in Proceedings of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 07) , Bellevue, WA, United States.

A Programming Model for Time-Synchronized Distributed Real-Time Systems

Yang Zhao
EECS Department
UC Berkeley
Berkeley, CA 94720 USA
ellen_zh@eecs.berkeley.edu

Jie Liu
Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA
liuj@microsoft.com

Edward A. Lee
EECS Department
UC Berkeley
Berkeley, CA 94720 USA
eal@eecs.berkeley.edu

Abstract

Discrete-event (DE) models are formal system specifications that have analyzable deterministic behaviors. Using a global, consistent notion of time, DE components communicate via time-stamped events. DE models have primarily been used in performance modeling and simulation, where time stamps are a modeling property bearing no relationship to real time during execution of the model. In this paper, we extend DE models with the capability of relating certain events to physical time. We propose a programming model, called PTIDES (Programming Temporally Integrated Distributed Embedded Systems), which has DE semantics, but with carefully chosen relations between model time and real time. Key to making this model effective is to ensure that constraints that guarantee determinacy in the semantics are preserved at runtime. To accomplish this, we give a distributed execution strategy that obeys DE semantics without the penalty of totally ordered executions based on time stamps. Our technique relies on having a distributed common notion of time, known to some precision. Based on causality analysis of DE models, we define relevant dependency *and* relevant orders *to enable out-of-order execution without compromising determinism and without requiring backtracking.*

1 Introduction

Distributed embedded systems are computer-based systems where multiple computers are connected on a network. Typically, each computer is connected to sensors, actuators or human-computer interfaces. Applications include manufacturing, instrumentation, surveillance, multi-vehicle control, avionics systems, automotive systems and scientific experiments. Since each computer interacts with physical processes, the passage of time becomes a central feature; it is this key constraint that distinguishes these systems from distributed computing in general.

In addition to interacting over a communication network, the nodes in a distributed embedded system interact through the physical world. Driving an actuator at one node, for example, may affect the data sensed at another node. Moreover, actuation may need to be orchestrated across nodes. The required precision of that orchestration, of course, depends on the application. Robotic applications, e.g. in manufacturing, may require precisions on the order of milliseconds. Instrumentation, where stimuli are generated and responses are measured, may require precisions on the order of nanoseconds or even higher. The question we address in this paper is how to construct the distributed software for such systems.

General-purpose distributed software is dominated by distributed object-oriented programming [32] using frameworks such as CORBA, SOAP, DCOM, EJB, and XML Web Services. Some extensions of these frameworks, such as ACE/TAO [31], support real-time scheduling concepts, and have caught on in certain communities (such as avionics) [30]. These technologies are viewed as being too heavyweight for many embedded applications such as industrial control, where software may be written in special purpose languages (e.g. based on the International Electrotechnical Commission's IEC 61131) and executed on special purpose hardware called Programmable Logic Controllers (PLCs). Extensions of these techniques to distributed control systems (e.g. IEC 61499), have not proved satisfactory, because of the non-determinism of implementation. That is, the same standard-compliant application running in two different implementations of the runtime environment may result in different behaviors [7].

Our approach to the nondeterminism challenge in constructing distributed real-time system is to rely on network time synchronization [20], where the computing nodes on the network share a common notion of time to a known precision. This has the potential for being lightweight and delivering repeatable and predictable behaviors at a variety of timing precisions.

Network time synchronization is available on a variety

of platforms, including standard computers on the Internet (e.g. NTP [29]), time-triggered buses such as TTA or FlexRay [21], TCP/IP over Ethernet (e.g. IEEE 1588), and wireless networks (e.g. RBS [11]). Implementations of IEEE 1588 have demonstrated time synchronization as precise as tens of nanoseconds over networks that stretch over hundreds of meters, more than adequate for many manufacturing and instrumentation 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 or client/server mechanisms.

Our approach in this paper builds on discrete-event (DE) modeling techniques [6, 23, 34]. DE models are concurrent compositions of components that interact via *events*. An event is a time-stamped value, where time is "logical time" or "modeling time" [22]. Correct execution of such models requires only that the ordering of time stamps be respected. DE is usually a *simulation* technology (e.g. in hardware description languages such as Verilog and VHDL and network modeling languages such as OPNET Modeler[1] and Ns-2[2]). When DE models are executed on distributed platforms, the objective is usually to accelerate simulation, not to implement distributed real-time systems [6, 12, 34].

