Programming Temporally Integrated Distributed Embedded Systems

Yang Zhao
Edward A. Lee
Jie Liu

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-82
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-82.html

May 28, 2006

Acknowledgement

This paper describes work that is part of the Ptolemy project, which is supported by the National Science Foundation (NSF award number CCR-00225610), and Chess (the Center for Hybrid and Embedded Software Systems), which receives support from NSF, the State of California Micro Program, and the following companies: Agilent, DGIST, General Motors, Hewlett Packard, Microsoft, and Toyota.

Programming Temporally Integrated Distributed Embedded Systems

Yang Zhao*, Edward A. Lee*, Jie Liu†,
*EECS Department, University of California at Berkeley,
Berkeley, CA 94720 USA, Email: {ellen_zh, eal}@eecs.berkeley.edu
†Microsoft Research, Redmond, WA 98052, USA. Email: liuj@microsoft.com

***Abstract*— Discrete-event (DE) models are formal system specifications that have analyzable deterministic behaviors in terms of event values and time stamps. However, since time is only a modeling property, they are primarily used in performance modeling and simulation. In this paper, we extend discrete-event models with the capability of mapping certain events to physical time and propose them as a programming model, called PTIDES. We seek analysis tools and execution strategies that can preserve the deterministic behaviors specified in DE models without paying the penalty of totally ordered executions. This is particularly intriguing in time synchronized distributed systems since there is a consistent global notion of time and intrinsic parallelism among the nodes. Based on causality analysis of DE systems, we define *relevant dependency* and *relevant orders* to enable out-of-order executions without hurting determinism and without requiring backtracking. For a given network characteristic, we can check statically whether deploying the model in the network can preserve the real-time properties in the specification.**

I. Introduction

Network time synchronization has the potential to significantly change how we design embedded real-time distributed systems. In the past, time synchronization has been a relatively expensive service. For many applications, it is not really required, since a logical notion of time (which constrains the ordering of events) is sufficient for correctness [15]. Loose coupling of model time with physical time is sufficient for many interactive distributed systems, such as computer games, where human-scale time precision is adequate. In many embedded systems, such as manufacturing, instrumentation, and vehicular control systems, much higher timing precision is essential. Engineers designing such systems resort to specialized bus architectures, such as time triggered architectures, CAN bus, or FlexRay (http://www.flexray.com/). Time synchronization over standard networks, such as provided by NTP [19], does not have adequate time precision for such applications. The recent standardization (IEEE 1588 [1]) of high-precision timing synchronization over ethernet that is compatible with conventional networking technologies promises to significantly change this picture. Ongoing work in time synchronization for wireless networks (see for example RBS [20]) also show considerable promise. Implementations of IEEE 1588 have demonstrated time synchronization as precise as tens of nanoseconds over networks that cover hundreds of meters, more than adequate for many manufacturing, instrumentation, and vehicular control systems. Such precise time synchronization enables coordinated actions over distances that are large enough that fundamental limits (the speed of light, for example) make it impossible to achieve the same coordination by conventional stimulus-response mechanisms.

A key question that arises in the face of such technologies is how they can change the way software is developed. Ideas include elevating the principles of time triggered architectures to the programming language level, as done for example in Giotto [12], and augmenting software component interfaces with timing information. In this paper, we describe a programming model that uses distributed discrete-event techniques [5], [8], [21], but rather than using them for accelerated simulation as previously, we use them as a temporally integrated distributed programming model. Our method relies on time synchronization with a known precision. The precision is arbitrary, so the method applies equally well to extremely fine precision (as is possible with IEEE 1588) as to coarse precision (as achieved by NTP). We call the resulting model PTIDES, Programming Temporally Integrated Distributed Embedded Systems.

discrete-event semantics is typically used for modeling physical systems where atomic events occur on a time line. For example, hardware description languages for digital logic design, such as Verilog and VHDL, are discrete-event languages. So are many network modeling languages, such as OPNET Modeler[2] and Ns-2[3]. Our approach is not to model physical phenomena, but rather to specify coordinated real-time events to be realized in software. Execution of the software will first obey discrete-event semantics, just as done in DE simulators, but it will do so with specified real-time constraints on certain actions. Our technique is properly viewed as providing a semantic notion of model time together with a relation between the model time of certain events and their real time.

Our premise is that since DE models are natural for modeling real-time systems, they should be equally natural for specifying real-time systems. Moreover, we can exploit their formal properties to ensure determinism in ways that evades many real-time software techniques. Network time synchronization makes it possible for discrete-event models to have a coherent semantics across distributed nodes. Just as with distributed DE simulation, it will not be practical nor

[1] http://ieee1588.nist.gov

[2] http://opnet.com/products/modeler/home.html

[3] http://www.isi.edu/nsnam/ns

efficient to use a centralized event queue to sort events in chronological order. Our goal will be to compile DE models for efficient and predictable distributed execution.

