates the constraint. Moreover, this constraint imposes a deadline on the processing of the event, because if the actor is not fired prior to real time $t = \tau$, then the event will remain on the queue past the point where it satisfies the constraint.

An output port o that is a real-time port has the constraint that if it produces an event e with time stamp τ at real time t, then

$$\tau + d_o \geq t \geq \tau \tag{2}$$

where $d_o \in \mathbb{R}^+ \cup \{\infty\}$ is a parameter of the port called its *maximum delay*. Here, what we mean by "producing an event" is delivering it to the input queue of all destination input ports.

Output ports of sensor and network interface actors are normally real-time ports. For a sensor, the time stamp of an output event represents the time at which the reported measurement was taken. The constraint that $t \geq \tau$ specifies that the sensor can only report about past properties of the physical environment, not future properties. The constraint $\tau + d_o \geq t$ asserts that the sensor does such reporting in bounded time, if $d_o < \infty$.

A *network interface* actor α abstracts a network connection, and has a single input port i called *network input port*, a single output port o called *network output port*, both of which are real-time ports. In this case, d_o is a bound on network latency. We make a special introduction of this actor here because this is the only actor that is used for different platforms to communicate between each other within a distributed systems.

Note that for any real-time port p, if there is another port p' where $\delta(p', p) < \infty$, then the real-time constraints on p imply real-time constraints on p'. Whether those real-time constraints can be satisfied is the *schedulability* question [22], which we do not address in this paper. Our focus instead is on a scheduling policy that is correct under the assumption that the model is schedulable.

4.2 Examples

The example in Figure 3 illustrates how we can use the real-time properties of ports. This model has one sensor and

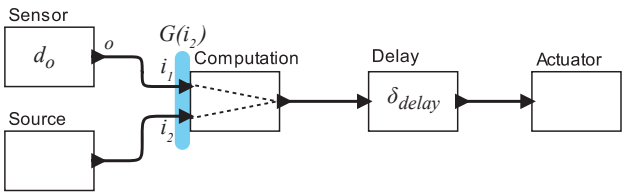

Figure 3: Simple DE model with a sensor and an actuator.

one actuator, as well as a source actor *Source*, which produces events with no constraints between model time and real time. This model has two real-time ports, output port o of the *Sensor* actor and input port of the *Actuator* actor.

Suppose that the source actor has produced an event e with time stamp τ, and that event is the *top* event (i.e., the one with the smallest time stamp) in the queue of the input port i_2 of the *Computation* actor. When is it safe to process that event (i.e., to fire the *Computation* actor)?

Following the general execution policy, we need to first choose a dependency cut $\mathbb{C}_{i_2}$. Recall that a simple choice we always have is $\mathbb{C}_{i_2} = G(i_2) = \{i_1, i_2\}$. With this choice, the general execution policy tells us that we can process the event if i_1 has received all events with time stamp less than or equal to τ. Because of the constraint (2) on o, this is guaranteed to have happened when real time t is greater than $\tau + d_o$.

Notice that this scheme allows us to check whether an event is safe to process by taking advantage of the properties of the real-time output port of the sensor actor and checking an event time stamp against the current real time. This motivates us to develop a cut selection algorithm that chooses input ports that are connected to real-time output ports which is the main focus of the next subsection.

If we assume, as usually is the case, that the *Sensor* actor delivers events in time-stamp order, then it is also safe to process e if the top event on i_1 has time stamp $\tau' > \tau$. Indeed, this latter condition is what Chandy and Misra rely on. This condition can be used in combination with the above real-time condition to allow for earlier processing of some events.

Notice that the real-time execution policy has an important robustness property. If the *Sensor* actor fails, and stops producing output events at all, then events from *Source* are processed anyway. The *Sensor* cannot block the *Source*. This property is missing from Chandy and Misra's technique.

If d_o is large or infinite, then the real-time policy does not help. In this case, it would be difficult to satisfy the real-time constraints if there is a bounded model-time delay path from a sensor actor to any actuator or network interface actor. This can be checked during static analysis of the model.