We call our programming model PTIDES (pronounced "tides"), for Programming Temporally Integrated Distributed Embedded Systems. In our approach, DE is not a simulation technology, but rather application specification language, which serves as a semantic basis for obtaining determinism in distributed real-time systems. Applications are given as distributed DE models, where for certain events, their modeling time is mapped to physical time. For example, a programmer may specify that an actuator must produce a physical output at the time determined by the time stamp of an event sent to the actuator. When these models are executed in a runtime environment that ensures DE semantics, we know that the applications will have deterministic behaviors regardless of the actual implementations.

Preserving DE semantics at runtime can be challenging, since the global, consistent notion of time may lead to a total ordering of execution in a distributed system, an unnecessary waste of resources. PTIDES takes an event-driven execution strategy. Unlike many hard real-time distributed systems that depend on domain specific network architectures, our only assumption of communication behavior is that it delivers packets reliably with a known bounded delay. We divide our execution strategies into two layers: global coordination, and local resource scheduling. When receiving an event from the network, the global coordination layer determines whether the event can be processed immediately or it has to wait for other potentially proceeding events. Once it is sure that the current event can be processed according to DE semantics, it hands the event over to local resource scheduler, which may use existing real-time scheduling algorithms, such as earliest deadline first (EDF) to prioritize the processing of all pending events. This paper only focuses on the global coordination layer, which is key to achieving DE semantics in distributed systems. We leverage and improve on distributed DE techniques to relax constraints on execution. In particular, we define a partial order called the *relevant order* that can be statically checked to enable the global coordination layer to release received events out of their time stamp order while preserving DE semantics and without requiring backtracking. This out-of-order execution also loosens some constraints for the local resource schedulers.

This paper is organized as follows. Section 2 discusses related work. Section 3 motivates our programming model using a networked camera application. Section 4 develops the *relevant dependency* concept using causality interfaces [25], 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 5, and its implementaion is in section 6. Future work is discussed in section 7.

2 Related Work

Synchronous languages [4] have been effectively applied to safety-critical embedded systems [5]. These languages (which include Esterel, SCADE, Lustre, Signal, etc.) provide deterministic concurrent semantics, but at the expense of tight coordination that makes distributed implementation difficult.

Another important innovation is the development of time-triggered languages and the concept of logical execution time [16]. One example is Simulink with Real-Time Workshop (RTW), from The MathWorks, which is widely used for designing embedded control systems in applications such as automotive electronics. Simulink with RTW leverages an underlying preemptive priority-driven multitasking operating system to deliver real-time behavior based on rate-monotonic scheduling [26]. A related approach is Giotto [16], which introduces additional latency but delivers better schedulability analysis [28]. While Giotto and Simulink/RTW are intended primarily for periodic real-time tasks, extensions that support less periodic behavior have emerged [27, 14]. Our approach in this paper is closest to timed multitasking (TM)[27], but goes a step further in embracing DE semantics and relating model time to real time at sensor and actuator interactions rather than through log-

[1] http://opnet.com/products/modeler/home.html
[2] http://www.isi.edu/nsnam/ns

ical execution time. Distributed implementations of time-triggered languages [17] usually rely on TDMA network. Our approach imposes fewer requirements on the network and leverages distributed DE simulation concepts.

In system modeling, many paradigms incorporate the notion of time in their semantics. Examples include timed automata [2], hybrid automata [15], and timed Petri nets [10]. Although these provide useful modeling frameworks, we are not aware of realizations that bind logical time to real time and provide deterministic real-time execution.

Existing methods for addressing real-time computation typically deal with a portion of the problem of constructing and executing real-time programs. Real-time operating systems (RTOSs) provide mechanisms for prioritizing tasks and triggering computations in response to timer or event interrupts. Time-triggered networking techniques such as the Time Triggered Architecture (TTA) provide deterministic sharing of networking resources and insulation from faults. Network time synchronization protocols such as NTP and IEEE 1588 provide a common time base across computers on a network [20]. All of these technologies, however, are used with relatively conventional concurrency models (threads and processes) and conventional programming languages. This paper elevates timing and distribution to the level of the programmer's model, so that applications are built by directly expressing timing and distribution properties [24].