We emphasize that while distributed execution of DE models has long been used to exploit parallel computation to accelerate simulation [21], we are not interested in accelerated simulation. Instead, we are interested in systems that are intrinsically distributed. Consider factory automation, for example, where sensors and actuators are spread out physically over hundreds of meters. Multiple controllers must coordinate their actions over networks. This is not about speed of execution but rather about timing precision. We use the global notion of time that is intrinsic in DE models as a binding coordination agent.

For accelerated simulation, there is a rich history of techniques. So-called "conservative" techniques advance model time to t only when each node can be assured that they have seen all events time stamped t or earlier. For example, in the well-known Chandy and Misra technique [6], extra messages are used for one execution node to notify another that there are no such earlier events. For our purposes, this technique binds the execution at the nodes too tightly, making it very difficult to meet realistic real-time constraints.

So-called "optimistic" techniques perform speculative execution and backtrack if and when the speculation was incorrect [13]. Such optimistic techniques will also not work in our context, since backtracking physical interactions is not possible.

Our method is conservative, in the sense that events are processed only when we are sure it is safe to do so. But we achieve significantly looser coupling than Chandy and Misra using a new method that we call *relevant dependency analysis*.

Reflecting the inadequacy of established methods, there has recently been considerable experimentation with techniques for programming networked embedded systems. For example, TinyOS and nesC [9] are designed for programming wireless networked sensor nodes with extreme resource constraints. Although TinyOS/nesC does not address real-time constraints, it provides an innovative concurrency model that supports creating very thin software wrappers around hardware (sensors and actuators). It also blurs the traditional boundary between the programming language and the operating system. Another interesting example is Click [14], which was created to support the design of software-based network routers. Handling massive concurrency and providing high throughput are major design goals in Click, although again it does not address real-time constraints.

An innovative embedded software programming system that does address real-time is Simulink with Real-Time Workshop (RTW), from The MathWorks. Simulink is widely used for designing embedded control systems in applications such as automotive electronics. RTW generates embedded programs from Simulink models. It leverages an underlying preemptive priority-driven multitasking operating system to deliver determinate computations with real-time behavior based on rate-monotonic scheduling [17]. It includes some clever techniques to minimize the overhead due to interlocks in communication between software components. Giotto [12] simplifies this scheme, achieving a model that is amenable to rigorous schedulability analysis, at the expense of increased latency. The Timed Multitasking [18] (TM) model extends this principle to a more event-driven style (vs. periodic).

A rather different approach is taken in the synchronous languages [2]. These languages have a rather abstracted notion of model time and no built-in binding between model time and physical time. In this notion, computations are aligned with a global "clock tick," and are semantically instantaneous and simultaneous. For example, SCADE [3] (Safety Critical Application Development Environment), a commercial product of Esterel Technologies, builds on the synchronous language Lustre [11], providing a graphical programming framework with Lustre semantics. Of the flagship synchronous languages, Esterel [4], Signal [10], and Lustre, Lustre is the simplest in many respects. All the synchronous languages have strong formal properties that yield quite effectively to formal verification techniques, but the simplicity of Lustre in large part accounts for SCADE achieving certification for use in safety critical embedded avionics software.[4] Although highly concurrent, synchronous languages are challenging to run on distributed platforms because of the notion of a global clock tick.

Our emphasis is on efficient distributed real-time execution. Our framework uses model time to define execution semantics, and constraints that bind certain model time events to physical time. A correct execution will simply obey the ordering constraints implied by model time and meet the constraints on events that are bound to physical time.

This paper is organized as following. Section II motivate our programming model using a distributed real-time application. Section III develops the *relevant dependency* concept using causality interfaces [16], and defines the *relevant order* on events based on relevant dependency to formally capture the ordering constraints of temporally ordered events that have a dependency relationship. A distributed execution strategy based on the relevant order of events is presented in section IV. In section V, we show how to analyze whether a discrete-event specification is feasible to be deployed on distributed nodes. Future work is discussed in section VI.

II. Motivating Example

We motivate our programming model by considering a simple distributed real-time application. Suppose that at two distinct machines A and B we need to generate precisely timed physical events under the control of software. Moreover, the devices that generate these physical events respond after generating the event with some data, for example sensor data. We model this functionality with an *actor* that has one input port and one output port, depicted graphically as follows:

[4]The SCADE tool has a code generator that produces C or ADA code that is compliant with the DO-178B Level A standard, which allows it to be used in critical avionics applications (see http://www.rtca.org).