Consider next the example in Figure 4. Here, we assume two distinct computational platforms (enclosing grey boxes) with a network connection between them. The output ports of the *Sensor* and *NetworkInterface* actors, and the input

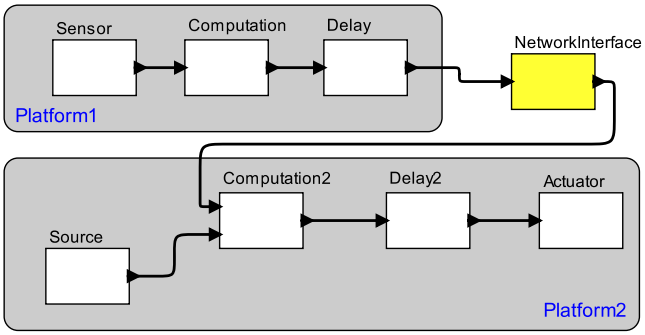

Figure 4: Distributed DE model with a sensor on one platform and an actuator on another.

ports of the *NetworkInterface* and *Actuator* actors, are real-time ports. A Chandy-and-Misra style of execution would require sending frequent null messages between the two platforms. We will eliminate these null messages while still preserving conservative style of execution.

In this case, the sub-model on platform 2 looks just like the model in Figure 3 if we treat the *NetworkInterface* actor and everything before it as a sensor actor. Similarly, the sub-model on platform 1 is similar to the one in Figure 3, but where the *NetworkInterface* actor and everything after it is considered analogous to an actuator actor. Thus, each platform can implement a similar execution policy to the one we used for Figure 3. Notice that this execution policy is distributed, in that the platforms need not consult one another to make scheduling decisions. They only need to share a notion of real time (with some bounded error that must be taken into account).

4.3 Dependency Cut Selection

Motivated by the above examples, we provide an algorithm for choosing a suitable dependency cut C_i for a given input port i. The goal of this algorithm is to find a dependency cut that consists of ports that relate model time to real time, in order to completely eliminate the need for null messages across computing platforms. As defined in the previous section, real-time ports have exactly this property, thus all real-time input ports and inputs connected to real-time output ports are candidates for the dependency cut. Another candidate for the dependency cut would be an input port of a source actor as introduced in section 3.1. The cut is formed by first including these candidates into the cut and then ensuring the cut is minimal.

Figure 5 shows an example of the cut C_{i_n} of port i_n for each $n \in \{1, 2, 3\}$ (notice that the cuts for all ports in $G(i_n)$ are identical). Ports i'_1, i'_4 and i'_5 are candidates because they are either real-time input ports or inputs connected to real-time output ports, while i'_2 and i'_3 are candidates because they represent input ports of source actors. Now to ensure the cut is minimal, we see that both i'_4 and i'_5 could be reached by traversing the graph from i'_3, thus they are removed from the cut. Thus we have $C_{i_n} = \{i'_1, i'_2, i'_3\}$ for each $n \in \{1, 2, 3\}$.

Using the above example as motivation, we formally describe the cut selection procedure using standard graph notation and algorithms. Let us define a directed graph $G = (V, E, W, L)$ that describes the PTIDES model being examined, where

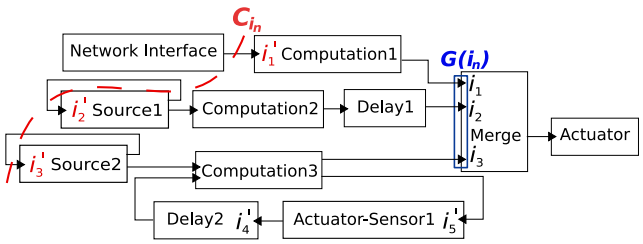

Figure 5: Distributed DE model with a sensor on one platform and an actuator on another.

$$
\begin{aligned}
&V = P,\\
&E = \{(v_1, v_2) \mid \forall v_1, v_2 \in V.\ \delta_0(v_1, v_2) < \infty\\
&\quad \wedge\, v_1 \text{is not a network input port}\\
&\quad \wedge\, v_2 \text{is not a network output port}\},\\
&W(v_1, v_2) = \delta_0(v_1, v_2), \text{and}\\
&L(v) = \begin{cases} 1 & \text{if } ((v', v) \in E \wedge v' \text{ is real-time output port})\\ & \quad \vee\, v \text{ is a real-time input port}\\ & \quad \vee\, v \text{ is a source input port,}\\ 0 & \text{otherwise.} \end{cases}
\end{aligned}
$$