Interface-based design [1], especially real-time and resource interfaces [18, 33, 8], also shows considerable promise for helping with real-time embedded software. Our analysis in this paper heavily leverages a special case of these called causality interfaces [25].

Our work builds on distributed discrete-event simulation, which has a rich history [13]. So-called "conservative" techniques advance model time to t only when each node can be assured that it has seen all events time stamped t or earlier. In the well-known Chandy and Misra technique [9], extra (null) 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 [19]. Such optimistic techniques will also not work in our context, since backtracking physical interactions is not possible.

Our approach 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 relevant dependency analysis.

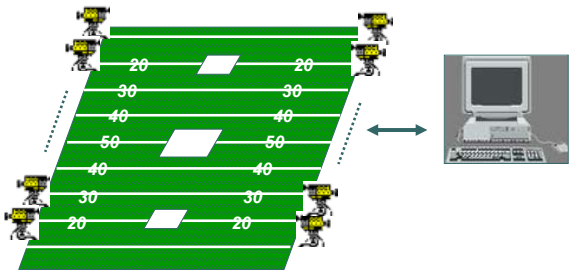


Figure 1. Networked camera application.

3 Distributed Camera Application

We use a camera network as a motivating scenario and a running example in later discussions. Throughout the paper, we use t for model time and τ for physical time.

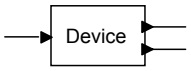
Consider N cameras connected via Ethernet are distributed over a football field as shown in figure 1. Suppose that the clocks at all the cameras and the central computer are precisely synchronized. All the cameras have computer-controlled picture and zoom capabilities. Each camera only has a partial view of the field. We can control a camera to take a picture or zoom precisely at some physical time. Precise time synchronization allows cameras to take sequences of pictures simultaneously. The images produced by each camera are time stamped and transferred over the network to the central computer, where the images get processed to produce a composite view of the field or a sequence of views for some interesting moment. A user sitting in front of the central computer may issue commands to the cameras to zoom or change the frequency at which images are taken. Suppose that zooming takes time κ to stabilize, and during this period of time no picture should be taken. Given that the commands controlling the cameras are transmitted over the network with a bounded delay, the challenges here are how to coordinate the zooming and picture taking actions properly on each camera so that the retrieved images are precisely synchronized.

This application is inspired by the "eye vision" project[3] at CMU and CBS Television. However, rather than focusing on the challenges in real-time image processing and control, we consider how to program the whole system at a high level and how to realize the timing relations in the system. The principles we develop are general enough to apply in many scenarios that require distributed time-coordinated physical actions.

Cameras in this application are both sensors and actuators. We need to generate precisely timed physical actions, like picture taking and zooming, at each camera, and the cameras respond with time stamped images. Taking an actor-oriented approach for building DE systems [23], we

[3] http://www.ri.cmu.edu/events/sb35/tksuperbowl.html

model a camera as an *actor* that has one input port and two output ports, depicted graphically as follows:



This actor is a software component that wraps the interaction with the camera driver. We assume that it does not communicate with any other software component except via its ports. (This assumption is made for functional level specification, which focuses on input/output data and timing properties; scheduling and resource sharing between actors are delayed to the implementation and execution stage.) 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 time stamp of an event specifies when an action should be taken, and the value of the event dictates what kind of action (zooming level or shutter speed) should be taken. Obviously, in order to give the actuator some leeway to react, it needs to receive an input event with time stamp t at some physical time $\tau \leq t$.

The first output port produces a time-stamped value for each input event, where the time stamp is strictly greater than that of the input event, to indicate the physical action has completed. The second output port produces the time-stamped image and sends it to the central computer. Each output event with time stamp t' is produced at some physical time $\tau' \geq t'$. These inequalities provide the basic relations between model time and physical time.

Figure 2 shows a distributed DE model to be executed on the cameras and the central computer platform. The dashed boxes divide the model into two parts, the top one to be executed on each camera and the bottom one to be executed on the central computer. The parts communicate via signal s_1 and s_2. We assume that events in these signals are sent over the network as time-stamped values.