This actor is a software component that wraps interactions with the device drivers. We assume that it does not communicate with any other software component except via its ports. At its input port, it receives a potentially infinite sequence of time-stamped values, called *events*, in chronological order. The sequence of events is called a *signal*. The output port produces a time-stamped value for each input event, where the time stamp is strictly greater than that of the input event. The time stamps are values of model time. This software component binds model time to physical time by producing some physical action at the real-time corresponding to the model time of each input event. Thus, the key real-time constraint is that input events must be made available for this software component to process them at a physical time strictly earlier than the time stamp. Otherwise, the component would not be able to produce the physical action at the designated time.

Figure 1 shows a distributed DE model to be executed on a two-machine, time-synchronized platform. The dashed boxes divide the model into two parts, one to be executed on each machine. The parts communicate via signal s_2. We assume that events in this signal are sent over a standard network as time-stamped values.

The `Clock` actors in the figure produce time-stamped outputs where the time stamp is some integer multiple of a period p (the period can be different for each clock). Upon receiving an input with time stamp t, the clock actor will produce an output with time stamp np where n is the smallest integer so that $np \geq t$. There are no real-time constraints on the inputs or outputs of these actors.

The `Merge` actor has two input ports. It merges the signals on the two input ports in chronological order (perhaps giving priority to one port if input events have identical time stamps). A conservative implementation of this Merge requires that no output with time stamp t be produced until we are sure we have seen all inputs with time stamps less than or equal to t. There are no real-time constraints on the input or output events of the `Merge` actor.

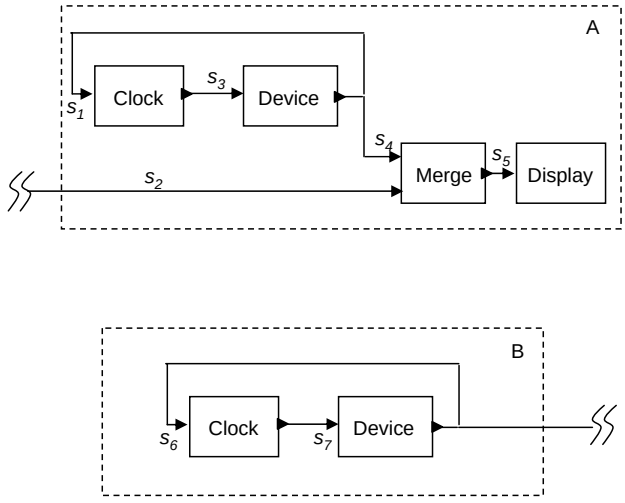

Fig. 1. A simple distributed instrumentation example.

The `Display` actor receives input events in chronological (time-stamped) order and displays them. It also has no real-time constraints.

A brute-force implementation of a conservative distributed DE execution of this model would stall execution in platform A at some time stamp t until an event with time stamp t or larger has been seen on signal s_2. Were we to use the Chandy and Misra approach, we would insert null events into s_2 to minimize the real-time delay of these stalls. However, we have real-time constraints at the `Device` actors that will not be met if we use this brute-force technique. Moreover, it is intuitively obvious that such a conservative technique is not necessary. Since the actors communicate only through their ports, there is no risk in processing events in the upper Clock-Device loop ahead of time stamps received on s_2. Our PTIDES technique formalizes this observation using causality analysis.

To make this example more concrete, we have in our lab prototype systems provided by Agilent that implement IEEE 1588. These platforms include a Linux host and simple timing-precise I/O hardware. Specifically, one of the facilities is a device driver API where the software can request that the hardware generate a digital clock edge (a voltage level change) at a specified time. After generating this level change, the hardware interrupts the processor, which resets the level to its original value. Our implementation of the `Device` actor takes input events as specification of when to produce these level changes. That is, it produces a rising edge at physical time equal to the model time of an input event. After receiving an input, it outputs an event with time stamp equal to the physical time at which the level is restored to its original value. Thus, its input time stamps must precede physical time, and its output events are guaranteed to follow physical time. This physical setup makes it easy to measure very precisely the real-time behavior of the system (oscilloscope probes on the digital I/O connectors tell it all).

The feedback loops around the two `Clock` and `Device` actors ensure that the Device does not get overwhelmed with requests for future level changes. It may not be able to buffer those requests, or it may have a finite buffer. Without the feedback loop, since the ports of the `Clock` actor have no real-time constraints, there would be nothing to keep it from producing output events much faster than real time.

This model is an abstraction of many realistic applications. For example, consider two networked computers controlling cameras pointing at the same scene from different angles. Precise time synchronization allows them to take sequences of pictures simultaneously. Merging two synchronous streams of pictures creates a 4D view for the scene (three physical dimensions and one time).

PTIDES programs are discrete-event models constructed as networks of actors, as in the example above. For each actor, we specify a physical host to execute the actor. We also designate a subset of the input ports to be *real-time ports*. Time-stamped events must be delivered to these ports before physical-time exceeds the time stamp. Each real-time port can optionally also specify a *setup time* τ, in which case it requires that each input event with time stamp t be received before physical time reaches $t - \tau$. A model is said

to be *deployable* if these constraints can be met for all real-time ports. Causality analysis can reveal whether a model is deployable, as discussed below in section III.

