Distributed Real-Time Software for Cyber-Physical Systems

John C. Eidson, *Life Fellow, IEEE,* Edward A. Lee, *Fellow, IEEE,* Slobodan Matic, *Member, IEEE,* Sanjit A. Seshia, *Member, IEEE,* and Jia Zou, *Student Member, IEEE*

***Abstract*—Real-time embedded software today is commonly built using programming abstractions with little or no temporal semantics. This paper addresses this problem by presenting a programming model called PTIDES that serves as a coordination language for model-based design of distributed real-time embedded systems. Specifically, the paper describes the principles of PTIDES, which leverages network time synchronization to provide a determinate distributed real-time semantics. We show how PTIDES can function as a coordination language, orchestrating components that may be designed and specified using different formalisms. We show the use of this environment in the design of interesting and practical cyber-physical systems, such as a power plant control system.**

I. Introduction

In cyber-physical systems (CPS) the passage of time becomes a central feature of system behavior — in fact, it is one of the important constraints distinguishing these systems from distributed computing in general. Time is central to predicting, measuring, and controlling properties of the physical world: given a 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 states. When that program is integrated into a system with physical dynamics, this makes principled design of the entire system difficult. Moreover, the disparity between the dynamics of the physical plant and the program potentially leads to errors, some of which can be catastrophic.

The challenge of integrating computing and physical processes has been recognized for some time, motivating the emergence of hybrid systems theories. 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 though traditional computer science concepts in distributed systems emphasize asynchrony, when these concepts are adapted to control problems, researchers often make the assumption of the oracle t. For example, in [21], consensus problems from computer science are translated into control systems formulations, but with the introduction of this global binding notion of time. In networked software implementations, such a uniform notion of time cannot be precisely realized. Time triggered networks [12] can be used to approximate a uniform model of time, but the analysis of the dynamics has to include the imperfections.

Although design of real-time software is not a new problem, there exist trends with a potential to change the landscape. Model-based design [11], for example, has caught on in industrial practice, through the use of tools such as Simulink, TargetLink, and LabVIEW. Domain-specific modeling languages are increasingly being used because they tend to have formal semantics that experts can use to describe their domain constraints. For CPS, models with temporal semantics are particularly natural to system designers. An example of such a language is Timing-Augmented Description Language [10], a domain-specific language recently developed within the automotive initiative AUTOSAR.

A formal semantics enables safety verification of individual components and helps with integration of components into systems. Safety and dependability are emphasized in the synchronous-reactive languages, particularly Esterel and SCADE [1], used primarily in safety critical applications. An example that addresses integration and mutual consistency issues of different modeling languages is the UML extension called MARTE (Modeling and Analysis of Real-Time and Embedded Systems). Some frameworks, like BIP [2], are component frameworks based on formal verification methods, and they address both issues. BIP focuses on compositional verification of properties such as deadlock freedom and relies on priorities to model scheduling policies. As far as we know, it has not been used to address modeling and design problems for components with explicit timing requirements. The model-based design approach we propose in this paper borrows sound fixed-point semantics from the synchronous languages, but is more flexible and concurrent.

To control timing in embedded software, programmers typically use platform-specific system timers. However, de-

The authors are with the EECS Department of UC Berkeley. Contact them at {eidson, eal, matic, sseshia, jiazou}@eecs.berkeley.edu.

This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET), #0931843 (ActionWebs), and #1035672 (CSR-CPS Ptides)), 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 Multiscale Systems Center (MuSyC), one of six research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program, and the following companies: Bosch, National Instruments, Thales, and Toyota. The fourth author was also supported in part by NSF grant #0644436 and an Alfred P. Sloan Research Fellowship.

Manuscript received February 28, 2011.

sign of a system should be as independent of implementation details as possible, to allow for portability and design space exploration. To this end, we have previously proposed a programming model called PTIDES (*programming temporally-integrated distributed embedded systems*, pronounced "tides") [29]. PTIDES structures real-time software as an interconnection of components communicating using timestamped events. Ptides leverages network time synchronization to provide a coherent global temporal semantics in distributed systems. With PTIDES, application programmers specify the interaction between the control program and the physical dynamics in the system model, without the knowledge of low-level details such as timers.