Here, V is the set of ports of the model; E is the set of edges; $W : E \to R^+ \cup \{\infty\}$ is a weight function that maps each edge to its minimal model time delay; $L : V \to \{0, 1\}$ is a labeling function that determines whether each vertex $v \in V$ is a candidate for dependency cut. This graph constructs the original model, while disconnecting the edges from network input ports or towards network output ports. Since using network interface actors is the only way data could be communicated across platforms, this in effect severs all connections across computation platforms. This ensures the cut always consists of the ports from the same platform as port group $G(i)$.

We determine a dependency cut $\mathbb{C}_i$ for port i in a two-step algorithm as follows:

1. *for each $v \in V$ such that $L(v) = 1$, start from v and traverse G. If the traversal leads to a vertex $i' \in G(i)$, add v to $\mathbb{C}_i$.*
2. *After step 1 is completed, start from each $v \in \mathbb{C}_i$ and traverse G. If during this process a vertex v' is reached such that $v' \in \mathbb{C}_i$, update $\mathbb{C}_i$ by removing v' from the cut.*

Notice that step *1* of our algorithm ensures the cut is complete. Step *2* of our algorithm guarantees the cut is minimal, since the traversal ensures any two vertices belonging to a path will not be part of the cut at the same time. Also, in the case where we have more than one candidate for the cut within a cycle, the above algorithm is not deterministic in specifying which one among them will become a member

of the cut. However, we do not care which port is chosen through step *2* of the algorithm, as long as exactly one among them is chosen, and step *2* ensures exactly that.

Also notice that if our only goal of the cut selection algorithm is to find the dependency cut, then any graph traversal algorithm could be used in step *1*. However, during traversal, it is also beneficial for us to obtain the value of minimum model time delay δ from each member of the cut to each element of $G(i)$. This value could be obtained by using a shortest path algorithm as a graph traversal algorithm.

4.4 Safe-to-Process Analysis

Using the dependency cut obtained by the algorithm in subsection 4.3, in this subsection we present an instance of the general execution strategy described in section 3.3, i.e., we present conditions for safe processing of events that obeys the DE semantics.

We first define the *real-time delay* function $D\colon I \rightarrow \mathbb{R}^+ \cup \{-\infty\}$ that maps each port $p \in I$ to

$$D(p) = \begin{cases} 0 & \text{if } p \text{ is a real-time input port (1),} \\ d_o & \text{if } p \text{ is a non-real-time input port} \\ & \quad \text{connected to a real-time output port (2),} \\ -\infty & \text{otherwise (3).} \end{cases}$$

Recall that d_o is the real-time delay specific to a real-time output port, as defined in equation (2) of section 4.1 (i.e., $\tau + d_o \geq t$). Note also that equation (1) in the same section can be rewritten as $\tau + 0 \geq t$. Thus, for each input port p that satisfies condition (1) or (2) of the definition $D(p)$ given above we have that an event with time stamp τ is delivered to the input queue of p at real time t such that $\tau + D(p) \geq t$. For all other ports, i.e., for case (3) above, $D(p) = -\infty$ because no such constraint between real time and model time exists.

Assume that model-time delay function δ and real-time delay function D are given and that for each input port $i \in I$ dependency cut $\mathbb{C}_i$ is determined according to the algorithm from section 4.3. In that case, a procedure for determining safe-to-process events based on the general execution strategy presented in section 3.3 can be given as follows:

An event at input port $i \in I$ with time stamp τ is safe to process when:

1. *(a) real time has exceeded*

$$\tau + \max_{p \in \mathbb{C}_i, i' \in G(i)} \{D(p) - \delta(p, i')\},$$

   and

   (b) at each source actor input port $p \in \mathbb{C}_i$ an event has been received with time stamp greater than

$$\tau + \max_{i' \in G(i)} (-\delta(p, i')),$$

   and

2. *for each port $p' \in I$ such that there exists port $p \in \mathbb{C}_i$ with $\delta(p, p') < \infty$, each event in input queue of p' has time stamp*

   (a) greater than or equal to τ for $p' \in G(i)$,

   (b) greater than $\tau + \max_{i' \in G(i)}(-\delta(p', i'))$ for $p' \notin G(i)$.