The key idea is that events only need to be processed in time-stamp order when they are causally related. We defined formal interfaces to actors that tells us when such causal relationships exist.

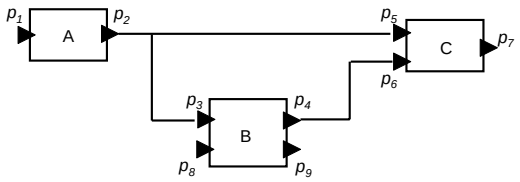

Fig. 2. A composition of actors.

III. Relevant Dependency

Model-time delays play a central role in the existence and uniqueness of discrete-event system behavior. Causality interfaces [16] provide a mechanism that allows us to analyze delay relationships among actors. In this section, we use causality interfaces to derive relevant dependencies among discrete events. Relative dependencies are the key to achieving out of order execution without disobeying the formal semantics of discrete-event specifications.

A. Causality Interfaces

The interface of actors contains ports on which actors receive or produce events. Each port is associated with a signal. A *causality interface* declares the dependency that output events have on input events. Formally, a causality interface for an actor a with input ports P_i and output ports P_o is a function:

$$\delta_a \colon P_i \times P_o \to D \tag{1}$$

where D is an ordered set with two binary operations $\oplus$ and $\otimes$ that are associative and distributive. That is,

$$\begin{aligned} &\forall d_1, d_2, d_3 \in D, \\ &(d_1 \oplus d_2) \oplus d_3 = d_1 \oplus (d_2 \oplus d_3) \\ &(d_1 \otimes d_2) \otimes d_3 = d_1 \otimes (d_2 \otimes d_3) \\ &d_1 \otimes (d_2 \oplus d_3) = (d_1 \otimes d_2) \oplus (d_1 \otimes d_3) \\ &(d_1 \oplus d_2) \otimes d_3 = (d_1 \otimes d_3) \oplus (d_2 \otimes d_3) \end{aligned} \tag{2}$$

In addition, $\oplus$ is commutative,

$$d_1 \oplus d_2 = d_2 \oplus d_1.$$

The $\otimes$ operator is for serial composition of ports, and the $\oplus$ operator is for parallel composition. The elements of D are called *dependencies*, and $\delta_a(p_1, p_2)$ denotes the dependency that port p_2 has on p_1.

For discrete-event models, $D = \mathbb{R}_0 \cup \{\infty\}$, $\oplus$ is the min function, and $\otimes$ is addition. With these definitions, D is a min-plus algebra [1]. Note that these operators are defined on model time.

Given an input port p_1 and an output port p_2 belonging to an actor a, $\delta_a(p_1, p_2)$ gives the model-time delay between input events at p_1 and resulting output events at p_2. Specifically, if $\delta_a(p_1, p_2) = d$, then any event e_2 that is produced at p_2 as a result of an event e_1 at p_1 will have time stamp $t_2 \geq t_1 + \delta$, where t_1 is the time stamp of e_1. For example, a `Delay` actor with a delay parameter d will produce an event with time stamp $t + d$ at its output p_2 given an event with time stamp t at its input p_1, so $\delta_{\texttt{Delay}}(p_1, p_2) = d$. Note that the causality interface gives the worst case (the smallest possible delay). An actor may produce an event e_2 with a larger time stamp, or may produce no event at all in response to e_1, and the actor still conforms with the causality interface.

A program is given as a composition of actors, by which we mean a set of actors and connectors linking their ports. Given a composition and the causality interface of each actor, we can determine the dependencies between any two ports in the composition by using $\otimes$ for serial composition and $\oplus$ for parallel composition. For example, to determine the dependencies for ports in the composition shown in figure 2, we need to determine the function:

$$\begin{aligned} &\delta \colon P \times P \to D \\ &\text{where } P = \{p_1, p_2, ...p_9\} \end{aligned} \tag{3}$$

We form a weighted, directed graph $G = \{P, E\}$, called the *dependency graph*, as shown in figure 3, where P is the set of ports in the composition. If p is an input port and p' is an output port, there is a edge in G between p and p' if p and p' belong to the same actor a and $\delta_a(p, p') < \infty$. In such a case, the weight of the edge is $\delta_a(p, p')$. If p is an output port and p' is an input port, there is an edge between p and p' if there is a connector between p and p'. In this case, the weight of the edge is 0. In all other cases, the weight of an edge would be ∞, but we do not show such edges. Note that this directed graph could by cyclic, and the classical requirement for a DE model to be executable is that the sum (or $\otimes$) of the edge weights in each cycle be greater than zero [16].

