Time-centric Models For Designing Embedded Cyber-physical Systems

John C. Eidson
Edward A. Lee
Slobodan Matic
Sanjit A. Seshia
Jia Zou

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-135
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-135.html

October 9, 2009

Acknowledgement

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

Time-centric Models For Designing Embedded Cyber-physical Systems [*]

John C. Eidson, Edward A. Lee, Slobodan Matic, Sanjit A. Seshia, Jia Zou

September 30, 2009

University of California at Berkeley
Berkeley, CA, 94720, USA
eidson@eecs.berkeley.edu
eal@eecs.berkeley.edu
matic@eecs.berkeley.edu
sseshia@eecs.berkeley.edu
jiazou@eecs.berkeley.edu

Abstract

The problem addressed by this paper is that real-time embedded software today is commonly built using programming abstractions with little or no temporal semantics. The focus is on computer-based systems where multiple computers are connected on a network and interact with and through physical processes (the plant) via sensors and actuators. Such systems are often termed cyber-physical systems (CPS).

The paper discusses the use of an extension to the Ptolemy II framework as a coordination language for the design of distributed real-time embedded systems. Specifically, the paper shows how to use modal models in the context of the PTIDES extension of Ptolemy II to provide a firm basis for the design of an important class of problems. Several examples are given to show the use of this environment in the design of interesting practical real-time systems.

KEYWORDS: Computation theory, Computer science, Design methodology, Discrete event systems, Distributed control, Modeling, Real-time systems, Programming environments, Synchronization

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

1 Introduction

In cyber-physical systems (CPS) the passage of time becomes a central feature — in fact, it is this key constraint that distinguishes these systems from distributed computing in general. Time is central to predicting, measuring, and controlling properties of the physical world: given a (deterministic) physical model, the initial state, the inputs, and the amount of time elapsed, one can compute the current state of the plant. This principle provides the foundations of control theory. However, for current mainstream programming paradigms, given the source code, the program's initial state, and the amount of time elapsed, we cannot reliably predict future program state. When that program is integrated into a system with physical dynamics, this makes principled design of the entire system difficult. Instead, engineers are stuck with a prototype-and-test style of design, which leads to brittle systems that do not easily evolve to handle small changes in operating conditions and hardware platforms. Moreover, the disparity between the dynamics of the physical plant and the program seeking to control it potentially leads to errors, some of which can be catastrophic.

The foundations of computing, rooted in Turing, Church, and von Neumann, are about the transformation of data, not about physical dynamics. Although computers have become fast enough to adequately measure and control many physical processes, modern computing techniques such as instruction scheduling, memory hierarchies, garbage collection, multitasking, best-effort networking, and reusable component libraries (which do not expose temporal properties on their interfaces), introduce enormous temporal variability. Those innovations are built on a key premise: that time is irrelevant to correctness; it is at most a measure of quality. By contrast, what a CPS needs is not faster computing, but physical actions taken at the right time. Time needs to be a semantic property, not a quality factor.

The challenge of integrating computing and physical processes has been recognized for some time, motivating the emergence of hybrid systems theories [16]. Progress in that area, however, remains limited to relatively simple systems combining ordinary differential equations with automata. These models inherit from control theory a uniform notion of time, an oracle called t available simultaneously in all parts of the system. Even adaptations of traditional computer science concepts to distributed control problems make the assumption of the oracle t. Olfati-Saber et al. [18], for example, translate consensus problems from computer science into control systems formulations, showing connections between such consensus problems and a variety of dynamical systems problems such as synchronization of coupled oscillators, flocking, formation control, and distributed sensor fusion. These formulations, however, break down without the uniform notion of time that governs the dynamics. In networked software implementations, such a uniform notion of time cannot be precisely realized. Time triggered networks [10] and time synchronization [9] can be used to *approximate* a uniform model of time, but the analysis of the dynamics has to include the imperfections. Without that, one could construct a control system model that violates causality, for example.

This paper is organized as follows. First, section 2 discusses the use of the PTIDES [23] extension to the Ptolemy II simulation framework [6] as a coordination language for the design of distributed real-time embedded systems. Section 3 then defines temporal semantics of PTIDES, and shows how the use of modal models in the context of PTIDES provides a firm basis for the design of an important class of CPS. This is followed by several examples in section 4 , which show the use of this environment in the design of interesting practical real-time systems. Finally, an outline of future work and conclusions are presented in section 5.

2 Design environment

The proposed design environment is an extension of the Ptolemy II framework. Ptolemy II supports modeling, simulation, and design of systems using mixed models of computation (MoC) [6]. Ptolemy II has been extended by the addition of a MoC for timing-centric distributed software called PTIDES (*programming temporally-integrated distributed embedded systems*) [26, 4, 25, 11]. PTIDES models define the interaction of distributed software components, the networks that bind them together, sensors, actuators, and physical dynamics.

PTIDES is based on discrete-event (DE) systems [20, 2, 1, 22], which provide a model of time and concurrency. We specify DE systems using the actor-oriented approach. Actors are concurrent components that exchange time-stamped events via input and output ports. The time in timestamps is a part of the model, playing a formal role in the computation. We refer to this time as *model time*. It may or may not bear any relationship to time in the physical world, which in this paper we will call *physical time* (without attempting define this precisely). In basic DE semantics, each actor processes input events in time-stamp order. There are no constraints on the physical time at which events are processed. We assume a variant of DE that has a rigorous, determinate, formal semantics [15, 12] and that has been shown to integrate well with models of continuous dynamics [14].

PTIDES extends DE by establishing a relationship between model time and physical time at sensors, actuators, and network interfaces. Whereas DE models have traditionally been used to construct simulations, PTIDES provides a programmer's model for deployable cyber-physical systems.

2.1 Constraints relating model time and physical time

In this section, we explain how in PTIDES the model time of DE is related to the physical time of the real world. There are three key constraints that define this relationship.

The basic PTIDES model is explained by referring to Figure 1, which shows three computational platforms (typically embedded computers) connected by a network and having local sensors and actuators. On Platform 3, a component labeled Local Event Source produces a sequence of events that drive an actuator through two other components. The component labeled Computation4 processes each event and (typically) produces an output event with the same timestamp as the input event that triggers the computation. Those events are merged in timestamp order by a component labeled Merge and delivered to a component labeled Actuator1.

In PTIDES, an actuator component interprets its input events as commands to perform some physical action at a physical time equal to the timestamp of the event. The physical time of this event is measured based on clocks commensurate with UTC or a local system-wide real-time clock. This interpretation imposes our first real-time constraint on all the software components upstream of the actuator. Each event must be delivered to the actuator at a physical time earlier than the event's timestamp.

In Figure 1, Platform 3 contains an actuator that is affected both by some local control and by messages received over the network. The local control commands are generated by the actor labeled Local Event Source, and modified by the component labeled Computation4. The Merge component can inject commands to the actuator that originate from either the local event source or from the network. The messages from the network may depend on sensor data obtained on platforms 1 and 2. The commands are merged in order of their timestamps.

In Figure 1, notice that the top input to the Merge component comes from components that get inputs from sensors on the remote platforms. The sensor components, like the actuator components, are (typi-

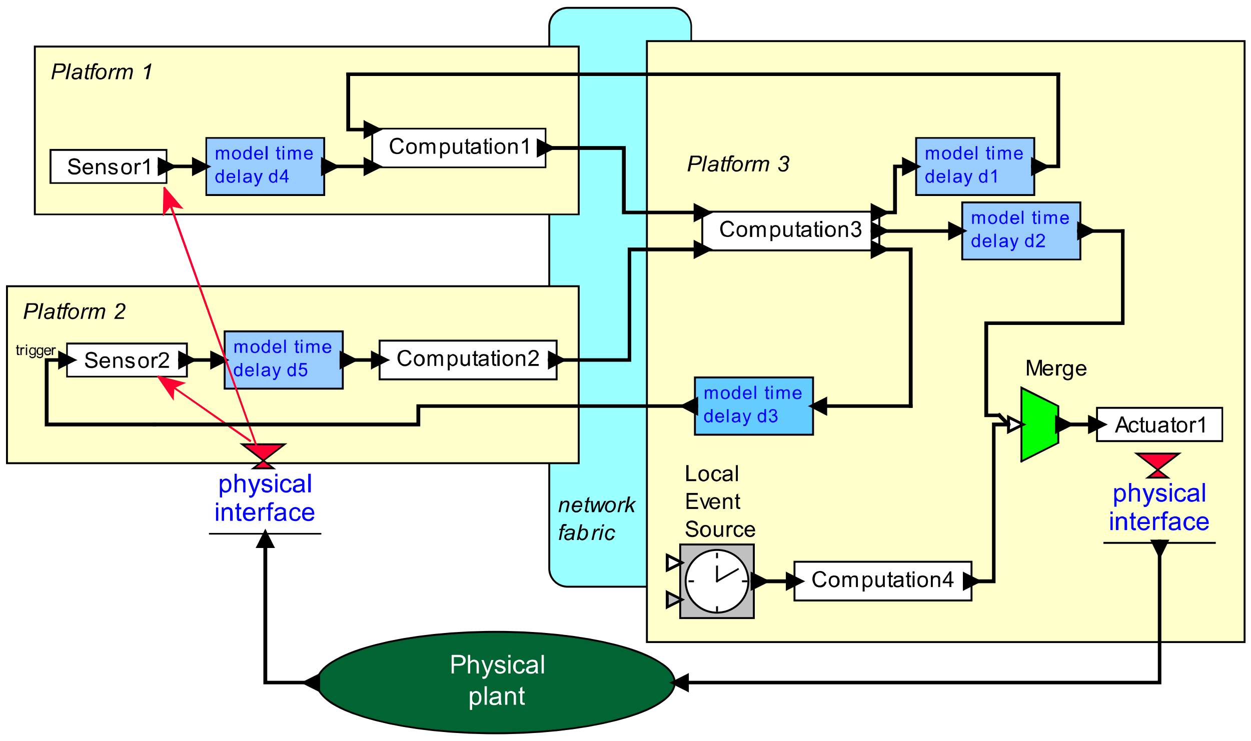

Figure 1. Prototypical CPS

cally thin) software wrappers around hardware drivers. They produce on their output ports time-stamped events. Here, the PTIDES model imposes a second relationship between model timestamps and physical time. Specifically, when a sensor component produces a time-stamped output event, that timestamp must be less than or equal to physical time, however physical time is measured. The sensor can only tell the system about the past, not about the future.