Prior work on PTIDES includes a study of the semantic properties of an execution model that permits out of order processing of events without sacrificing determinacy and without requiring backtracking [32]. The same work introduces feasibility analysis for PTIDES programs, a problem that is only partially solved. PTIDES has been used to coordinate real-time components written in Java [31] and C [33], where the "glue code" is automatically generated from PTIDES models. A simulator has been developed that supports joint modeling of PTIDES controllers, physical plants, and communication networks [5], [33].

The goal of this paper is to review the principles of PTIDES and to demonstrate its usefulness for time-critical CPS applications. We first explain how PTIDES models provide deterministic processing of events. Building on [32], in this paper (Sec. II-C), we consider the networked case, deriving bounds for deadlines for event network transmissions. Then we illustrate how to specify timed reactions to events in PTIDES models. This results in identical traces from model simulation and execution of automatically generated code.

In order to account for different modes of operation, modal models have been widely used in embedded system design [8]. By "modal models," we mean the use of state machines that define modes of operation and govern the switching between modes [16]. In this paper, we introduce the use of modal PTIDES models. This combination enables precise specification of timed mode transitions.

This paper is organized as follows. First, section II discusses the PTIDES design environment, which enables a programmer to first model and simulate the design and then implement it through a target-specific code generator. This environment extends Ptolemy II [7] to realize a coordination language for the design of distributed real-time embedded systems and to provide a co-simulator for embedded software, physical plants, and networks. Section III then explains 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 a detailed application example in section IV. We conclude in section V.

II. Design Environment

A. PTIDES Workflow

Fig. 1 shows our workflow, which includes modeling and analysis, code generation and implementation, and co-simulation with physical plants and networks. The PTIDES

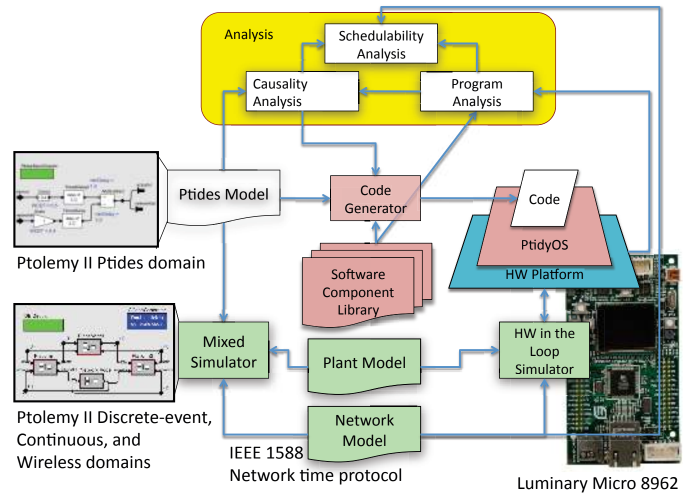


Fig. 1. PTIDES Code Generation and Analysis Workflow

design environment is an extension of the Ptolemy II framework, which supports modeling, simulation, and design of systems using mixed models of computation. The physical part of the system can be modeled in the continuous domain. The simulation can be instantiated with different ODE solvers that suit diverse time scales present in the model. PTIDES models define the functional and temporal interaction of distributed software components, the networks that bind them together, sensors, actuators, and physical dynamics. Simulation can be done on such models, such that functionality and timing can be tested. In particular, to get a picture of the temporal behavior of a particular implementation of a model, each actor can be endowed with a platform-dependent execution time, and simulation can be performed to determine whether real-time deadlines can be met for a given set of inputs.

As we will discuss in detail in the next section, PTIDES has a formal operational semantics, and the corresponding domain in Ptolemy II defines a suitable graphical syntax. PTIDES defines the interaction between components called actors, but the detailed functionality of the actors is not given in PTIDES. It is given instead either in a conventional programming language (typically C for embedded systems) or as a submodel. Thus, PTIDES is a coordination language [22].