The Command actor is a software component that wraps interactions with the user input device. When a user input comes in, the Command actor checks with its synchronized clock for the current time, uses the returned time value to time stamp the input message and sends the time stamped message to all the cameras. The right part of the model on the central computer processes the images taken at each camera and displays the result.

The Clock actor produces time-stamped outputs where the time stamp is some integer multiple of a period p. The time stamps are used to control when the camera takes pictures. The period can be different for each clock and can be changed during run time upon receiving an input on the second input port. If there is an event with value v and time stamp t at the second input, the clock actor will scale its period from p to $p' = p * v$ and produce an output with time

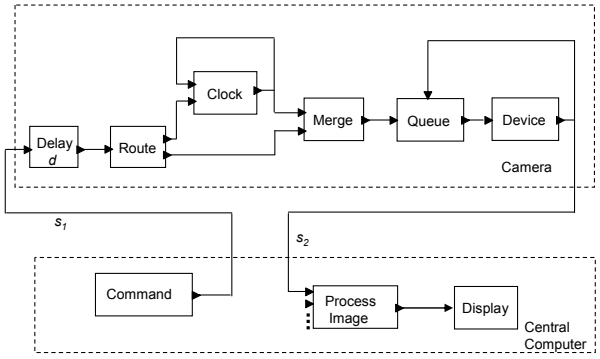


Figure 2. Specification of the Networked camera application.

stamp $t_0 + np'$ where t_0 is the time stamp of the last output and n is the smallest integer so that $t_0 + np' \geq t$. The feedback loop around the clock actor is used to trigger the next output, and we assume there is an initial event on the first input at the beginning.

The Delay actor with a delay parameter d will produce an event with time stamp $t + d$ at its output given an event with time stamp t at its input.

Two kinds of user commands are received at each camera, the *change frequency* command and the *adjust zoom* command. The Router actor separates these events and sends the *change frequency* events to the Clock actor and the *adjust zoom* events to the Merge actor.

The Merge actor merges the events on the two input ports in chronological order. It gives priority to the second input port if input events have identical time stamps. That is, we give higher priority to the user to control a camera.

The Queue actor buffers its input event until an event is received at the trigger port, which is the one at the top of the actor. The Device actor sends a trigger event to the Queue actor when a physical action is done, and we assume there is an initial event on the trigger port at the beginning. The feedback loop around the Queue and Device actor ensures that the Device does not get overwhelmed with future requests. It may not be able to buffer those requests, or it may have a finite buffer.

In the above discussion, the time stamps are values of model time. Some actors in the model bind model time to physical time. The Command actor binds model time to physical time by producing an event with model time corresponding to the physical time when the user input happens. The Device actor binds model time to physical time by producing some physical action at the real-time corresponding to the model time of each input event. The Device actor also imposes real-time constraints on the model. The input events must be made available for the Device actor

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. We limit the relationship of model time to physical time to only those circumstances where this relationship is needed. For other actors in the model, there is no real-time constraint and model time is used to define execution semantics.

As shown in this example, PTIDES programs are discrete-event models constructed as networks of actors. 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*, which imposes the constraint that time-stamped events must be delivered to these ports before physical-time exceeds the time stamp. Each real-time port can optionally specify a *setup time* σ, in which case it requires that each input event with time stamp t be received before physical time reaches $t - \sigma$.

We view the model shown in figure 2 as a representative scenario for many distributed embedded applications. For example, the Command actor can be viewed as an example of sensor components, the Device actor is an example of actuator components, and other actors in between are the control or computation part. The problems discussed in this paper are common to many distributed sensing and actuation systems, such as manufacturing, instrumentation, surveillance, and scientific experiments.

How to build a run-time environment to execute the distributed model shown in figure 2 to deliver the correct behavior and meet the real-time constrains is a challenging problem. A first-come-first-serve strategy cannot preserve deterministic DE semantics, since the network may alter the order that events are delivered. A brute-force implementation of a conservative distributed DE execution of this model would stall execution in a camera at some time stamp t until an event with time stamp t or larger has been seen on signal s_1. Were we to use the Chandy and Misra approach [9], we would insert null events into s_1 to minimize the real-time delay of these stalls. However, as we will show later, this brute-force technique will unnecessarily postpone the release time of these events even when these events can be safely processed. The so-called "optimistic" techniques for distributed DE execution will also not work in our context. Optimistic approaches perform speculative execution and backtrack if and when the speculation was incorrect [19]. Since we have physical interactions in the system, backtracking is not possible.