The third and final relationship refers to network interfaces. In this work we assume that the act of sending an event via a network is similar to delivering an event to an actuator; i.e., the event must be delivered to the network interface by a deadline equal to the timestamp of the event. Consider Platform1 in Figure 1 as an example. When an event of timestamp τ is to be sent into the network fabric, the transmission of this event needs to happen no later than physical time τ. In general, we could set the deadline to something other than the timestamp, but for our purposes here, it is sufficient that there be a deadline, and that the deadline be a known function of the timestamp. Our assumption that it equals the timestamp makes the analysis particularly simple, so we proceed with that.

2.2 Real time delays in sensors and actuators

It is clearly possible for a sensor or an actuator to introduce real delays. Specifically, in Figure 1, the "physical interface" that provides inputs to the sensor actors may introduce delays. These delays would be any elapsed (physical) time between the actual sampling of the physics and the generation of the event timestamp. In addition there can be a computational or communication delay within a processor between the time a sensor generates a timestamp and when this timestamp is available for computation in the PTIDES environment. The first delay may affect how the closed-loop system behaves, and the second may affect whether deadlines at actuators can be met (i.e., whether the model is feasible, as discussed in section 5).

Actuators have similar issues. There may be significant time between the delivery of an event to an actuator actor and the time the actuator affects the physics. This corresponds to setup time for a digital circuit for example. The model designer may need to take this into account.

2.3 Event processing in PTIDES

Under benign conditions [15, 12], DE models are determinate in that given the time-stamped inputs to the model, all events are fully defined. Thus, any correct execution of the model must deliver the same time-stamped events to actuators, given the same time-stamped events from the sensors (this assumes that each software component is itself determinate). An execution of a PTIDES model is required to follow DE semantics, and hence deliver this determinacy. It is this property that makes executions of PTIDES models *repeatable*. A test of any "correct" execution of a PTIDES model will match the behavior of any other correct execution.

The key question is how to deliver a "correct" execution. For example, consider the Merge component in Figure 1. This component must merge events in time-stamp order for delivery to the actuator. Given an event from the local Computation4 component, when can it safely pass that event to the actuator? Here lies a key feature of PTIDES. The decision to pass the event to the actuator is made locally at run time by comparing the timestamp of the event against a local clock that is tracking physical time. This strategy results in decentralized control, removing the risks introduced by a single point of failure, and making systems much more modular and composable.

How is this done? There are two key assumptions made in PTIDES. First, distributed platforms have real-time clocks synchronized with bounded error. These real-time clocks provide our measurement of physical time. The PTIDES model of computation works with any bound on the error, but the smaller the bound, the tighter the real-time constraints can be. Time synchronization techniques [9] such as IEEE 1588 [8] can deliver real-time clock synchronization with bounded errors. In fact, an Ethernet PHY chip supporting IEEE 1588 introduced in 2007 by National Semiconductor advertises a clock precision on the order 8 ns on a local-area network [21].

Second, PTIDES requires that there be a bound on the communication delay between any two hardware components. Specifically, sensors and actuators must deliver time-stamped events to the run-time system within a bounded delay, and a network must transport a time-stamped event with a bounded delay. Bounding network delay is potentially more problematic when using generic networking technologies such as Ethernet and TCP/IP, but bounded network delay is already required today in the applications of interest here. This has in fact historically forced deployments of these applications to use specialized networking techniques (such as time-triggered architectures [10], FlexRay, and CAN buses). For now it is sufficient to observe that these boundedness assumptions are achievable in practice. Since PTIDES allows detection of run-time timing errors, it is possible to model responses to failures of these assumptions.

Once these two assumptions (bounded time synchronization error and bounded communication latencies) are accepted, together with deadlines for network interfaces and actuators, local decisions can be made to deliver events in Figure 1 without compromising the determinate DE semantics. Specifically, in

Figure 1, notice that the top input to the Merge comes from Sensor1 and Sensor2 through a chain of software components and a network link. Static analysis of these chains reveals the operations performed on timestamps. In particular, in this figure, assume that the only components that manipulate timestamps are the components labeled *model time delay* d_i. These components accept an input event and produce an output event with the same data payload but with a timestamp incremented by d_i.

Assume we have an event e with timestamp τ at the bottom input of Merge, and that there is no other event on Platform 3 with an earlier time stamp. This event can be passed to the output only when we are sure that no event will later appear at the top input of Merge with a timestamp less than or equal to τ. This will preserve DE semantics. When can we be sure that e is safe to process in this way?

We assume that events destined to the top input of Merge must be produced by a reaction in Computation3 to events that arrive over the network. Moreover, the outputs of Computation3 are further processed to increment their timestamps by d_2. Thus, we are sure e is safe to process when no events from the network will arrive at Platform 3 with timestamps less than or equal to $\tau - d_2$. When can we be sure of this? Let us assume a network delay bound of n and a clock synchronization error bound of s between platforms. By the network interface assumption discussed above, we know that all events sent by Platform 1 or Platform 2 with timestamps less than $\tau - d_2$ will be sent over the network by the physical time $\tau - d_2$. Consequently, all events with timestamp less than or equal to $\tau - d_2$ will be received on Platform3 by the physical time $\tau - d_2 + n + s$, where the s term accounts for the possible disagreement in the measurement of physical time. Thus when physical time on Platform 3 exceeds $\tau - d_2 + n + s$, event e will be safe to process.

It is easy to see that if the model is static (components are not added during runtime and connections are not changed), then given enough information about each component, a similar analysis can be made for all paths through the model. This analysis is done at design time. PTIDES components include causality interfaces with algebraic compositionality properties [24], enabling automatic analysis. At runtime, the only test performed to ensure DE semantics is to compare timestamps to physical time with an offset (in the previous example, the offset is $-d_2 + n + s$). This is not expensive to implement.

Note that the distributed execution control of PTIDES introduces another valuable form of robustness in the system. For example, in Figure 1, if, say, Platform 1 ceases functioning altogether, and stops sending events on the network, that fact alone cannot prevent Platform 3 from continuing to drive its actuator with locally generated control signals. This would not be true if we preserved DE semantics by conservative techniques based on the work by Chandy and Misra [3]. It is also easy to see that PTIDES models can include components that monitor system integrity. For example, Platform 3 could raise an alarm and change operating modes if it fails to get messages from Platform 1. It could also raise an alarm if it later receives a message with an unexpectedly small timestamp. Time synchronization with bounded error helps to give such mechanisms a rigorous semantics.

Moreover, since execution of a PTIDES model carries timestamps at run time, run time violations of deadlines at actuators can be detected. PTIDES models can be easily made adaptive, changing modes of operation, for example, when such real-time violations occur. In general, therefore, PTIDES models provide adequate runtime information for detecting and reacting to a rich variety of timing faults.

Recall that in PTIDES models, timestamps represent a model time, and that model time need not have any relationship to time in the physical world. In PTIDES, we establish such a relationship at sensors, actuators and network interfaces. Thus an execution of a PTIDES model has considerable freedom to process an event earlier or later than the physical time corresponding to the timestamp of the event. As long as events are delivered on time and in time-stamp order to actuators, the execution will look exactly the same to the environment. This makes PTIDES models much more robust than typical real-time software, because small changes in the (physical) execution timing of internal events are not visible to the environment (as long as real-time constraints are met at sensors, actuators and network interfaces).

3 Temporal semantics in PTIDES

PTIDES semantics is fully described in [23, 26] and is based on a tagged-signal model [13]. For this discussion the important point is that actors define a functional relationship between a set of tagged signals on the input ports and a set of tagged signals on the output ports of the actor.

$$F_a : S^I \rightarrow S^O \tag{1}$$

Here, I is a set of input ports, O is a set of output ports, and S a set of signals. The signals $s \in S$ are sets of (timestamp, value) pairs of the form $(\tau, v) \in T \times V$ where the time set T represents time and V is a set of values (the data payloads) of events.