Our implementation of PTIDES leverages the Ptolemy II code generation framework to generate target-specific glue code from the PTIDES model. The generated executable integrates a lightweight real-time microkernel that we call PtidyOS. Its real-time scheduler implements PTIDES semantics and therefore preserves the timed semantics given in the PTIDES model. Like TinyOS [18], PtidyOS is a set of C libraries that glues together application code, which then runs directly ("bare-metal") on the hardware platform. Currently, our code generation framework supports a Luminary Micro board as our target platform. It is of course possible to realize a PTIDES execution environment on top of a conventional operating system or a real-time operating system (RTOS). As of this writing, we have done this only on a real-time variant of Java [31]. In fact, there is no impediment to having a PTIDES deployment that mixes bare metal, RTOS, and conventional

OS platforms, although overall system tolerances (like clock synchronization precision) may end up being determined by the worst case platforms.

The goal of PTIDES timed semantics is to capture system timing requirements. It is then necessary to check that the generated code running on the target platform will be able to meet these requirements. There are several approaches to answer this schedulability question. One approach is detailed in the workflow diagram, which shows schedulability analysis, the goal of which is to statically determine whether all deadlines can be met. This analysis requires platform-specific information such as worst-case execution time (WCET) for each actor and an event model for each sensor and network input (i.e., the rate and pattern of event streams arriving at sensors and network input ports). The values of these parameters are not always known definitively, so schedulability analysis may be approximate. In fact, it is easy to show that the schedulability analysis problem is undecidable in general, so approximations will be necessary.

In this paper, we describe a co-simulation approach to schedulability analysis. The programmer annotates each actor with a WCET (which may be approximated using program analysis tools [23], [24]) and specifies the input event models. Simulation then lends insight into the real-time behavior of the system, building confidence in the design. Deadline misses are recorded during simulation. GAMETIME, a systematic measurement-based approach to execution time analysis [24], [25], [26], can be used not only for predicting the WCET, but also to generate test cases that are reasonably comprehensive for use in simulation.

Though we have carried out modeling, simulation, and implementation of a number of small examples using the PTIDES simulator and PtidyOS, in this paper we only focus on the modeling and simulation aspects in order to illustrate how one can program distributed cyber-physical systems using explicit timing constraints.

B. Model Time and Physical Time

PTIDES is based on discrete-event (DE) systems [3] [28], which provide a model of time and concurrency. We specify DE systems using the actor-oriented approach [14]. In this case, the actors are concurrent components that exchange time-stamped events via input and output ports. The time in time stamps 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*. 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 been shown to integrate well with models of continuous dynamics [17]. The purpose of this paper is not to study its rigorous and determinate semantics. For that an interested reader is referred to [19] and [13].

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 the specification of both functional and temporal properties of deployable cyber-physical systems. There are three key constraints that define the relationship between model time and physical time: 1) sensors produce events with timestamp τ at physical time $t \geq \tau$; 2) actuators receive events with timestamp τ at physical time $t \leq \tau$, and 3) network interfaces act as actuators when sending messages and as sensors when receiving them. We explain these constraints in detail below.

The basic PTIDES model is explained by referring to Figure 2, 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 produces an output event with the same time stamp as the input event that triggers the computation. Those events are merged in time stamp order by a component 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 time stamp 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 time stamp to meet deadlines. Either PtidyOS or the design of the actuator itself ensures that the actuation affects the physical world at a time *equal to* the event time stamp. Therefore the deployed system exhibits the *exact temporal behavior specified in the design* to within the limits of the accuracy of clock synchronization between platforms and the temporal resolution of the actuators and clocks.

In Figure 2, 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 Local Event Source, and modified by the component Computation4. The Merge component can inject commands to the actuator that originate from either the local event source or from the network. The commands are merged in order of their time stamps. Notice that the top input to the Merge component comes from components that get inputs from sensors on the remote platforms. The sensor components produce on their output ports time-stamped events. Here, the PTIDES model imposes a second relationship between model time stamps and physical time. Specifically, when a sensor component produces a time-stamped output event, that time stamp 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 time stamp of the event. Consider Platform 1 in Figure 2 as an example. When an event of time stamp τ is to