The conditions *1* and *2* correspond to the conditions *1* and *2* of the general execution strategy respectively. The forms of the corresponding conditions *2* are similar. However, conditions *1* differ because the intention here is to take advantage of the particular dependency cut $\mathbb{C}_i$. If $D(p) > -\infty$ for an input port $p \in I$, then constraint $\tau + D(p) \geq t$ between real time and model time can be exploited as explained above. In addition, condition *1(a)* takes into account all ports of $\mathbb{C}_i$ and $G(i)$ and model-time delay δ between those. Thus, after the specified real time the event can be safely processed because no event with an earlier time stamp can arrive at the ports in the group $G(i)$. Note that for each $i \in I$, given $\mathbb{C}_i$ and $G(i)$, the value of $\max_{p \in \mathbb{C}_i, i' \in G(i)}\{D(p) - \delta(p, i')\}$ can be computed at compile time.

The condition *1(b)* addresses the condition *1* of the general execution strategy for ports $p \in \mathbb{C}_i$ for which $D(p) = -\infty$, i.e., for input ports of source actors on the dependency cut. Namely, the model of a source actor presented in section 3.1 guarantees that the events in the queue of its input port are delivered in time stamp order. Thus, if condition *1(b)* is satisfied for such a port p then all events with time stamps less than or equal to $\tau + \max_{i' \in G(i)}(-\delta(p, i'))$ have been received at p, i.e., the condition *1* of the general execution strategy is satisfied. Note that we do not assume that events at other input ports are received in time stamp order. If that is the case, the execution strategy could be made simpler. In particular, if this is true for all ports of dependency cut $\mathbb{C}_i$ the condition *1(a)* could be relaxed similar to condition *1(b)* to form a less conservative strategy.

4.5 Simulation of PTIDES Models

We developed a simulation environment for the PTIDES programming model as an experimental domain in Ptolemy II [2]. The Ptolemy framework is a Java-based environment for modeling and simulation of heterogeneous concurrent systems. Ptolemy supports an actor-oriented design methodology. A special actor in a Ptolemy model, the director, manages the interaction of other actors thus representing the model of computation. The implementation of a model of computation in Ptolemy is called a domain.

Figure 6 shows an example of a PTIDES model in Ptolemy. A PTIDES model consists of platforms represented by composite actors on the top-level of the model. These platforms are supposed to execute in parallel and communicate via events. Inside each platform, the set of actors include sensors, actuators, source actors, delay actors or other computation actors. The worst case execution time can be associated with an actor.

To keep track of the execution of actors, the PTIDES domain maintains a notion of *physical time*. The physical time is not the real time but a new model time that simulates real time and is manipulated by the framework. The physical time is different from the model time used in discrete event simulation which defines the execution semantics. During

simulation, the physical time is used to determine whether an event is safe to process.

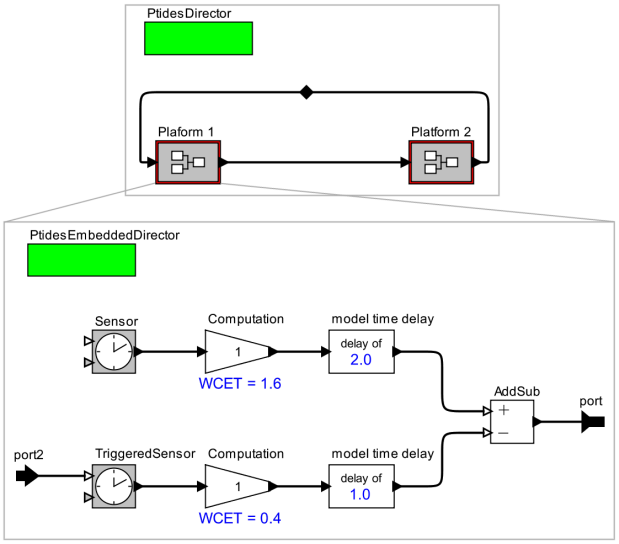

Figure 6: A PTIDES model in Ptolemy.