For simulation, the most common use of DE modeling, timestamps typically have no connection with real time, and can advance slower or faster than real time [22]. The timestamps can be real numbers (or their approximations as floating point numbers). This is the choice of many commonly used discrete-event simulators. They may instead be given by integers, as used by most hardware description languages. PTIDES approximates a superdense model of time [17], because it facilitates models that mix continuous dynamics with discrete-event models [14]. Superdense timestamps are tuples (t, n) that support a notion of a sequence of causally-related simultaneous events. The set of superdense time stamps that we assume here is $T = \mathbb{R}_{\geq 0} \times \mathbb{N}$. Here $\mathbb{R}_{\geq 0}$ represents the set of values of time in the PTIDES model environment and $\mathbb{N}$ represents a set of index ordering events with the same value of model time. For a particular value of a timestamp $\tau = (t, n)$, t is called the model time and n the *microstep*.

Actors are permitted to modify the timestamp and most commonly will modify the model time member t of the timestamp $\tau = (t, n)$ to indicate the passage of model time. For example, a delay actor has one input port and one output port and its behavior is given by $F : S \rightarrow S$ where for each $s \in S$:

$$F_\delta(s) = \{((t + \delta, n), v) \mid ((t, n), v) \in s\} \tag{2}$$

That is, the output events are identical to input events except that the model time is increased by δ, a parameter of the actor.

Consider the simple sensor, actor, actuator system of Figure 2. In this example we assume $F_a(s) = \{((t, n), 2 * v) \mid ((t, n), v) \in s\}$; i.e., the output is the same as the input but with its value scaled by a factor of 2. Both variants (a) and (b) of this figure show a serial combination of a sensor, delay, scaling, and actuator actors. The sensor actors produce an event (25 seconds, 15 volts) where the timestamp 25 seconds is the physical time at the time of sensing. For this discussion the value of the superdense microstep n of an event $((t, n), v)$ is omitted. The delay actor increments the model time part of the timestamp by 10 and the scale actor doubles the value member from 15 volts to 30 volts. In both cases the actuator receives an event (35 seconds, 30 volts), which is in accordance with the PTIDES model it interprets as a command to the actuator to instantiate the value 30 volts at a physical time of 35 seconds. As long as deadlines at the actuators are met, models (a) and (b) are identical in that all observable effects are identical, regardless of computation times and scheduling decisions.

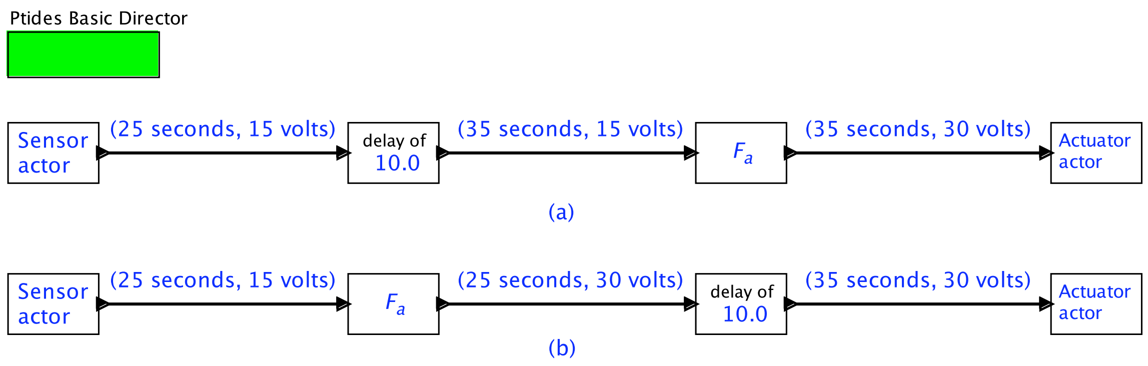

Figure 2. Linear combination of actors

3.1 Feasibility

Rather than simply using these models for simulation, this design environment uses these models as the basis for generating executable code for a particular target environment. Since PTIDES is based on DE, it is possible to generate code that preserves DE semantics, thus ensuring that the implementation is correct in the sense that events are processed in timestamp order. It is also necessary to consider whether this generated code can actually meet the specified deadlines on a given platform. A solution that meets both DE semantics and the specified execution deadlines is termed a *feasible* solution. To determine feasibility, we must consider the execution times of the various actors in order to generate a feasible schedule if one exists.

For a feasible solution the delay between the physical time of a sensor input and the physical time of a resulting actuation must not be less than the total execution time of the PTIDES actors involved. For example, let the total execution time of the actors of Figures 2 (a) or (b) be E. Unlike model time delays, which are in general application specific, execution times are platform specific, and are unrelated to model time delays. E must also include any time interval between the actual sensing time and the time the sensor value is available to the PTIDES environment and importantly the interval between delivering the final tagged signal to the actuator and the time at which is can be applied to the physical (analogous to setup time for a digital circuit).

It is easy to prove that for simple designs such as that of Figure 2 the following two constraints imply feasibility.

$$\delta \geq E \tag{3}$$

where δ is the total model time delay, e.g. 10 in Figure 2, and E is the total execution time. Further assume that the sensor inputs are either periodic with period Δ or sporadic with a minimum time between occurrences of Δ, then in addition the following constraint must hold.

$$\Delta \geq E \tag{4}$$

It is easy to see that in the case of Figure 2 both variants (a) and (b) are also equivalent with respect to feasibility.

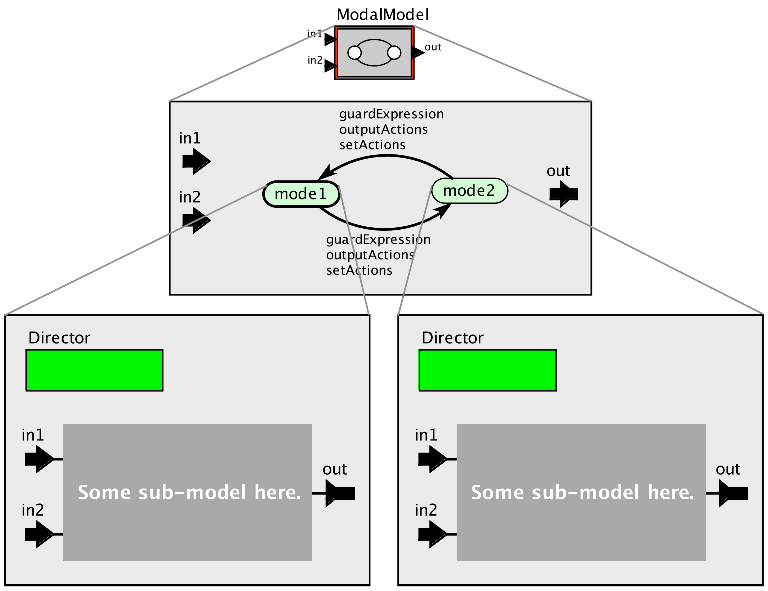

Figure 3. General pattern of a modal model with two modes, each with its own refinement.

3.2 Modal models

The use of modal models is well established both in the literature, for example Statecharts [7], UML [19], and in commercial products such as Simulink®/Stateflow® from the MathWorks™. Our style for modal models follows the pattern shown in Figure 3. A modal model is an actor, shown in the figure with two input ports and one output port. Inside the actor is a finite state machine (FSM), shown in the figure with two states, labeled *mode1* and *mode2*. The transitions between states have guards and actions, and each state has a *refinement* that is a submodel. The meaning of such a modal model is that the input-output behavior of the ModalModel actor is given by the input-output behavior of the refinement of the current state.

In the proposed design environment, modal models can be incorporated into both the model of the real-world as well as the PTIDES models of the embedded world. This requires careful specification of the temporal semantics of all actors including modal models to allow analysis of the resulting system and to enable code generation that preserves the timing semantics when executing as a run-time artifact on a computing platform.

Modal models introduce additional temporal considerations into a design. This is especially true for modal models that modify the timestamp $\tau = (t, n)$ of a signal. There are several possible semantics that could be used to define the behavior of a modal model consistent with DE semantics. The principal temporal semantics in this design environment are that events are executed in timestamp order. For modal models it is also necessary to specify the order of execution of the internal aspects of the model. While the Ptolemy II environment provides several modal model execution options such as a preemptive evaluation of guards prior to execution of a state refinement, the principal features critical to the discussion of the examples in this paper are as follows. A modal model executes internal operations in the following order:

- When the modal model reacts to a set of input events with timestamp τ, it first presents those input