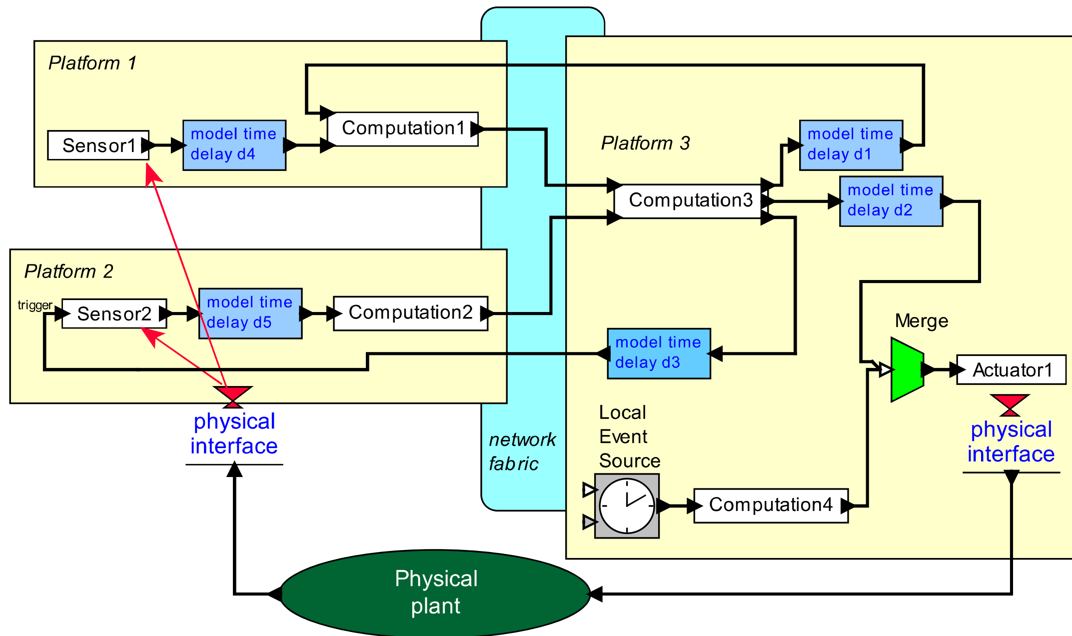


Fig. 2. Prototypical CPS

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 time stamp, but for our purposes here, it is sufficient that there be a deadline, and that the deadline be a known function of the time stamp. We discuss options for this function in the following subsection.

C. Event Processing in PTIDES

Under benign conditions [13], 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 2. 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 time stamp 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.

There are two key assumptions made in PTIDES. First, distributed platforms have real-time clocks synchronized with bounded error. 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 such as IEEE 1588 [9] can deliver real-time clock precision on the nanosecond order.

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, 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 [12], FlexRay [20], and CAN buses [27]). One of the goals of our research is to use PTIDES on less constraining networking architectures, e.g. to allow more flexibility in processing aperiodic events. In the time-triggered architectures, all actions are initiated by the computer system at known time instants. In our approach, events coming from the environment are allowed and are treated deterministically. Here 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 communication delays) are accepted, together with deadlines for network interfaces and actuators, local decisions can be made to deliver events in Figure 2 without compromis-

ing DE semantics. Specifically, in Figure 2, 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 time stamps. In particular, in this figure, assume that the only components that manipulate time stamps are the components labeled *model time delay* d_i. These components accept an input event and produce an output event with the same data but with a time stamp incremented by d_i.

Assume we have an event e with time stamp 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 time stamp 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 time stamps by d_2. Thus, we are sure e is safe to process when no events from the network will arrive at Platform 3 with time stamps 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 time stamps less than $\tau - d_2$ will be sent over the network by the physical time $\tau - d_2$. Consequently, all events with time stamp 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. In other words, to ensure that the processing of an event obeys DE semantics, at run time, the only test that is needed is to compare time stamps to physical time with an offset (in the previous example, the offset is $-d_2 + n + s$). This operation, thus, takes constant time per event. Notice, if we assume the model is static (components are not added during runtime and connections are not changed); minimum bounds on model time delays (d_i's) for components are known statically; and the upper bounds for sensor processing times, network delays, and network synchronization errors are known, then the offsets can be calculated statically using a graph traversal algorithm which takes linear time in the number of model actors.