$\forall p, p' \in P$, to determine the value of $\delta(p, p')$, we need to consider all the paths between p and p'. We combine parallel paths using $\oplus$ and serial paths using $\otimes$. In particular, the weight of a path is the sum of the weights of the edges along the path ($\otimes$). The $\oplus$ operator is minimum, so $\delta(p, p')$ is the weight of the path from p to p' with the smallest weight. For example, $\delta(p_1, p_7)$ is calculated as:

$$\begin{aligned} \delta(p_1, p_7) =& min(ph_1, ph_2), \text{ where} \\ ph_1 =& \delta_A(p_1, p_2) + 0 + \delta_C(p_5, p_7), \\ ph_2 =& \delta_A(p_1, p_2) + 0 + \delta_B(p_3, p_4) + 0 + \delta_C(p_6, p_7) \end{aligned} \tag{4}$$

Note that paths with infinite weight in parallel with any path that is shown in our graph would have no effect, which is why we do not show such paths. If there is no path from a port p back to itself, then $\delta(p, p) = \infty$.

Note that these dependency values between ports do not tell the whole story. Consider the `Merge` actor in figure 1. It

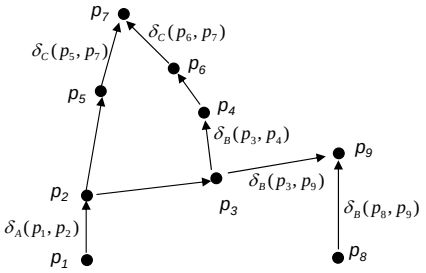

Fig. 3. A graph for computing the causality interface of a composition of actors.

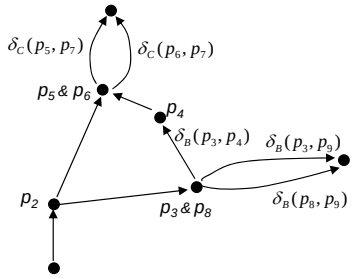

Fig. 4. The relevant dependency graph for the model in figure 2.

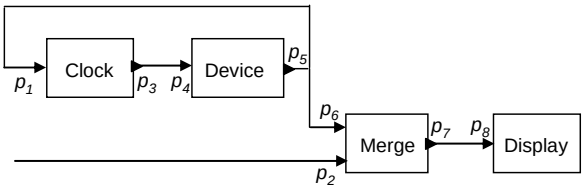

Fig. 5. The motivating example with names of ports.

has two input ports, but when we construct the dependency graph, we will find that there is no path between these ports. However, these ports have an important relationship, noted above. In particular, the `Merge` actor cannot react to an event at one port with time stamp t until it is sure it has seen all events at the other port with time stamp less than or equal to t. This fact is not captured in the dependencies. To capture it, we define *relevant dependencies*.

B. Relevant Dependency

Based on the causality interface of actors, the *relevant dependency* on any pair (p_1, p_2) of *input* ports specifies whether an event at p_1 will affect an output signal that may also depend on an event at p_2.

The relevant dependency between ports in a composition is calculated in a way similar to the dependency above, but we aggregate some of the ports into equivalence classes. Specifically, considering an individual actor a, two input ports p_1 and p_2 of a will be "equivalent" if there is an output port that depends on both. Formally, p_1 and p_2 are equivalent if

$$\exists p \in P_o, \text{ such that } \delta_a(p_1, p) < \infty \text{ and } \delta_a(p_2, p) < \infty,$$

where P_o is the set of output ports of a.

For example, in figure 2, assume that both input ports of actor C affect its output port, i.e. that $\delta_C(p_5, p_7) < \infty$ and $\delta_C(p_6, p_7) < \infty$. Then p_5 and p_6 are equivalent.

In addition, we assume that if any actor has state that is modified or used in reacting to events at more than one input port, then that state is explicitly treated as an output port. Thus, with the above definition, two input ports are equivalent if they are coupled by the same state variables of the actor. For example, in figure 2, port p_9 might represent the state of actor B. If both input ports p_3 and p_8 affect the state, then the dependencies are $\delta_B(p_3, p_9) = \delta_B(p_8, p_9) = 0$, and p_3 and p_8 are equivalent.

We next modify the dependency graph by aggregating ports that are equivalent to create a new graph that we call the *relevant dependency graph*. Consider the graph in figure 3. Suppose, as above, that p_5 and p_6 are equivalent and p_3 and p_8 are equivalent. Then the relevant dependency graph for the model in figure 2 becomes that shown in figure 4.

Note that in the relevant dependency graph, there is a path from p_8 to p_6 that was not present in the dependency graph. Thus, although events at p_8 do not affect events at p_6 (they have no dependency), there is nonetheless a relevant dependency because events at p_3 affect events at p_9 (which are also affected by events at p_8) and at p_6. These effects imply an ordering constraint in processing events at p_8 and p_6.