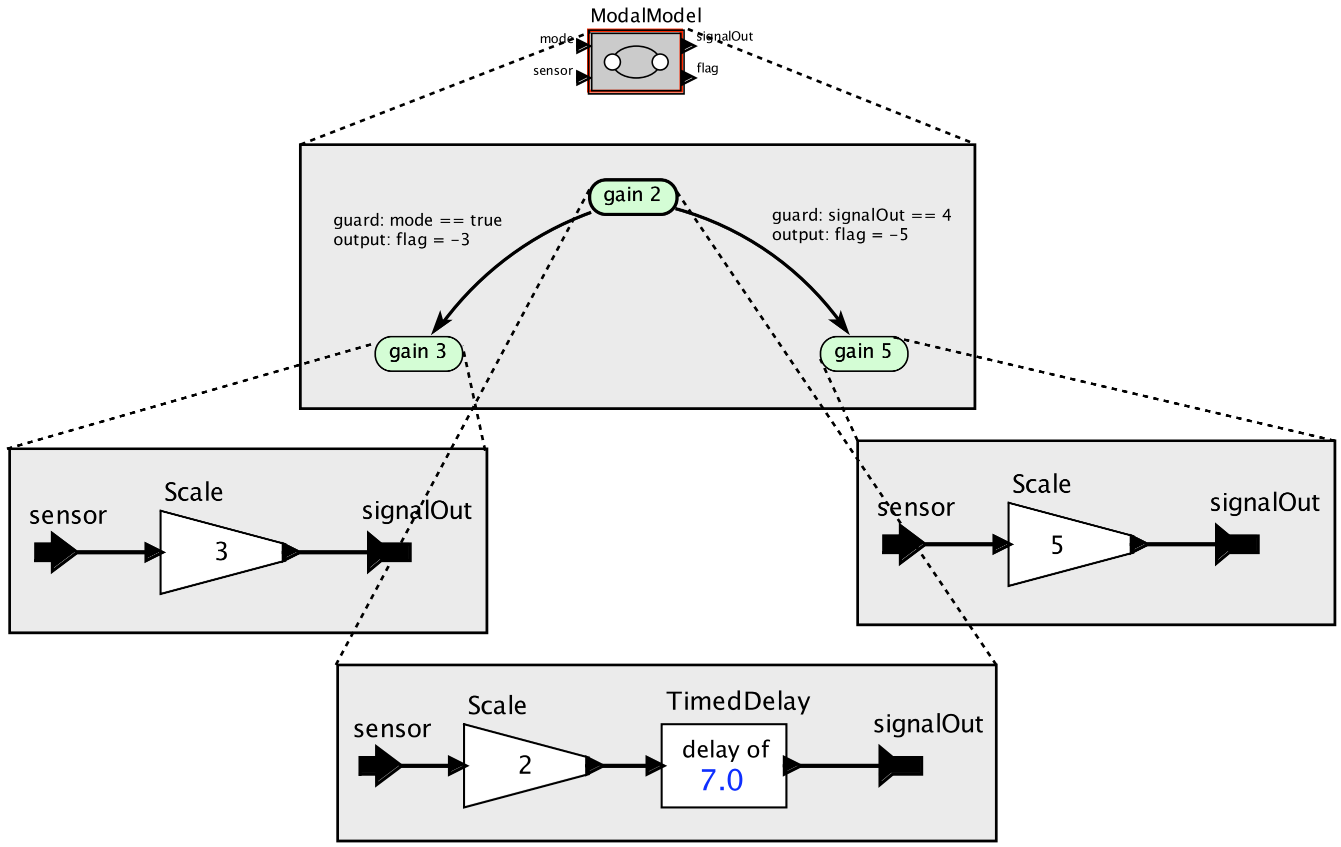

Figure 4. Simple time-sensitive modal model

events to the refinement of the current state i. That refinement may, in reaction, produce output events with timestamp τ.

- If any of input events have an effect within the refinement at a later timestamp $\tau' > \tau$, that effect is postponed. The modal model is invoked again at timestamp τ', and only if the current state is still i will the effect be instantiated.
- The guards of all transitions originating from the current state are evaluated based on the current inputs, state variables, and outputs of the current state refinement with the same timestamp τ as the current inputs.
- If one of the guards evaluates to true, the transition and any associated actions are executed, and the new current state i' becomes that at the destination of the transition.

Thus all phases of the execution of a modal model occur in strict timestamp order in accordance with DE semantics. While straightforward, these rules can yield surprises particularly when one or more of the refinements modify the model time of a signal.

For example consider the simple modal model of Figure 4. The two inputs to this state machine are *mode* and *sensor*. The two outputs are *signalOut* and *flag*. For this example it is assumed that the guards are never both true.

Suppose a *sensor* event $(t, v) = (10, 30)$ (again omitting the microstep) is received while the FSM is in state *gain 2*. The refinement of this state generates an output $(17, 60)$. If no state transition occurs before time $t = 17$ then at time $t = 17$ the postponed *signalOut* event $(17, 60)$ will be produced.

However suppose that at time $t = 12$ a *mode* event $(12, true)$ occurs. This will cause a transition to state *gain 3* at time $t = 12$. In this case the postponed *signalOut* event $(17, 60)$ is not produced. While in state *gain 3* a *sensor* event, say $(15, 3)$, will result in a *signalOut* event $(15, 9)$. The event is not postponed since the refinement does not contain a delay actor.

Similarly, suppose *sensor* events $(5, 1)$ and $(9, 2)$ are received with the FSM in state *gain 2*. The refinement of this state generates output events $(12, 2)$ and $(16, 4)$ which must be postponed until times $t = 12$ and $t = 16$ respectively. Following the rules above, at time $t = 12$, a *signalOut* event $(12, 2)$ occurs. At $t = 16$ the FSM again executes to handle the postponed event $(16, 4)$. The first thing that happens is the instantiation of the *signalOut* event $(16, 4)$. Next, the guards on the FSM are evaluated and a transition occurs at $t = 16$ to the state *gain 5*. A subsequent *sensor* signal $(17, 1)$ then results in a *signalOut* event $(17, 5)$.

These examples illustrate that careful attention must be paid to the temporal semantics of the modal models to ensure that the desired application behavior results. It is still an open question whether the semantics implemented are appropriate to enable the design of target applications or whether alternate temporal semantics must be provided in place of or in addition to the current design.

3.3 Constraints on model time delays

For every causal path between a sensor input and an actuator output, the path model delay must exceed the sum of the execution times of the actors along the path, including any execution time from other paths that impacts the execution time of the path in question. This raises an interesting design question since 1- a minimum model time delay must be provided in order to avoid causality problems, and 2- these required minimum delays obviously may influence the choice of model delays within modal models and elsewhere which are usually delay specifications of applications. An example of this will be discussed in section 4.3.

Consider the system of Figure 5. For simplicity, consider a single event from the Sensor actor propagating through the model without any mode changes during its propagation. There are two possible paths between the sensor and the actuator, depending on the state of the modal model. For the path through state 1 (call that path *path1*), the total delay is $\delta_{path1} = \delta_4 + \delta_8$ and the total execution time is $E_{path1} = E_1 + E_2 + E_3 + E_7 + E_9$ (assuming the execution time of the model delay actors is negligible). For the path through state 2 (call it *path2*), the corresponding delay and execution time values are $\delta_{path2} = \delta_6 + \delta_8$ and $E_{path2} = E_1 + E_2 + E_5 + E_7 + E_9$. The feasibility constraints require that $\delta_{path1} \geq E_{path1}$ and $\delta_{path2} \geq E_{path2}$.

A complete feasibility analysis, of course, will need to take into account the sporadic nature of events from the Sensor actor, and also the time at which state transitions can be taken. In addition, application requirements may impose specific bounds on these parameters of the form $\delta_{path1} = 25$ seconds and $\delta_{path2} = 5$ seconds. Finding a feasible solution requires considering all these constraints while preserving the desired modal model temporal semantics of the application by properly distributing the delays among the delay actors of the system. While understandable for this simple example, in a complex system these considerations point out the need for automated or semi-automated analysis tools. Such tools will require visibility of causality via actor interfaces and in particular the visibility of any model time

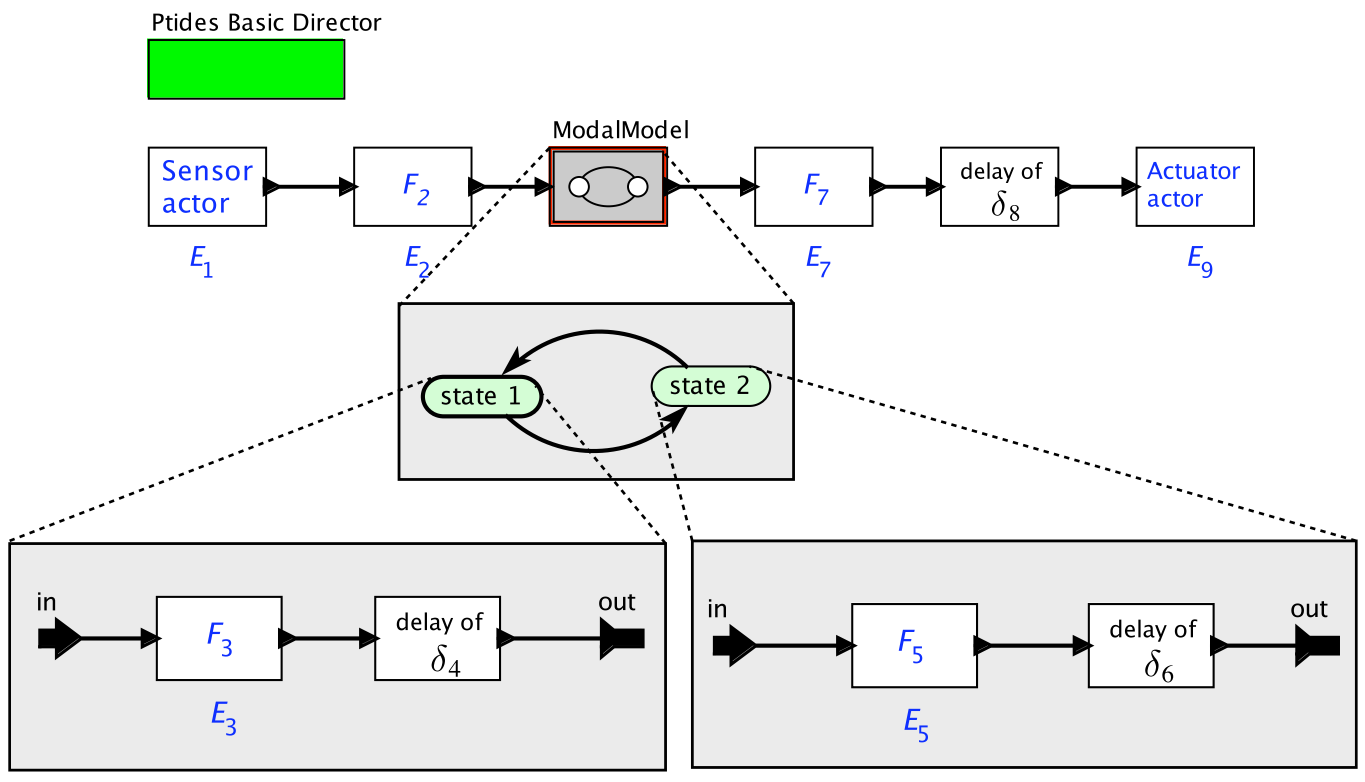

Figure 5. Specifying model delay

delays present within the actor.