The expression presented in the previous paragraph was derived under the assumption that the deadline at a network interface for the transmission of an event with time stamp τ is equal to the time stamp, $D(\tau) = \tau$. However, given the PTIDES program with known model time delay values for all actors, there is actually a range of possible network interface deadlines, where the range is defined by constraints imposed at sensor-actuator boundaries. The lower bound of this range is determined by the constraint that sensor events are produced at physical time greater than the time stamp. In particular, in Figure 2, the network interface deadline of an event with time stamp τ at the output of the Computation1 actor cannot be lower than $D_l(\tau) = \tau - d_4$, because this is the earliest physical time a sensor event with time stamp $\tau - d_4$ can be detected at Sensor1. This bound assumes zero execution time of the Computation1 actor. The upper bound of the network interface deadline range is determined by the constraint that actuator events are received at physical time smaller than the time stamp. In Figure 2 this upper bound for the network interface at the output of the Computation1 actor is determined by Actuator1 on the receiving platform Platform3 and the network delay bound n and clock synchronization error bound of s. This bound is $D_u(\tau) = \tau + d_2 - n - s$ because an event at the output of the Computation1 actor with time stamp $\tau_1 = \tau - d_2$ could become available at the upper input of the Merge actor no later than at physical time $D_u(\tau_1) + n + s = D_u(\tau - d_2) + n + s = ((\tau - d_2) + d_2 - n - s) + n + s = \tau$, which is the upper bound on physical time an event with time stamp τ should be received by Actuator1. This bound assumes zero execution time of Computation3 actor. So, the theoretical bounds on network interface deadline in this case are $D_l(\tau) = \tau - d_4$ and $D_u(\tau) = \tau + d_2 - n - s$. Our assumption that the deadline equals the time stamp τ makes the analysis in next subsections particularly simple, so for the purposes of this paper we proceed with that. A possible consequence of this assumption on performance will be addressed in our future work.

Notice, the *safe-to-process* expression presented in the above paragraphs only determines whether an event can be causally affected by another event from outside of the platform. However, there might exist another event inside the platform that can render the event of interest unsafe. A simple solution for this problem is to maintain an ordered queue for all system events. We then enforce that only the event of the smallest timestamp can be processed. Since for any other event within the platform to render the event of interest unsafe, that other event must have smaller timestamp. By using this scheme, combined with the safe-to-process expression earlier, the safe-to-process scheduling analysis is complete. Notice this algorithm is the same as Strategy C defined in [32]. That work formally presents PTIDES execution model together with a general strategy. This strategy takes time that is linear in the number of events in the queue.

Note that the distributed execution control of PTIDES introduces another valuable form of robustness in the system. For example, in Figure 2, 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 [4]. 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 time stamp. Time synchronization with bounded error helps to give such mechanisms a rigorous semantics.

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). Moreover, since execution of a PTIDES model carries time stamps 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.

III. Temporal Semantics in PTIDES

PTIDES semantics is fully described in [29] and [32], and is based on a tagged-signal model [15]. 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, $F_a : S^I \rightarrow S^O$. 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 (time stamp, 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, time stamps typically have no connection with real time, and can advance slower or faster than real time [28].

Actors are permitted to modify the time stamp and most commonly will modify the model time member, i.e. the time stamp, 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 $F_\delta(s) : S \rightarrow S$ where for each $s \in S$ we have $F_\delta(s) = \{(t + \delta, v) \,|\, (t, v) \in s\}$. 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 3. In this example we assume $F_a(s) = \{(t, 2 * v) \,|\, (t, v) \in s\}$; 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 time stamp 25 seconds is the physical time at the time of sensing. The delay actor increments the model time by 10 and the scale actor doubles the value from 15 volts to 30 volts. In both cases the actuator receives an event (35 seconds, 30 volts), which 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, all observable effects with models (a) and (b) are identical, regardless of computation times and scheduling decisions.

As mentioned earlier, the Ptides simulator allows the simulation of execution time. The model in Fig. 3 is simulated and a visualization of the physical times at which system events occur is presented in Figure 4. Events that occur in the top actor paths are plotted on top, and the one on the bottom bottom. A connected line indicates one actor on that path is executing during that period of time, and a dot indicates that an event is produced at that physical time.