The interaction of platforms on the top-level is managed by the *PtidesDirector* and the interaction of actors inside a single platform is directed by the *PtidesEmbeddedDirector*. The sequence diagram in Figure 7 describes the main steps during the simulation of a PTIDES model. Before starting the simulation, the PtidesDirector creates a new thread for every platform. During the simulation, the threads execute in parallel.

The dependency cut and execution strategy we use for simulation of PTIDES models are those presented in sections 4.3 and 4.4 respectively. At each simulation step a set of events that are safe to process is determined. These events are taken from the event queues associated with input ports of actors inside a platform. The PtidesEmbeddedDirector director of a platform does not use a single platform-level event queue. Note that the execution strategy does not necessarily select for processing the event with the least time stamp. Note also that an event might be safe to process according to PTIDES model even though it is not possible to process it on the platform at the current physical time. An example is an event that is safe to process but another actor is still in execution and cannot be preempted. If at the current physical time there are no safe to process events the platform requests the PtidesDirector to resume execution of that platform at a future instance of physical time and the platform thread waits. When all threads are waiting, the PtidesDirector increases physical time and notifies all platforms. The platforms then continue processing events.

4.6 A Simple PTIDES Implementation

In this section we briefly discuss a modification of the strategy presented in section 4.4 that can result in implementations with relatively simple checks. In this approach an event queue is available within each platform to store and sort all events within the platform. Also, in contrast to the strategy from section 4.4 or 4.5, only the event with the least

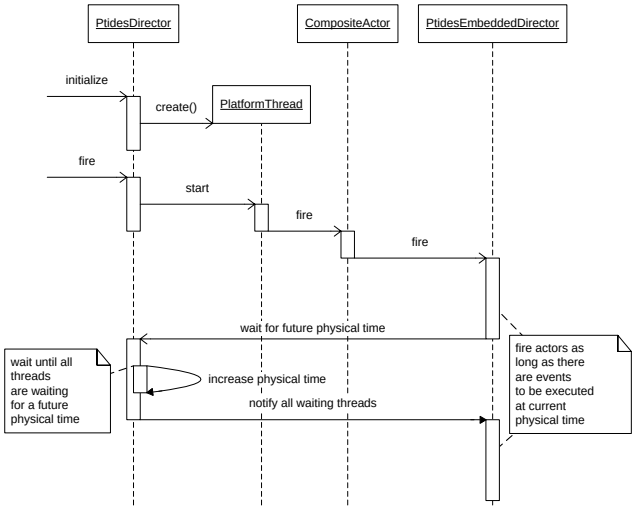

Figure 7: Simulation of a PTIDES model in Ptolemy.

time stamp in the queue is checked whether it is safe to process. The advantage in this scheme is that we only have to deal with one event at a time and the event queue allows us to compare time stamps implicitly, thus greatly simplifies the execution strategy. In particular, if only the top event of the queue is considered for processing, there is no need to check the condition *2* of the general execution strategy from section 3.3 because it will always be satisfied. However, since an event in the queue with a larger time stamp may be safe to process even when the top event is not, this approach may result in more conservative strategy. This depends both on the underlying graph and the characteristics of the input event sequences.

Note that the condition *1(b)* of the strategy in section 4.4 is needed because no relationship between real time and model time is available for all ports of the dependency cut. Thus, checking time stamp against real time is not applicable in general. However, this lack of information also implies that the source actors can produce a large number of events and potentially overflow the event queue. We can throttle the execution of a source actor by simply putting a fixed sized buffer at the output of each source actor. If no events are currently safe to process, the source actor can keep producing events until the buffer is full. If the buffer is full, the firing of the source actor is blocked. Aside from the maximum buffer limit, we also need a way to ensure at least one event is at the output of a source actor at any time. Take Figure 3 for instance, the event queue will be used to compare time stamps of events at each port of $G(i_1)$, and as long as one event is present at port i_2, we will always be able to process events in time stamp order without the need to check condition *1(b)*. When the number of events in the buffer becomes zero we would fire the source actor exactly once in order to ensure one event is always available.