4 Application studies

PTIDES can be used to integrate models of software, networks, and continuous dynamics. A practical consequence is to enable co-design and co-simulation of software controllers, networks, and the physical plant. It also facilitates *hardware in the loop* (HIL) simulation, where deployable software can be tested (at greatly reduced cost and risk) against simulations of the physical plant. The DE semantics of the model ensures that simulations will match implementations, even if the simulation of the plant cannot execute in real time. Conversely, prototypes of the software on generic execution platforms can be tested against the actual physical plant. The model can be tested even if the software controllers are not fully implemented. This (extremely valuable) property cannot be achieved today because the temporal properties of the software emerge from an implementation, and therefore complete tests of the dynamics often cannot be performed until the final stages of system integration, with the actual physical plant, using the final platform.

Closed loop control typically requires a fixed or at least a bounded delay in the controller that is compatible with loop time stability considerations. PTIDES enables enforcement of deterministic delays in the code generated for embedded systems.

The inclusion of a network into an embedded system introduces three principal complications in the design embedded systems:

- To preserve DE semantics and the resulting determinism system wide, it is necessary to provide a common sense of time to all platforms. As noted in section 2 this is often based on a time-slotted network protocol but can also be based on a clock synchronization protocol such as IEEE 1588.

- The design of model delays must now account not only for execution time within an actuation platform, e.g. the platform containing an actuator causally dependent on signals from other platforms, but must include network delay as well as execution time in platforms providing signals via the network to the actuation platform.
- To ensure bounded network delay it is usually necessary to enforce some sort of admission control explicitly controlling the time that traffic is introduced onto the network.

The introduction of timed reactions further complicates the design and analysis of system temporal semantics, particularly when these reactions must be synchronized across a multi-platform system. PTIDES is well suited in managing these multi-platform design issues.

In addition it appears that a design environment with the properties of PTIDES is required for the robust generation of executable code (or FPGA images) which preserves the timing semantics of the design. Since generating feasible execution schedules is critical, it helps that the PTIDES environment gives the user explicit control of timing within the system.

The solid foundation of PTIDES should enable design and code generation for high value embedded systems including high confidence medical devices and systems, traffic control and safety, advanced automotive systems, process control, energy conservation, environmental control, avionics, instrumentation, critical infrastructure control (electric power, water resources, and communications systems for example), distributed robotics (telepresence, telemedicine), defense systems, and manufacturing.

The remainder of this section discusses the use of PTIDES for some important industrial applications, namely:

- The use of time-based detection of missing signals to drive mode changes in the operation of power plants. This technique is also applicable in implementing holdover properties of reference clocks used in telecommunication systems.
- The use of time-based models of the plant in testing controller implementations of power plants.
- The use of timed sequences of operations to define startup, normal, shutdown, and emergency sequencing of the power supplies in a test and measurement system.

4.1 Power plant control

The design of the control systems for large electric power stations is interesting in that the physical extent of the plant requires a networked solution. The two critical design issues of interest here are the precision of the control loop, since this effectively controls the electrical output, voltage, and frequency which must be matched to that of the grid, and the reaction time to failures. Due to the large mass of the turbine and generator, the loop time is relatively long. Failure reaction time specifications must be met to prevent expensive or dangerous conditions due to the large amounts of stored energy in the system. For example, in the case where the electric grid is disconnected, the fuel supply to the turbine must typically be reduced within a few milliseconds.

A typical power plant can involve up to 3000 nodes comprising monitoring equipment separated by several hundred meters. Many of these nodes have modest sampling intervals and data requirements but in the aggregate produce rather large volumes of data. Typical are temperature and pressure gauges, equipment status, etc. on cooling towers, motors, pumps, pipelines, and all the other elements peripheral to the turbines and generators themselves. These measurements are sent, possibly after some local processing and data reduction, to a central repository for later analysis and archiving and some are sent to the system controllers. Since the purpose of this data is to make decisions about the state of the physical world, it is critical that the time at which each measurement is made be known to an accuracy and precision appropriate to the physics being measured. The PTIDES design system allows these measurement times to be precisely specified and time-stamped with respect to the real-time clocks in the separate platforms.

Figure 6 illustrates a model of a power plant that is hopefully readable without much additional explanation. The model includes a Generator/Turbine Model, which models continuous dynamics, a model of a communication network, and a model the supervisory controller. The details of these three components are not shown. Indeed, each of these three components can be quite sophisticated models, although for our purposes here will use rather simple versions.

The model in Figure 6 also includes a local controller, which is expanded showing two main components, a Heartbeat Detector and Plant Control block. The Plant Control block is a modal model with four states, as shown. The Down state represents the off state of the power plant. Upon receipt of a (time-stamped) startup event from the supervisory controller, this modal model transitions to the Startup state. When the measured discrepancy between electric power output and the target output gets below a threshold given by *errorThreshold*, the modal model transitions to the Normal state. If it receives a (time-stamped) *emergency* event from the Heartbeat Detector, then it will transition to the Shutdown state, and after achieving shutdown, to the Down state. Each of these states has a refinement (not shown) that uses input sensor data to specify the amount of fuel to supply to the generator/turbine. The fuel amount is sent over the network to the generator/turbine.

This model is executable. The plots generated by the two Plotter actors in Figure 6 are shown in Figure 7 for one simulation. In this simulation, the supervisory controller issues a startup request at time 1, which results in the fuel supply being increased and the power plant entering its Startup mode. Near time 7.5, a *warning* event occurs and the supervisory controller reduces the target output level of the power plant. It then reinstates the higher target level around time 13. The power plant reaches normal operation shortly before time 20, and around time 26, a warning and emergency occur in quick succession. The power plant enters its Shutdown state, and around time 33 its Down state. Only a startup signal from the supervisory controller can restart the plant.

This model has a number of interesting features that are typical in such distributed control applications. First, it has two levels of control, supervisory control and local control. Emergency shutdown is handled entirely by the local control, whereas reduction in power output due to a warning is handled by the supervisory control. One of the challenges in designing such mixed control strategies is to prevent accidental overrides, where for example the supervisory control might reinstate normal operation while the local controller is attempting to shut down the plant. The PTIDES model ensures that even commands that originate from different places on the network have a deterministic ordering determined by their time stamps.

The timestamps not only give a determinate semantics to the interleaving of events, but they can also be explicitly used in the control algorithms. This power plant control example illustrates this point by the technique it uses to send *warning* and *emergency* events. Specifically, as shown in Figures 6 and 7, the Generator/Turbine Model sends (time-stamped) sensor readings over the network to the Local