In section 5, we describe a new coordination strategy that permits out-of-order releasing of events without sacrificing determinacy and without requiring backtracking. This will give underlying resource scheduling layer looser end-to-end delay constraints to work with, thus improve schedulability. The key idea is that events only need to be processed in time-stamp order when they are causally related.

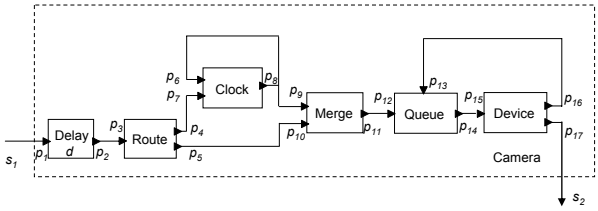


Figure 3. The program on the camera.

We define formal interfaces to actors that tells us when such causal relationships exist in the next section.

4 Relevant Dependency

Model-time delays play a central role in assuring the existence and uniqueness of discrete-event system behavior. We achieve correct, out-of-order execution by statically analyzing the causality relations among events. Causality interfaces [25] provide a mechanism that allows us to specify and reason about delay relationships among actors. In this section, we use causality interfaces to derive relevant dependencies among discrete events, which is the key to achieving out-of-order execution without disobeying the formal semantics of discrete-event specifications.

4.1 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 (in fact an algebra) 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 [3]. Note that these operators are completely 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 minimum model-time delay between input events at p_1 and resulting output events at p_2. Intuitively, if $\delta_a(p_1, p_2) = d$, then any event $e_2 = (t_2, v_2)$ that is produced at p_2 as a result of an event $e_1 = (t_1, v_1)$ at p_1 will satisfy $t_2 \geq t_1 + d$. 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 captures the smallest possible delay, i.e. the fastest response time. 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.

When a program is given as a composition of actors, i.e. a set of actors and connectors linking their ports, 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, figure 3 shows the program running on each camera and names the ports. To determine the dependencies for ports in the model, we compute the function:

$$\begin{aligned} &\delta \colon P \times P \to D \\ &\text{where } P = \{p_1, p_2, ...p_{17}\} \end{aligned} \tag{3}$$

This forms a weighted, directed graph $G = \{P, E\}$, called the *dependency graph*, as shown in figure 4, 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 [25].

$\forall p, p' \in P$, to compute 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 weight of the shortest weighted path from p to p'. For

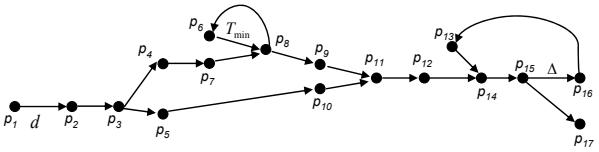


Figure 4. A graph for computing the causality interface of a composition of actors.

example, $\delta(p_1, p_{11})$ is calculated as:

$$\begin{aligned} \delta(p_1, p_{11}) =& min(ph_1, ph_2, ph_3), \text{ where} \\ ph_1 =& \delta_{Delay}(p_1, p_2) + 0 + \delta_{Router}(p_3, p_4) + 0 + \\ & \delta_{Clock}(p_7, p_8) + 0 + \delta_{Merge}(p_9, p_{11}), \\ ph_2 =& \delta_{Delay}(p_1, p_2) + 0 + \delta_{Router}(p_3, p_5) + 0 + \\ & \delta_{Merge}(p_{10}, p_{11}), \\ ph_3 =& \delta_{Delay}(p_1, p_2) + 0 + \delta_{Router}(p_3, p_4) + 0 + \\ & \delta_{Clock}(p_7, p_8) + 0 + \delta_{Clock}(p_6, p_8) + 0 + \\ & \delta_{Merge}(p_9, p_{11}) \end{aligned} \tag{4}$$

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 $\delta(p, p) = \infty$.