Thus, in this implementation our safe-to-process analysis can be simplified to a time stamp checking against real-time as follows:

An event at input port $i \in I$ with time stamp τ is safe to process when the real time has exceeded

$$\tau + \max_{p \in \mathbb{C}_i, i' \in G(i)} (D(p) - \delta(p, i')).$$

If an event can be processed immediately, it is passed to the corresponding actor for processing. If no event can be processed immediately, the real time at which an event can be processed can be determined, and a timed interrupt can be set to occur at that time. Our preliminary implementation of this scheme runs on a set of Linux-based Agilent demo boxes equipped with FPGAs that perform time synchronization according to the IEEE 1588 Precision Time Protocol over the local Ethernet network.

5. SUMMARY

We first presented a general execution strategy that enables correct event processing for timed models with discrete-event semantics. The strategy allows independent events to be processed out of time stamp order. For our PTIDES programming model we established relationships between model time and real time for certain actors. Motivated by open and precise clock synchronization protocols that are becoming available for distributed real-time systems, we next presented an instance of the general execution strategy that makes use of these relationships, clock synchronization and static model analysis for aggressive processing of events. In the presented approach null messages are not needed to synchronize the computing platforms. Beside working further on PTIDES implementation and simulation tools our future work will include scheduling mechanisms for events that are safe to process and the related schedulability problems.

6. REFERENCES

[1] F. Baccelli, G. Cohen, G. J. Olsder, and J.-P. Quadrat. *Synchronization and Linearity: An Algebra for Discrete Event Systems*. Wiley, New York, 1992.

[2] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. Heterogeneous concurrent modeling and design in Java. Technical Report Technical Memorandum UCB/ERL M04/27, University of California, July 29 2004.

[3] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transaction on Software Engineering*, 5(5), 1979.

[4] L. deAlfaro and T. A. Henzinger. Interface theories for component-based design. In *First International Workshop on Embedded Software (EMSOFT)*, volume LNCS 2211, pages 148–165, Lake Tahoe, CA, October, 2001 2001. Springer-Verlag.

[5] T. H. Feng and E. A. Lee. Real-time distributed discrete-event execution with fault tolerance. In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 08)*, St. Louis, MO, USA, April 2008.

[6] R. M. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10):30–53, 1990.

[7] I. Instrumentation and M. Society. 1588: Ieee standard for a precision clock synchronization protocol for networked measurement and control systems. Standard specification, IEEE, November 8 2002.

[8] D. Jefferson. Virtual time. *ACM Trans. Programming Languages and Systems*, 7(3):404–425, 1985.

[9] S. Johannessen. Time synchronization in a local area network. *IEEE Control Systems Magazine*, pages 61–69, 2004.

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

[11] E. Lee, S. Neuendorffer, and M. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers*, 12(3):231–260, 2003.

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

[13] E. A. Lee and A. L. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 17(12):1217–1229, 1998.

[14] E. A. Lee and H. Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems, October 2007.

[15] Y. Z. Lee and E. A. Causality interfaces for actor networks. *ACM Transactions on Embedded Computing Systems (TECS)*, 2008.

[16] X. Liu and E. A. Lee. CPO semantics of timed interactive actor networks. Technical Report EECS-2006-67, UC Berkeley, May 18 2006.

[17] X. Liu, E. Matsikoudis, and E. A. Lee. Modeling timed concurrent systems. In *CONCUR 2006 - Concurrency Theory*, volume LNCS 4137, Bonn, Germany, August 27-30 2006. Springer.

[18] D. L. Mills. A brief history of NTP time: confessions of an internet timekeeper. *ACM Computer Communications Review*, 33, 2003.

[19] J. Misra. Distributed discrete-event simulation. *ACM Computing Surveys*, 18(1):39–65, 1986.

[20] R. D. Vries. Reducing null messages in misra's distributed discrete event simulation method. *IEEE Transactions on Software Engineering*, 16(1):82–91, 1990.

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

[22] Y. Zhao, J. Liu, and E. A. Lee. A programming model for time-synchronized distributed real-time systems. In *Proceedings of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 07)*, pages 259–268, Bellevue, WA, USA, Apr 2007.