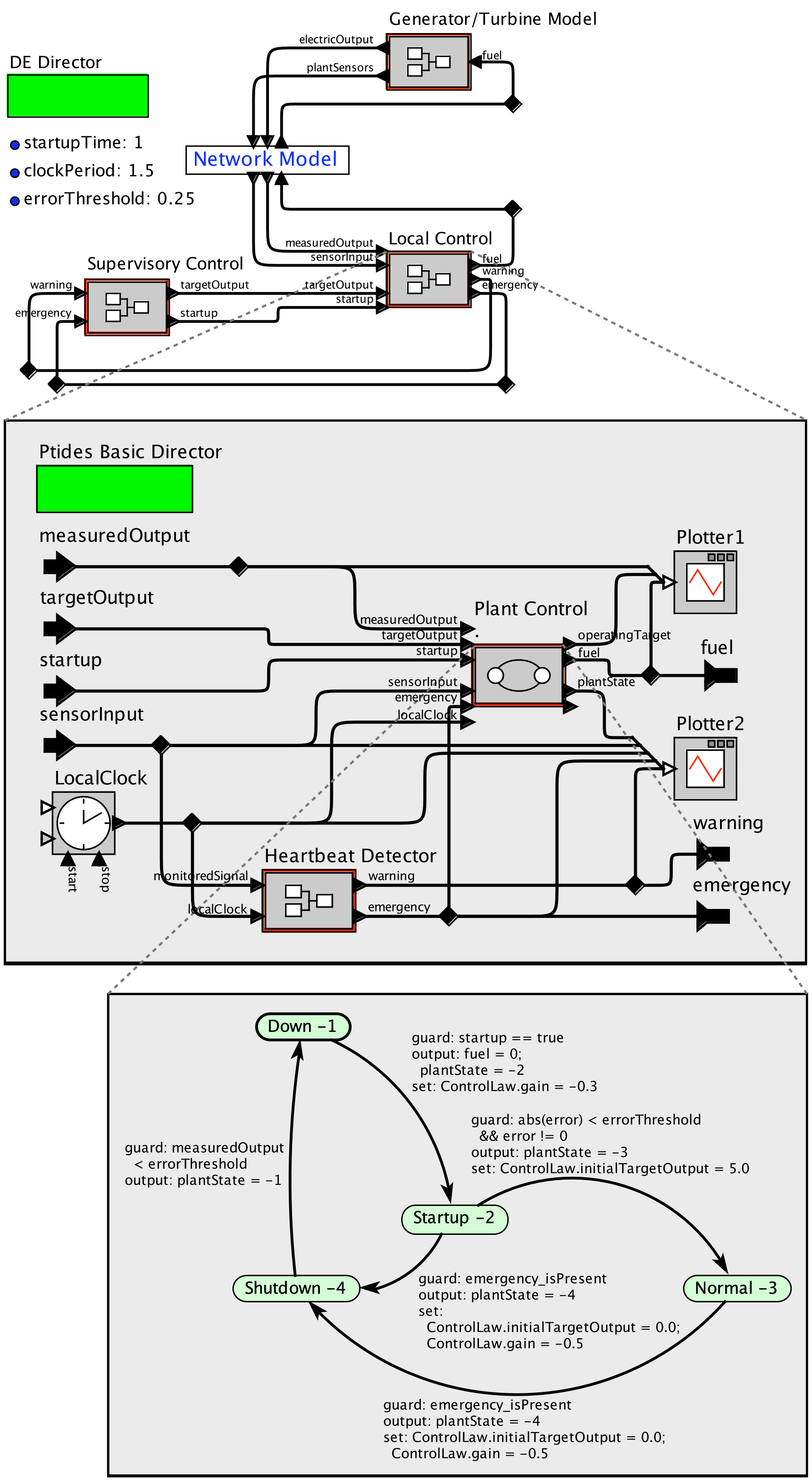

Figure 6. Model of a small power plant.

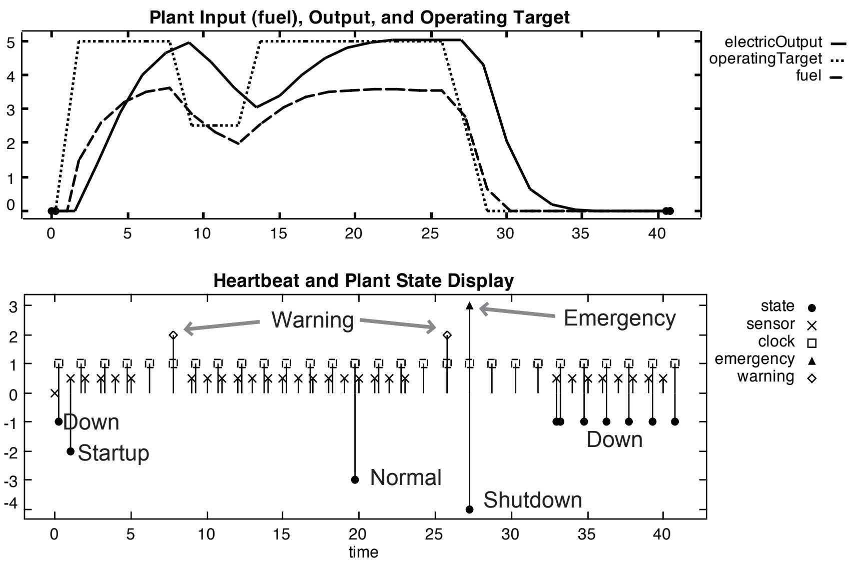

Figure 7. Power plant output and events

Control component. These sensor events are shown with "x" symbols in Figure 7. Notice that just prior to each *warning* event, there is a gap in these *sensor* events. Indeed, this Local Control component declares a warning if between any two local clock ticks it fails to receive a sensor reading from the Generator/Turbine Model. If a second consecutive interval between clock ticks elapses without a sensor message arriving, it declares an emergency and initiates shutdown.

The mechanism for detecting the missing sensor reading messages is shown in Figure 8. In that figure, the *monitoredSignal* input provides time-stamped sensor reading messages. The *localClock* input provides time-stamped events from the local clock. The MissDetector component is a finite state machine with two states. It keeps track of whether the most recently received event was a sensor message or a local clock event. This is possible because PTIDES guarantees that these message will be delivered to this component in time-stamp order, *even when the messages and their timestamps originate on a remote platform elsewhere in the network*. This MissDetector component issues a missed event (with value true) if two successive local clock events arrive without an intervening sensor event. The missed event will have the same timestamp as the local clock event that triggered it.

The second component, labeled StatusClassifier, determines how to react to missed events. In this design, upon receiving one missed event, it issues a warning event. Upon receiving a second missed event, it issues an emergency event. Note that this design can be easily elaborated, for example to require some number of missed events before declaring a warning. Also note that it is considerably easier in this framework to evaluate the consequences of design choices like the local clock interval. Our point is not to defend this particular design, but to show how explicit the design is. The PTIDES semantics guarantee that an implementation will behave exactly like the simulation, given the same time-stamped inputs, and hence the resulting design is principled and complete. Moreover, it is easy to integrate a simulation model of the plant, thus evaluating design choices well before system integration.

A more detailed discussion of the design issues illustrated in this example for an actual commercial power plant control system is found in [5].

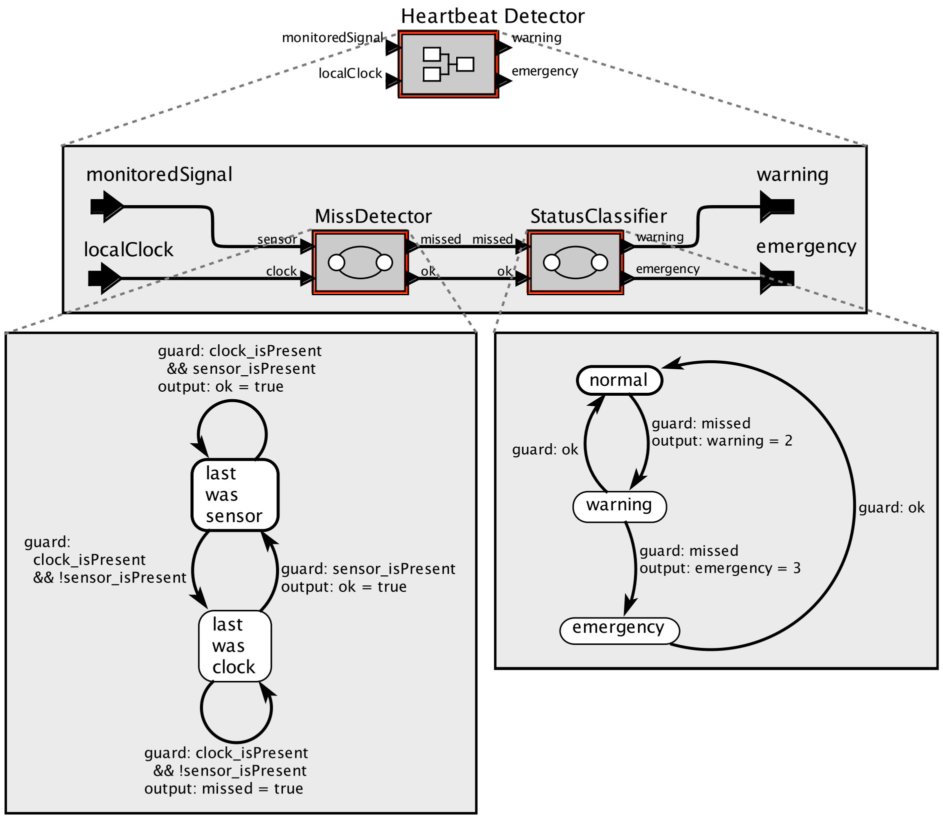
Heartbeat Detector
monitoredSignal
localClock
warning
emergency
monitoredSignal
localClock
MissDetector
sensor
clock
missed
ok
StatusClassifier
missed
ok
warning
emergency
warning
emergency
guard: clock_isPresent && sensor_isPresent output: ok = true
last was sensor
guard: clock_isPresent && !sensor_isPresent
guard: sensor_isPresent output: ok = true
last was clock
guard: clock_isPresent && !sensor_isPresent output: missed = true
normal
guard: missed output: warning = 2
guard: ok
warning
guard: ok
guard: missed output: emergency = 3
emergency

Figure 8. Heartbeat detector that raises warning and emergency alarms.

4.2 Holdover applications

The modal model and missing signal techniques discussed in section 4.1 above are also applicable to the design of reference clocks in a telecommunications system. Telecommunications systems represent one of the largest and most complex distributed applications in the world. A critical feature of these systems is the distribution of precise frequency and in some cases precise time among the various devices. For example, the transmission of digital information for services such as E1/T1 lines requires that transmitter and receiver pairs agree on transmission frequency to prevent loss of information. Specifications on frequency accuracy are set by several standards organizations such as the ITU and are typically $\leq 1 \times 10^{-11}$ over long periods of time.