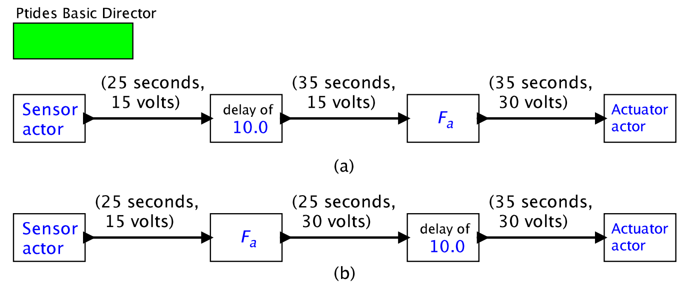


Fig. 3. Linear combination of actors

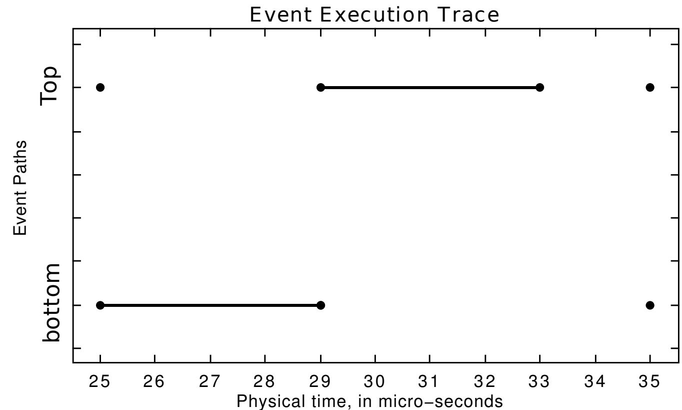


Fig. 4. Visualization of the Physical Times at which Events Occur

Both paths receive inputs at physical time $25secs$. Recall the Ptides scheduler schedules by the order of event's timestamps. If there are multiple events with the same timestamp, then it randomly picks one to process. In this case, the one on the bottom path is processed first. The execution time of both F_a actors are set to $4secs$. Thus the bottom F_a actor executes from $25secs - 29secs$, and produces an event at $29secs$. Immediately following that, the time delay actor on the top path fires. This actor introduces a *model time* delay of 10, while taking $0secs$ physical time to execute. Thus we see an output of timestamp 35 on the top path at physical time $29secs$. Then the bottom time delay actor fires. This actor performs the same functionality as the delay actor on top, and it produces an output event of timestamp 35 at physical time $29secs$ (Note two events are produced at this physical time, but only one dot is shown in the figure). F_a on the top path then fires, produces an event with timestamp 35 at physical time $33secs$. Finally, since deadlines are met at both actuators, actuation events are produced at physical time 35. Notice the processor is idle from time $33secs - 35secs$, since no other sensor events have occurred, and the actuator is simply waiting to actuate when physical time equals the events' timestamps.

Modal Models. The use of modal models is well established both in the literature, for example Statecharts [8], UML, and in commercial products such as Simulink/Stateflow from The MathWorks. Here, the term *modal* refers to "modes of operation," where modal models extend finite-state machines by associating with each state of an FSM a behavior given by a submodel. The semantics of modal models, and particularly their handling of temporal behavior, is described in [16].

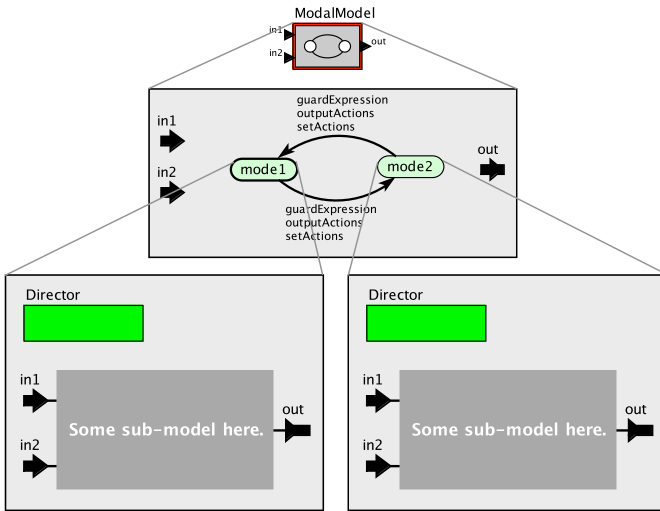


Fig. 5. General pattern of a modal model with two modes, each with its own refinement

The time-centric modal models discussed here are particularly useful for the specification of modes of operation in a CPS as we explain in section IV-A. Our style for modal models follows the pattern shown in Figure 5. 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.