Below we will show that the relevant dependency induces a partial order on events that defines the constraints on the order in which we can process events.

The *relevant dependency* for a composition of actors is constructed as follows. Let Q be the set of equivalence classes of input ports in a composition. For example, $q_{3,8} = \{p_3, p_8\} \in Q$ in figure 2. Then, the relevant dependency is a function of the form

$$d \colon Q \times Q \to D$$

where for example in figure 2,

$$Q = \{q_1, q_{3,8}, q_{5,6}\} = \{\{p_1\}, \{p_3, p_8\}, \{p_5, p_6\}\}.$$

Similar to ordinary dependencies, relevant dependencies are calculated by examining weights of the relevant dependency graph. $\forall q, q' \in Q$, to determine the value of $d(q, q')$, we need to consider all the paths between q and q'. We again combine parallel paths using $\oplus$ and serial paths using $\otimes$. In particular, the weight of a path is the sum of the weights of the edges along the path ($\otimes$). The $\oplus$ operator is minimum, so $d(q, q')$ is the weight of the path from q to q' with the smallest weight.

When the relevant dependency is $d(q, q') = r, r \in \mathbb{R}_0$, this means that any event with time stamp t at any port in q' can be processed when all events at ports in q are known up to time stamp $t - r$.

When the relevant dependency is $d(q, q') = \infty$, this means that events at any port in q' can be processed without knowing anything about events at any port in q.

Figure 5 shows a portion of the model in figure 1 and names each port. The causality interface for each actor in the model

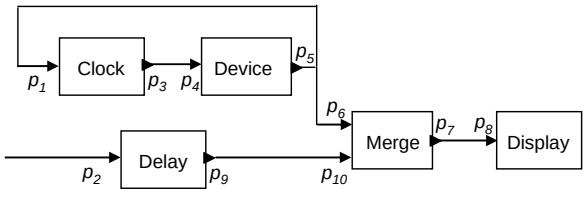

Fig. 6. The motivating example with a delay actor.

is:

$$\begin{aligned} \delta_{\texttt{Clock}}(p_1, p_3) &= 0 \\ \delta_{\texttt{Device}}(p_4, p_5) &= d_0 \\ \delta_{\texttt{Merge}}(p_6, p_7) &= 0 \\ \delta_{\texttt{Merge}}(p_2, p_7) &= 0 \end{aligned} \tag{5}$$

where $d_0 > 0$ is the response delay of the digital output device (i.e. the minimum model-time delay across the `Device` actor).

Recall that p_4 is a real-time port. It is easy to check that the relevant dependency in this composition $d(q_{2,6}, q_4)$ is ∞, where using the same notation as above, $q_{2,6} = \{p_2, p_6\}$ and $q_4 = \{p_4\}$. This means that events at p_4 can be processed without knowing anything about events at p_2 or p_6. This is precisely the result we were after. It means that the arrival events over the network into p_2 need not interfere with meeting real-time constraints at p_4. This would not be achieved with a Chandy and Misra policy. And unlike optimistic policies, there will never be any need to backtrack.

If we modify the model in figure 5 by adding a `Delay` actor with a delay parameter d, we get a new model as shown in figure 6. The relevant dependency becomes $d(q_2, q_{6,10}) = d$. Now an event with time stamp t at p_6 can be processed if all events with time stamps smaller than or equal to $t - d$ at p_2 have been processed. With the same assumptions as discussed in section II (an event with model time t is produced at physical time t by the `Device` process, and the network delay is bounded by C), at physical time $t - d + C$ we are sure that we have seen all events with time stamps smaller than $t - d$ at p_2. Hence, an event e at p_6 with time stamp t can be processed at physical time $t - d + C$ or later. Note that although the `Delay` actor has no real-time properties at all (it simply manipulates model time), its presence loosens the constraints on the execution.

IV. Execution Based on the Relevant Order

What we gain from the dependency analysis is that we can specify which events can be processed out of order, and which events have to be processed in order.

A. Relevant Order

We define the *relevant order* as follows. Suppose e_1 is an event with time stamp t_1 at a port in q_1 and e_2 is an event with time stamp t_2 at a port in q_2. Then

$$e_1 <_r e_2 \Leftrightarrow t_1 + d(q_1, q_2) < t_2.$$

We use notation $<_r$ for the relevant order. It is straightforward to show that this is a partial order on events. We interpret $e_1 <_r e_2$ to mean that e_1 must be processed before e_2. Two events e_1 and e_2 are not comparable, denoted as $e_1 ||_r e_2$, if neither $e_1 <_r e_2$, nor $e_2 <_r e_1$. If $e_1 ||_r e_2$, then e_1, e_2 can be processed in any order. What we mean by "processed" is that the actor that is the destination of the event is *fired*, meaning that it is executed and allowed to react to the event.