In these systems, frequency and time are established by a suite of primary reference clocks (PRCs) which are synchronized to a common source such as GPS and which in turn distribute time and frequency information to client devices. PRCs consist of a very high quality quartz oscillator and an atomic clock, usually either cesium or rubidium based. The quartz oscillator provides short term stability and is disciplined by the atomic clock which provides long term stability. The best of these devices can achieve an accuracy of $\leq 5 \times 10^{-13}$ with a stability of $\leq 8 \times 10^{-14}$ over changes in all environmental variables and $\leq 1 \times 10^{-14}$ averaged over a 5 day period. PRC clients are typically lower performance (and lower cost) devices which requires them to be synchronized to a PRC to meet international telecommunications standards. Both PRCs and their clients are typically single platform devices. To a PRC client the reference source typically appears as a component (analogous to the sensors in the power plant discussion) providing synchronization information over a network.

What happens when a PRC client device loses contact with the PRC? Again organizations such as the ITU have set specifications, called *holdover specifications*, that detail the required frequency accuracy as a function of time since the loss of contact with the PRC. The PRC client holdover architecture is much like the startup, normal and shutdown modes of the power plant example and indeed the basic structure of the controller of Figure 6 can be used but with the state refinements and transitions redefined as discussed below.

A PRC client typically generates an error signal based on the difference between the output of the local clock and the incoming reference signal from the PRC. This error signal, perhaps in conjunction with environmental sensor signal inputs, is used by the controller to compute a correction to the local clock.

The input governing the state transitions of the controller FSM is a signal indicating whether the PRC signal is valid and present. In the case of a PRC client controller the substates are:

- State 1: Startup. At system startup or after coming out of the holdover state it must be presumed that the error signal is large and more importantly that any parameters of the algorithms are out of date. In this state a correction signal is computed based on either a default or the most recent parameters and the values of these parameters are updated to reflect current conditions including any environmental information available from sensors. In addition, there are often application requirements that limit the rate of change of the output and these would appear as a slew rate limitation on the correction signal.

- State 2: Normal. In this state the error is below some application set limit. In this state the correction algorithm may be somewhat different than in the startup state while making use of the algorithm parameters established in the startup state. Algorithm parameters are updated to reflect long term drift characteristics based on the current environment information.

- State 3: Holdover. In this state the PRC reference signal is absent and the correction signal must be calculated using an algorithm based only on the best projection available using the parameters on exit from the normal state and the current environmental information from sensors. It may also be possible to estimate how long the output will remain within specification based on these parameters, a model of the clock physics and the environmental information.

The transition between the startup and normal states occurs when the correction signal falls below an application defined threshold. The transition between the normal and holdover states occurs when the PRC signal indicates a failure of the PRC reference signal. The transition between holdover and startup occurs when the PRC reference signal is reestablished.

The reliable generation of the presence of absence of the PRC reference signal is critical to the operation of the controller. Since this signal is typically received from networked source, the heartbeat detection techniques discussed in section 4.1 in connection with the power plant example are applicable.

4.3 Shutdown sequences

A common application requirement is for a single primary event to spawn a sequence of events which have a specific time relationship to the primary event. Often this primary event is some sort of system or component fault condition which may occur or be detected at M multiple points in the system and the spawned events may likewise occur at N multiple locations each with a different time relationship to the primary event. Whereas the power-plant example focused on detecting the absence of regular, expected events, in this section we focus on sporadic or unpredictable events and the chain of events triggered by them. PTIDES is equally well suited to specifying such chains of events and precisely controlling the timing between them, even across a networked system.

For these $M \times N$ applications, a multicast or publish/subscribe model is appropriate since this allows events with the same name to be detected and published from more than a single location and permits the interpretation to vary by recipient. If precise timing is required then the inclusion of the primary event timestamp in the message enables the recipients to meet the timing requirements independent of network and local delay and jitter, provided causality is not violated. Of course a one-to-one communication model could be used, but, except in the rather rare case where all recipients take exactly the same action with the same temporal semantics, this model requires a separate message to each recipient. This presents scaling problems and typically results in increased network traffic to the detriment of low network delay and determinism.

The requirements call for enforcing precise timing specifications on events executing on different platforms each with their own local clock. To do this, it is necessary to synchronize these local clocks to within some bounded error small enough to meet the application requirements. This can be done in networked systems by means of a protocol such as IEEE 1588 as discussed in section 2.3.

PTIDES is well suited for this type of application. An example is illustrated in Figures 9 and 10.

In many test systems, and probably in operational systems, the failure of a power supply, or another device, can cause serious damage to instrumentation and operational systems. These failures may be caused by some internal fault in the power supply or may be due to over current demand by the load. In many cases system specifications require that in the event of such a power supply failure that other power supplies and other equipment in the system be shut down in a specific order and with specific time constraints. Furthermore the order, and possibly the timing, of the system shutdown may depend on the identity of the first component to fail. This is a very common problem and typically quite expensive to implement since the solution must be embedded in the primary application without undue degradation of primary application function or timing.

This problem can be solved by the use of a named event, possibly with an attribute indicating the source, and with a timestamp indicating the time the failure was detected. The detecting device, e.g. the power supply that experienced the over current, multicasts or publishes this event. Recipient devices are preprogrammed with the correct reaction to such an event with the reaction possibly depending on the timestamp and identity attributes.

A typical test system consists of the device under test (DUT), several power supplies and other devices such as signal generators or digitizers. All communicate via a network. A typical test run is a follows:

- At turn on all equipment is powered down.

- A system controller then issues a startup command which causes the power supplies to turn on in a specific order and with specific relative turn on times. These specifications will be DUT dependent. The other devices typically power up immediately but do not generate outputs in this phase.

- After all supplies are at their target voltage the system moves to the normal or test phase in which the controller and other devices combine to execute the desired test on the DUT.

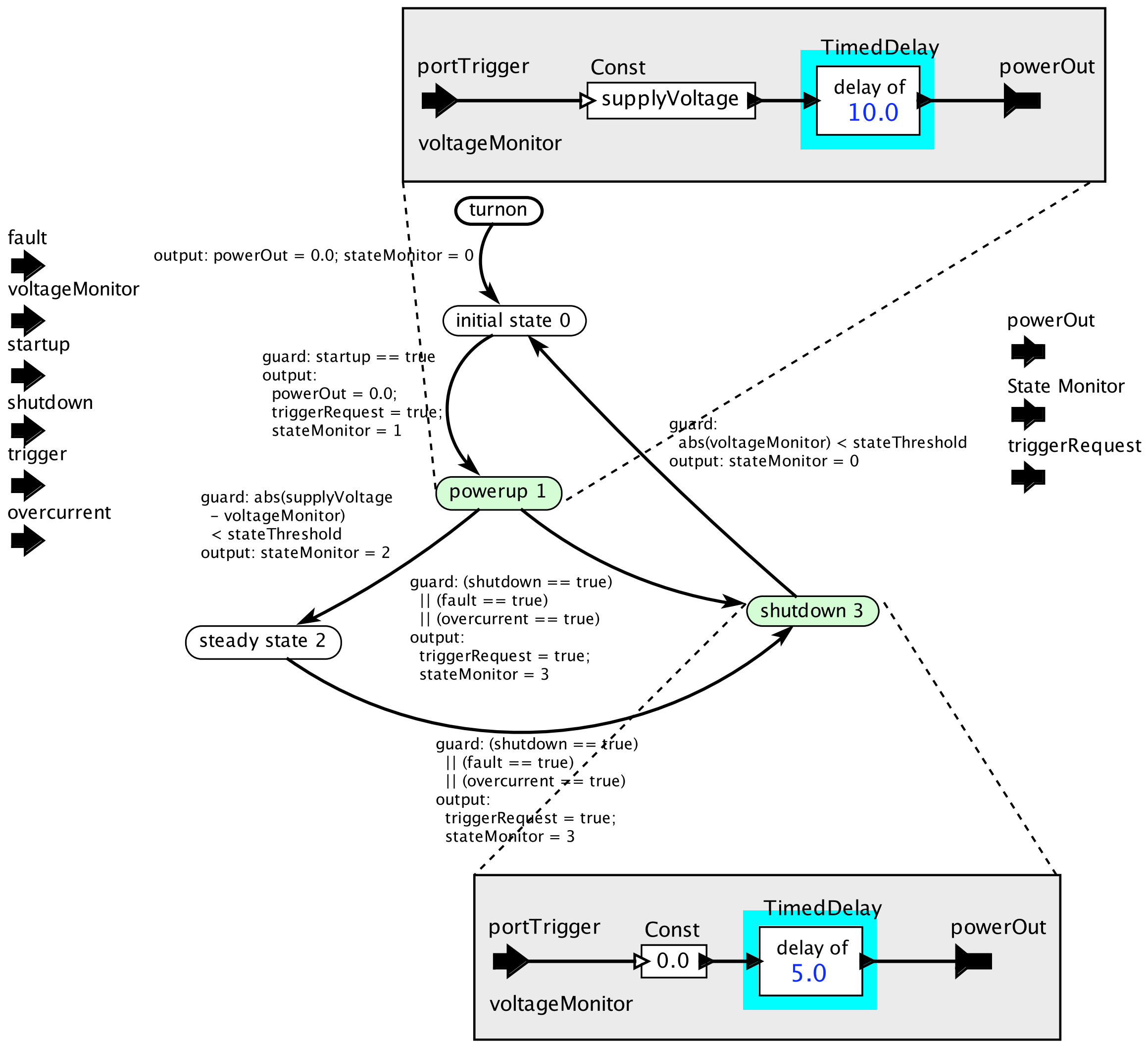

Figure 9. Power supply controller FSM

- At the conclusion of the test sequence the controller issues a shutdown command which causes the power supplies and other devices to turn off in a specific order and with specific relative turnoff times.