Causality interfaces form a powerful tool to analyze determinism in discrete-event systems, but these dependency values between ports do not tell the whole story. Consider the `Merge` actor in figure 3, with two input ports. When we construct the dependency graph, it is easy to find that there is no path between these ports. But, these ports are not completely independent. In fact, 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*.

4.2 Relevant Dependency

Based on the causality interfaces 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

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

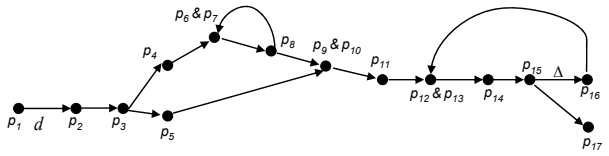


Figure 5. The relevant dependency graph for the model in figure 3.

where P_o is the set of output ports of a.

For example, in figure 3, both input ports of the Merge actor affect its output port, i.e. $\delta_{Merge}(p_9, p_{11}) < \infty$ and $\delta_{Merge}(p_{10}, p_{11}) < \infty$. Thus p_9 and p_{10} 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 treated as a hidden output port. Thus, with the above definition, two input ports are equivalent if they are coupled by the same state variables of the actor.

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 4. Suppose, as above, that p_6 and p_7 are equivalent, p_9 and p_{10} are equivalent and p_{12} and p_{13} are equivalent. Then the relevant dependency graph for the model in figure 3 becomes that shown in figure 5.

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, $q_{6,7} = \{p_6, p_7\} \in Q$ in figure 4. Then, the relevant dependency is a function of the form:

$$d\colon Q \times Q \to D. \tag{5}$$

For instance, in figure 4, we have:

$$\begin{aligned} Q =& \{q_1, q_3, q_{6,7}, q_{9,10}, q_{12,13}, q_{15}\} \\ =& \{\{p_1\}, \{p_3\}, \{p_6, p_7\}, \{p_9, p_{10}\}, \{p_{12}, p_{13}\}, \{p_{15}\}\}. \end{aligned} \tag{6}$$

Similar to ordinary dependencies, relevant dependencies are calculated by examining weights of the relevant dependency graph. $\forall q, q' \in Q$, to compute 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 $\otimes$. So $d(q, q')$ is the weight of the weighted shortest path from q to q' in the relevant dependency graph.

Intuitively, relevant dependency indicates how much we can advance time at a port without knowing all the events on the other ports in a composition. A relevant dependency $d(q, q') = r$ for some $r \in \mathbb{R}_0$, means that any event with time stamp t at ports in q' can be processed when all events at ports in q are known up to time stamp $t - r$. In particular, a relevant dependency $d(q, q') = \infty$ indicates that events at ports in q' can be processed without knowing anything about events at ports in q.

Consider the model running at each camera as shown in figure 3. The causality interface for each actor in the model is:

$$\begin{aligned} &\delta_{Delay}(p_1, p_2) = d, \\ &\delta_{Router}(p_3, p_4) = 0, \;\; \delta_{Router}(p_3, p_5) = 0, \\ &\delta_{Clock}(p_6, p_8) = T_{min}, \;\; \delta_{Clock}(p_7, p_8) = 0, \\ &\delta_{Merge}(p_9, p_{11}) = 0, \;\; \delta_{Merge}(p_{10}, p_{11}) = 0, \\ &\delta_{Queue}(p_{12}, p_{14}) = 0, \;\; \delta_{Queue}(p_{13}, p_{14}) = 0, \\ &\delta_{Device}(p_{15}, p_{16}) = \Delta, \;\; \delta_{Device}(p_{15}, p_{17}) = 0 \end{aligned} \tag{7}$$

where T_{min} is the minimum time interval between two consecutive picture taking actions at the camera, and $\Delta > 0$ is the response delay of the digital output device (i.e. the minimum model-time delay across the input and the first output of the `Device` actor).