Modal models introduce additional temporal considerations into a design. This is especially true for modal models that modify the time stamp of a signal. 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 time stamp τ, it first presents those input events to the refinement of the current state i. That refinement may, in reaction, produce output events with time stamp τ.
- If any of input events have an effect within the refinement at a later time stamp $\tau' > \tau$, that effect is postponed. The modal model is invoked again at time stamp 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 time stamp 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 time stamp order in accordance with DE semantics.

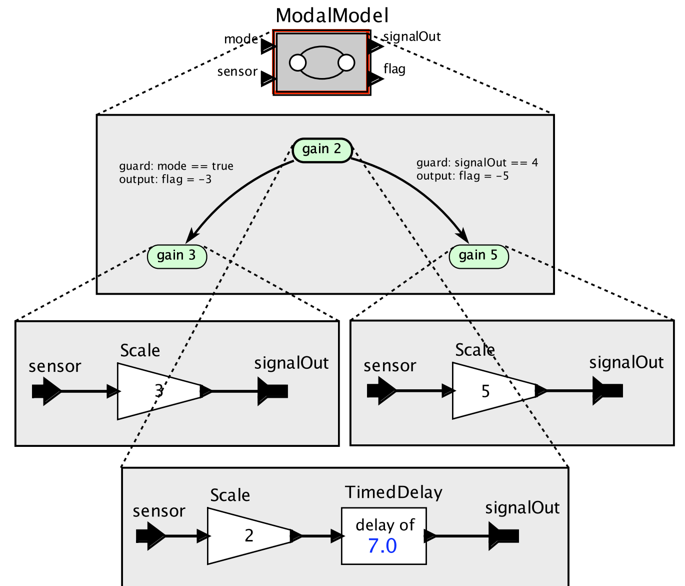


Fig. 6. Simple time-sensitive modal model

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 6. 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 $(t, v) = (10, 30)$ 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 that time 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.

IV. Application Study

PTIDES can be used to integrate models of software, networks, and physical plants. This is achieved by adopting the

fixed-point semantics that makes it possible to mix continuous and discrete-event models [17]. A practical consequence is to enable CPS co-design and co-simulation. 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.

The inclusion of a network into an embedded system introduces three principal complications in the design of 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 II this is often based on a time-slotted network protocol but can also be based on a clock synchronization protocol such as IEEE 1588 [9].
- 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. The remainder of this section illustrates the following features of the PTIDES design environment:

- The use of time-based models of the plant in testing controller implementations of power plants.
- The use of a modal model to specify the temporal behavior of the operational modes of a device.
- The use of time-based detection of missing signals, based on local clocks, to drive mode changes in the operation 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.
- 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
- The enforcement of deadlines for sending events over the network to ensure correct (w.r.t. DE semantics) event processing on receiving platforms.