B. Execution Strategies

We now design execution strategies based on the relevant order to enable out of order execution without hurting determinism. One execution algorithm may work as follows:

1) Start with E, a set of events in the event queue.
2) Choose $r \subset E$, s.t. each event in r is *minimal* in E.
3) Process events in r, which may produce a set of new events E'.
4) Update E to $(E \setminus r) \cup E'$.
5) Go to 2.

An event e is *minimal* in E if $\forall e' \in E, e <_r e',$ or $e ||_r e'$.

This strategy, however, fails when there are events coming over the network. The pitfall here is that it assumes all the events that have been generated in the system are in E, but in a distributed system with network delays, this is not true.

An input port is called *a network port* if it receives events from external hardware. Here we use the word network in a loose sense, which covers both communication network and external I/O. For the model shown in figure 5, p_2 is a network port as it receives events from another computer. Both p_1 and p_6 are also network ports as they receive events from an external device. Let P_d denote the set of network ports in a composition.

For network ports, we assume that events that are received on those ports have time stamps that are related to physical time. Specifically, let Δ_p be a non-negative real number associated with network port p. Then we assume that an event with time stamp t on port p will be received at real time no later than $t + \Delta_p$. We call Δ_p the network delay associated with port p.

For any input port p, let $Q(p)$ denote the equivalence class that contains p. An event e with time stamp t at a port p is said to be Δ-*minimal* if e is minimal in E, and the current physical time is no less than $T = \max_{p' \in P_d}\{t - d(Q(p'), Q(p)) + \Delta_{p'}\}$. That is, an event is Δ-minimal if it is minimal in E and we are assured that we have seen all events that are less than it in the relevant order.

The execution algorithm becomes:

1) Start with E, a set of events in the event queue.
2) Choose $r \subset E$, s.t. each event in r is Δ-minimal in E.
3) Process events in r, which may produce a set of new events E'.
4) Update E to $(E \setminus r) \cup E'$.
5) Go to 2.

If the clocks in the distributed systems are not perfectly synchronized, we also need to take into account the time synchronization error to the estimated physical time T. In particular, if the difference of the clock time between any two nodes in the systems is bounded by ξ, we need to wait until the current physical time is $T + \xi$ to make sure e is minimal.

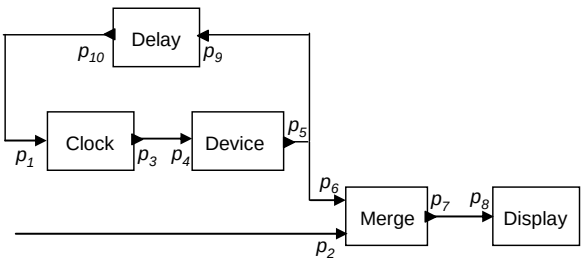

Fig. 7. Model in figure 5 with a delay actor.

V. Towards Deployability Analysis

A key requirement for preserving runtime determinism of PTIDES programs, as we mentioned in section II, is that each event e with model time t at a real-time port must be received before the physical time exceeds $t - \tau$, where τ is the *setup time* of the real-time port. We call a PTIDES program *deployable* if this requirement can be guaranteed. We are interested in statically checking deployability for a given PTIDES program and a system characteristic such as communication delay and execution time bounds. In this section, we describe a partial solution.

When the execution time is negligible comparing to the network delays and setup time, deployability checking becomes straightforward by using the relevant order of events under our new execution strategy. Let P_n denote the set of network ports in a composition and $Q(p)$ denote the equivalence class containing port p. For a real-time port p, if there is a port $p' \in P_n$ such that the relevant dependency $d(Q(p'), Q(p)) < \infty$, then there are order constraints on the events at p' and p. That is, an event received at p' with model time stamp $t' < t - d(Q(p'), Q(p))$ is less than (in the relevant order, $<_r$) an event e with model time stamp t at p. Hence the event e cannot be processed until we are sure that all events at p' with time stamp less than $t - d(Q(p'), Q(p))$ have been received.

For the worst case, when the network delay achieves its upper bound, we need to wait until the physical time reaches $T_p(p', t) = t - d(Q(p'), Q(p)) + \Delta_{p'}$. Recall $\Delta_{p'}$ is the network delay at port p', i.e. an event with model time t' can be received at p' as late as physical time $t' + \Delta_{p'}$. If for all $p' \in P_n$, $T_p(p', t) \leq t - \tau$, i.e.,

$$\Delta_{p'} + \tau \leq d(Q(p'), Q(p)),$$

then we call the real-time port p *deployable*. We can guarantee that an event e at port p is processed before the physical time reaches the model time of e. A system is deployable if all its real-time ports are deployable.

As an example, consider again the system shown in figure 5, where $P_n = \{p_1, p_2, p_6\}$ is the set of network ports and $P_r = \{p_4\}$ is the set of real-time ports. Assume the set up time for p_3 is 0. Assume the network delay Δ_{p_2} is C, and $\Delta_{p_1} = \Delta_{p_6} = C'$. We can perform the following analysis.