Based on the dependencies specified for each actor, It is easy to check the relevant dependency in this composition. As an example, the relevant dependency $d(q_1, q_{9,10})$ is d, where using the same notation as above, $q_4 = \{p_4\}$ and $q_{9,10} = \{p_9, p_{10}\}$. This means that any event with time stamp t at port p_9 can be processed when all events at port p_1 are known up to time stamp $t - d$. Assume the network delay is bounded by d', at physical time $\tau = t - d + d'$ we are sure that we have seen all events with time stamps smaller than $t - d$ at p_1. Hence, an event e at p_9 with time stamp t can be processed at physical time τ 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. By choosing d properly, i.e. $d > d'$, we can deliver e to p_{15} before physical time reaches t and thus satisfy the actuation constraint at p_{15}. This is precisely the result we were after. It would not be achieved with a Chandy and Misra policy. And unlike optimistic policies, there will never be any need to backtrack.

5 Execution Based on the Relevant Order

What we gain from the dependency analysis is that we can process certain events out of their linear chronological order. In fact, we effectively defined a partial ordering of discrete events.

5.1 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 event is released to the actor that reacts to the event.

5.2 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 3, p_1 is a network port as it receives events from another computer. But, p_{13} is also a network port as it receives events from an external device. Let P_d denote the set of all 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:

```
#include <sys/ioctl.h>
#include "ptpHwP1000LinuxDr.h"
...
void* handleInterrupt(TypedPort** outputs);
class Device : public Actor{
public:
    Device(Scheduler* s) : Actor(s), input(this), ... {}
    void fire() ;
    void initialize() ;
    TypedPort input, output1, output2;
};
void Device::fire() {
    if(input.hasEvent()) {
        //get the time stamp of the input event.
        Event e = input.getEvent();
        Time t = e.getTime();
        //Get the curret physical time.
        Time currentTime;
        ioctl(1,  FPGA_IOC_GET_TIME, &currentTime);
        if (t > currentTime) {
            //set time trigger with the value of t.
            ioctl(1,  FPGA_IOC_SET_TIMETRIGGER, &t);
        } otherwise {
            ...
        }
    }
}
void Device::initialize() {
    input.isRealTime(true);
    //...
    TypedPort* ports[2];
    ports[0] = &output1; ports[1] = &output2;
    pthread_t  p_thread;
    pthread_create(&p_thread, NULL, handleInterrupt, ports);
}
void* handleInterrupt(TypedPort** outputs) {
  Time t;
  do {
      /* Block until the next interrupt. */
      ...
      /* Wake up here. */
      /* Get time from the hardware driver. */
      ioctl(1, FPGA_IOC_GET_TIME, &t);
      /* Send out an event to output1. */
      Token token (1);  Event e (token, t);
      outputs[0]->send(e);
      /* Get sensor data and send an event to output2. */
      ...
  } while (1);
  pthread_exit(NULL);
}
```

Figure 6. Skeleton code of the Device actor

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.

When clocks in the distributed systems are not perfectly synchronized, we also need to take into account the time synchronization errors in 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.

6 Implementation

We use a lab prototype systems with Agilent IEEE 1588 implementation to illustrate the PTIDES programming model. 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 edge (from voltage level 0 to level 1) at a specified time. The edge can be used to trigger some sensor (e.g. an camera) to perform an action (e.g. taking a picture). After generating this pulse, the hardware interrupts the processor and resets itself to voltage level 0. This physical setup makes it easy to measure very precisely the real-time behavior of the system (for example using an oscilloscope probe).

Figure 6 shows the skeleton code of our implementation of the Device actor in C++ on that platform. The `fire` method is called by the run-time environment when there is an event at the input port. It takes the time stamp of the input event as specification of when to produce the level change. That is, it requests the hardware, using the `ioctl` function of the device driver, to produce a rising edge at physical time equal to the model time of an input event. The `initialize` method is invoked at the beginning of an execution and shall be invoked exactly once per execution of a model. The `initialize` method in the Device actor sets the real-time flag of the input port to be true and creates a new thread to listen to the hardware interrupts after the level changes have been generated. The `handleInterrupt` method defines the execution of the thread. When an interrupt happens, the thread wakes up, outputs an event to the first output to indicate the completion of the level change with time stamp equal to the physical time at which the level is restored to its original value, and outputs another event that encapsulates the sensor data to the second output with time stamp equal to the time when the data is generated.