- If during the test a fault is detected, e.g. power supply over current, that can potentially damage either the instrumentation or the DUT the shutdown sequence is started. Since such an event is usually time critical, this function is often difficult to meet if it requires involvement of the system controller.

The modal model of Figure 9 illustrates a typical design for a controller that implements these requirements. The *shutdown* and *startup* inputs typically are generated either by a front panel or via the network from a supervisory controller. The *voltageMonitor* signal is generated elsewhere in the power supply and represents the actual output voltage of the supply. The *trigger* input is connected externally to the FSM via a feedback loop to the *triggerRequest* output of the FSM. The *triggerRequest* output is generated

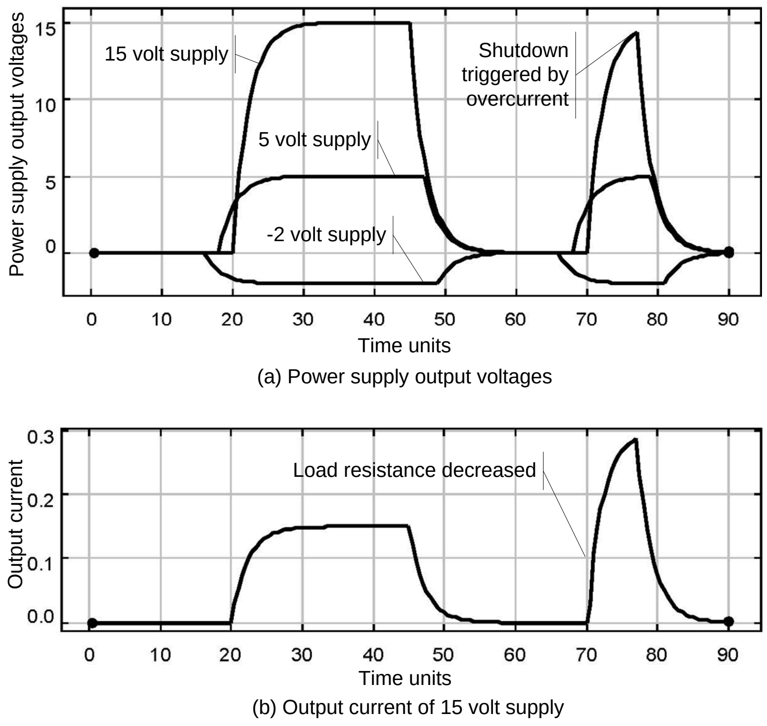

Figure 10. Power supply system outputs

during selected state transitions as shown and serves to generate an execution cycle of the modal model refinements.

The key inputs for the requirements discussed here are the *fault* or *overcurrent* signal inputs, which initiate an immediate start to the shutdown sequence from either the steady state or powerup states. The *overcurrent* signal is generated internal to the supply and is also transmitted via a multicast transmission to the *fault* input of other power supplies in the system.

Note that in the refinements of both the powerup and shutdown states the output of the appropriate *powerOut* signal, indicating the desired output voltage of the supply, are delayed by amounts that allow each supply to be configured to meet the sequence timing requirements. From the temporal semantics rules of section 3 it is clear that if a *shutdown*, *fault* or *overcurrent* input arrives at the FSM with a model time t earlier than the model time of the *powerOut* event of the powerup state, that this output will not occur, and the transition to the shutdown state will be initiated. Otherwise the transition to the shutdown state will occur while the power supply is reaching final voltage or is in steady state, thus meeting the stated application requirements. This also illustrates how the temporal semantics of an application can be adjusted or changed by placing a model delay inside a modal model, as shown in Figure 9, in which case the output can be preempted by a mode change as discussed, or outside the modal model, as illustrated by actor *delay-8* in Figure 5, in which case the output will occur, at the specified model time irrespective of the state of the modal model at that time.

The operation of this controller is illustrated in Figure 10. Figure 10 (a) shows the actual output voltages from the 15, 5, and -2 volt power supplies in the system. Figure 10 (b) shows the output current of the 15 volt supply. The delay actors in the powerup state refinements of the FSMs of the supplies delay the turn on of the supplies after receipt of a *startup* signal by 10, 8, and 6 time units respectively for the 15, 5, and -2 volt supplies. The corresponding delays after a transition to the shutdown state are 5, 7, and 9 respectively. In this example a *startup* is received by all supplies at 10 time units and a *shutdown* is received at 40 time units. As expected the times at which the various supplies begin to turn on are 16, 18, and 20 time units for the -2, 5, and 15 volt supplies. The supplies turn off in the reverse order at 45, 47, and 49 for the 15, 5, and -2 volt supplies respectively.

Following this sequence a second *startup* is received at time 60 with the resulting sequence of turn on times shown. However in this case the 15 volt supply experiences double the expected output current as shown in Figure 10 (b) resulting in an *overcurrent* signal at approximately time 72. As noted this signal is transmitted to the FSM of the 15 volt supply and as a *fault* signal to all other supplies. The resulting shutdown sequence is shown where again the supplies turn off in the reverse order from the turn on sequence.

This example illustrates several features of the PTIDES environment:

- The use of a timed delay actor within a modal model to specify the temporal behavior of the device. Because these time delay actors manipulate timestamps, the relative ordering of the resulting events is guaranteed by the PTIDES semantics.
- The use of synchronized clocks in a multi-platform system to allow FSMs and other actors in each platform to enforce system-wide temporal behavior.
- The enforcement of correspondence between model and physical time at sensors and actuators to ensure that such timing specifications are realized, subject of course to finding a feasible solution as discussed in section 3.3.
- The enforcement at platform network outputs of sending deadlines to ensure that multi-platform feasible solutions are computable.

5 Conclusion and future work

This paper reviewed Ptolemy II enhancements for several important aspects of CPS, namely PTIDES for distributed real-time systems, and modal models for multi-mode system behavior. We presented the semantics for these enhancements, and discussed related constraints. We then demonstrated the relevance of these enhancements through several examples that apply PTIDES and modal models in common industrial applications.

Our future activities include work on several components of the PTIDES framework. PTIDES relies on software components providing information about model delay they introduce. This information is captured by causality interfaces [24], and causality analysis is used to ensure that DE semantics is preserved in an execution. The precise causality analysis when modal models are allowed is undecidable in general, but we expect that common use cases will yield to effective analysis. We plan to identify subsets of modal models for which causality analysis is decidable, and study the complexity of the resulting analysis.

Another challenge is to provide feasibility analysis for the PTIDES programming model, which would allow for a static analysis of the deployability of a given application on a set of resources. As noted, feasibility considerations in multi platform designs require managing execution times and model delay actors. Research is also needed on additions to the design environment to enable designers to optimize the distribution of actors on a set of platforms to meet these temporal requirements.

A major component of our work will be the design of a distributed execution platform for PTIDES. The code generator integrated within the Ptolemy II environment will generate C code from PTIDES models and glue them together with the preexisting software components to produce executable programs for each of the platforms in the network. The code will be executed in the context of the PtidyOS runtime environment that can be considered as a lightweight operating system with PTIDES semantics.

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

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

[4] P. Derler, E. Lee, and S. Matic. Simulation and implementation of the ptides programming model. In *International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, Vancouver, Canada, 2008. IEEE.

[5] J. C. Eidson. *Measurement, Control, and Communication Using IEEE 1588*, pages 194–200. Springer, London, 2006.

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

[7] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[8] IEEE Instrumentation and Measurement Society. 1588: IEEE standard for a precision clock synchronization protocol for networked measurement and control systems. Standard specification, IEEE, July 24 2008.

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

[10] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[11] E. Lee, S. Matic, S. Seshia, and J. Zou. The case for timing-centric distributed software. In *International Workshop on Cyber-Physical Systems*, Montreal, Canada, 2009. IEEE.

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

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

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

[15] X. Liu and E. A. Lee. CPO semantics of timed interactive actor networks. *Theoretical Computer Science*, 409(1):110–125, 2008.

[16] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In *Real-Time: Theory and Practice, REX Workshop*, pages 447–484. Springer-Verlag, 1992.

[17] Z. Manna and A. Pnueli. Verifying hybrid systems. *Hybrid Systems*, pages 4–35, 1992.

[18] R. Olfati-Saber, J. A. Fax, and R. M. Murray. Consensus and cooperation in networked multi-agent systems. *Proceedings of the IEEE*, 95(1):215–233, 2007.

[19] O.M.G. U.m.l. specification Version 1.3. *Object Management Group*, 1999.

[20] P. Ramadge and W. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.

[21] D. Rosselot. Simple, accurate time synchronization in an ethernet physical layer device. In *ISPCS: International IEEE Symposium on Precision Clock Synchronization for Measurement, Control and Communication. Vienna*, 2007.

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

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

[24] Y. Zhou and E. A. Lee. Causality interfaces for actor networks. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–35, 2008.

[25] J. Zou, J. Auerbach, D. Bacon, and E. Lee. Ptides on flexible task graph: Real-time embedded system building from theory to practice. In *Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Dublin, Ireland, 2009. ACM.

[26] J. Zou, S. Matic, E. Lee, T. Feng, and P. Derler. Execution strategies for ptides, a programming model for distributed embedded systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, San Francisco, CA, USA, 2009. IEEE.