The real time port p_4 only has relevant dependency with the network port p_1. It has no relevant dependency with the other two ports in P_n, i.e. $d(Q(p_2), Q(p_4)) = d(Q(p_6), Q(p_4)) = \infty$. The dependency $d(Q(p_1), Q(p_4)) = 0$, which is less than the network delay of port p_1. This indicates that the system is not deployable. To see this, consider an event e with model time t that can be received at physical time $t + C'$ at port p_1. Assume $t_0 + n\alpha \leq t \leq t_0 + (n+1)\alpha$, where α is the period of the clock, and t_0 is the time of the first event) of the clock. As a result of e, the `Clock` actor produces an output event e' with model time $t_0 + (n+1)\alpha$ to p_4 at physical time $t + C'$. Since the model time t of the event e may be arbitrarily close to $t_0 + (n+1)\alpha$, the event e' at p_4 cannot be processed before the physical time reaches its model time.

If we change the system by adding a `Delay` actor with a delay parameter $d \geq C'$ as shown in figure 7, the system becomes deployable. In particular, the distributed ports for this new system are p_2, p_6 and p_9. The real time port p_4 now only has relevant dependency with p_{10}. Note that the relevant dependency $d(Q(p_{10}), Q(p_4)) = d$. In order for the system to be deployable, $d \geq C'$ is sufficient.

The analysis discussed above is the beginning of what allows us to statically check whether a PTIDES specification is feasible to be deployed over a network of nodes. A full analysis, when the execution time is not negligible, requires integrating relevant dependency analysis with real-time scheduling and worse case execution time analysis on individual nodes. This is an interesting future direction.

VI. Conclusions

This paper describes the use of discrete event models as programming specifications for time-synchronized distributed real-time systems. We call the technique PTIDES, Programming Temporally Integrated Distributed Embedded Systems. We limit the relationship of model time to physical time to only those circumstances where this relationship is needed, and provide an execution model that permits out of order processing of events without sacrificing determinacy and without requiring backtracking. We give a formal foundation based on the concepts of relevant dependency and relevant order. The resulting foundation is particularly valuable in time-synchronized distributed real-time systems, since we can take advantage of the globally consistent notion of time as a coordination channel. Based on relevant orders, we can statically analyze whether a given model is deployable on a network of nodes, given we know the network delays.

We are building PTIDES programming interface in Ptolemy II [7] and a runtime system on Agilent prototype devices with IEEE 1588 time synchronization. PTIDES is implemented based on the classical discrete event (DE) domain in Ptolemy II, extending it with real-time semantics. We are leveraging ongoing work in the Ptolemy Project on code generation.

VII. Acknowledgements

This paper describes work that is part of the Ptolemy project, which is supported by the National Science Foundation (NSF award number CCR-00225610), and Chess (the Center for Hybrid and Embedded Software Systems), which receives support from NSF, the State of California Micro Program, and the following companies: Agilent, DGIST, General Motors, Hewlett Packard, Microsoft, and Toyota.

REFERENCES

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

[2] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

[3] G. Berry. The effectiveness of synchronous languages for the development of safety-critical systems. White paper, Esterel Technologies, 2003.

[4] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

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

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

[7] 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(1):127–144, Jan. 2003.

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

[9] D. Gay, P. Levis, R. v. Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *Programming Language Design and Implementation (PLDI)*, 2003.

[10] P. L. Guernic, T. Gauthier, M. L. Borgne, and C. L. Maire. Programming real-time applications with signal. *Proceedings of the IEEE*, 79(9), 1991.

[11] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1319, 1991.

[12] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. In *EMSOFT 2001*, volume LNCS 2211, Tahoe City, CA, 2001. Springer-Verlag.

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

[14] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.

[15] L. Lamport, R. Shostak, and M. Pease. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[16] E. A. Lee, H. Zheng, and Y. Zhou. Causality interfaces and compositional causality analysis. *Invited paper in Foundations of Interface Technologies, Satellite to CONCUR 2005*, August 2005.

[17] C. L. Liu and J. W. Leyland. Scheduling algorithms for multiprogramming in a hard real time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[18] J. Liu and E. A. Lee. Timed multitasking for real-time embedded software. *IEEE Control Systems Magazine*, pages 65–75, 2003.

[19] D. Mills. A brief history of ntp time: confessions of an internet timekeeper. *ACM Computer Communications Review 33*, April 2003.

[20] H. Wang, L. Yip, D. Maniezzo, J. Chen, R. Hudson, J. Elson, and K. Yao. A wireless time-synchronized COTS sensor platform part ii: applications to beamforming. In *Proc. IEEE CAS Workshop on Wireless Communications and Networking*, 2002.

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