A set of C++ classes, including Actor, Event, Scheduler, Time, Token and TypedPort have been implemented as the core package of PTIDES run time. Developers only need to focus on the functionality of each actor in a model. For example, other actors in the model shown in figure 2 can be implemented in a similar manner as the Device actor by defining how to react to input events. These actors are generally easier to implement than the Device actor since they do not need to interact with the hardware drivers and deal with threads.

7 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 relate model time and physical time only where this relationship is necessary, and provide an analysis framework and an execution strategy that permit 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.

A key requirement for preserving runtime determinism of PTIDES programs is that each event e with model time t at a real-time port must be received before the physical time exceeds $t-\sigma$, where σ is the setup time of the real-time port. We call a PTIDES program *deployable* if this requirement can be guaranteed. We are working on methods for statically checking deployability for a given PTIDES program and a system characteristic such as communication delay and execution time bounds.

8 Acknowledgments

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, Bosch, DGIST, General Motors, Hewlett Packard, Microsoft, National Instruments, and Toyota.

References

[1] L. d. Alfaro and T. A. Henzinger. Interface-based design. In M. Broy, J. Gruenbauer, D. Harel, and C. Hoare, editors, *Engineering Theories of Software-intensive Systems*, volume NATO Science Series: Mathematics, Physics, and Chemistry, Vol. 195, pages 83–104. Springer, 2005.

[2] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

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

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

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

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

[7] G. Cengic, O. Ljungkrantz, and K. kesson. Formal modeling of function block applications running in iec 61499 execution runtime. In *11th IEEE International Conference*

on Emerging Technologies and Factory Automation, Prague, Czech Republic, September 20-22 2006.

[8] A. Chakrabarti, L. de Alfaro, T. A. Henzinger, and M. Stoelinga. Resource interfaces. In *Proc. of the Third International Conference on Embedded Software (EMSOFT'03)*. Lecture Notes in Computer Science, Springer-Verlag, 2003.

[9] 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.

[10] L. Corts, P. Eles, and Z. Peng. A petri net based model for heterogeneous embedded systems, 1999.

[11] J. Elson, L. Girod, and D. Estrin. Fine-grained network time synchronization using reference broadcasts. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002), Boston, MA.*, December 2002.

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

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

[14] A. Ghosal, T. A. Henzinger, D. Iercan, C. M. Kirsch, and A. Sangiovanni-Vincentelli. A hierarchical coordination language for interacting real-time tasks. In *Sixth Annual Conference on Embedded Software (EMSOFT)*, Seoul, Korea, October 2006. ACM.

[15] T. A. Henzinger. The theory of hybrid automata. *NATO ASI Series F: Computer and Systems Sciences*, 170:265–292, 2000.

[16] 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.

[17] T. A. Henzinger, C. M. Kirsch, and S. Matic. Composable code generation for distributed Giotto. In *International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 21–30. ACM, 2005.

[18] T. A. Henzinger and S. Matic. An interface algebra for real-time components. In *Proc. of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06), San Jose, CA*, April 2006.

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

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

[21] H. Kopetz. *Real-Time Systems : Design Principles for Distributed Embedded Applications*. Springer, 1997.

[22] 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.

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

[24] E. A. Lee and Y. Zhao. Reinventing computing for real time. In *Proc. of the Monterey Workshop 2006, LNCS 4322*, pages 1–25, 2007.

[25] 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.

[26] 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.

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

[28] S. Matic and T. A. Henzinger. Trading end-to-end latency for composability. In *26th Annual Real-Time Systems Symposium (RTSS)*, pages 99–110. IEEE Computer Society Press, 2005.

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

[30] J. L. Paunicka, D. E. Corman, and B. R. Mendel. A CORBA-based middleware solution for UAVs. In *Fourth International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 261 – 267, Magdeburg, Germany, May 2-4 2001. IEEE.

[31] D. C. Schmidt, D. L. Levine, and S. Mungee. The design of the TAO real-time object request broker. *Computer Communications*, 21(4), April 1998.

[32] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture - Patterns for Concurrent and Networked Objects*. Wiley, 2000.

[33] E. Wandeler and L. Thiele. Interface-based design of real-time systems with hierarchical scheduling. In *Proc. of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06), San Jose, CA*, April 2006.

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