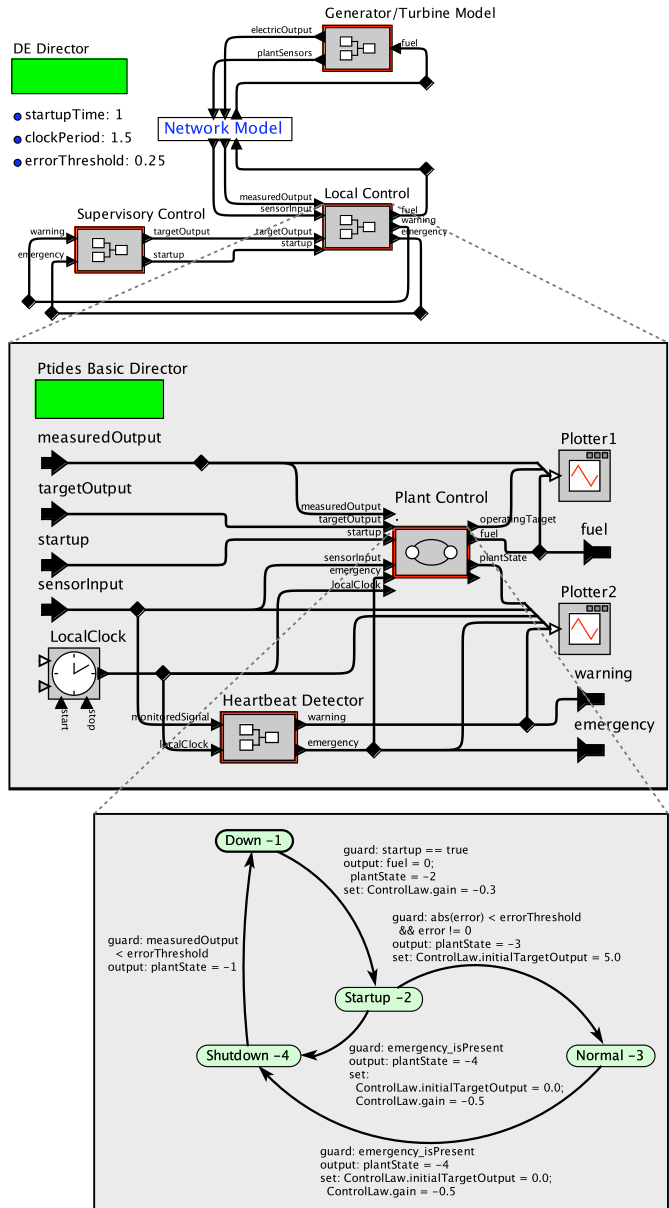


Fig. 7. Model of a small power plant. This model can be opened, run, and even modified by clicking on the figure above (if you are reading this on a computer), or by going to http://ptolemy.org/PowerPlant on a Java-capable machine.

A. 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 turbine speed control loop and the system reaction time to failures. The loop time is relatively long but for serious failures the fuel supply to the turbine must typically be reduced within a few milliseconds.

A typical power plant can involve sampling of up to 3000 nodes comprising monitoring equipment separated by several hundred meters. Since the purpose of the monitored data is to make decisions about the state of the physical plant, 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 synchronized real-time clocks in the separate platforms.

Figure 7 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 of 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 we use rather simple versions. The model in Figure 7 also includes a local controller, which is expanded showing two main components, a Heartbeat Detector and Plant Control block. A power plant, like many CPS, can be characterized by several modes of operation each of which can have different time semantics. This is reflected in the design of the Plant Control block that is implemented with a four state modal model based on the discussion of section III . 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 actuators on the generator/turbine. Because both the controller sensor input data and the resulting fuel control signal sent to the actuators are time stamped, the designer is able to use PTIDES construct to precisely specify the delay between sensors and actuators. Furthermore as described earlier executable code generated from the PTIDES models shown here, forces these time stamps to correspond to physical time at both sensors and actuators thus ensuring deterministic and temporally-correct execution meeting the designed specifications *even across multiple platforms linked by a network*.

To further aid the designer these models are executable. For example, the plots generated by the two Plotter actors in Figure 7 are shown in Figure 8 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.

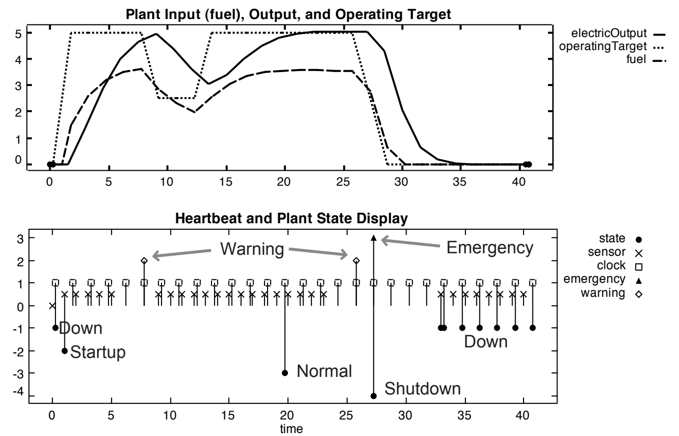


Fig. 8. Power plant output and events

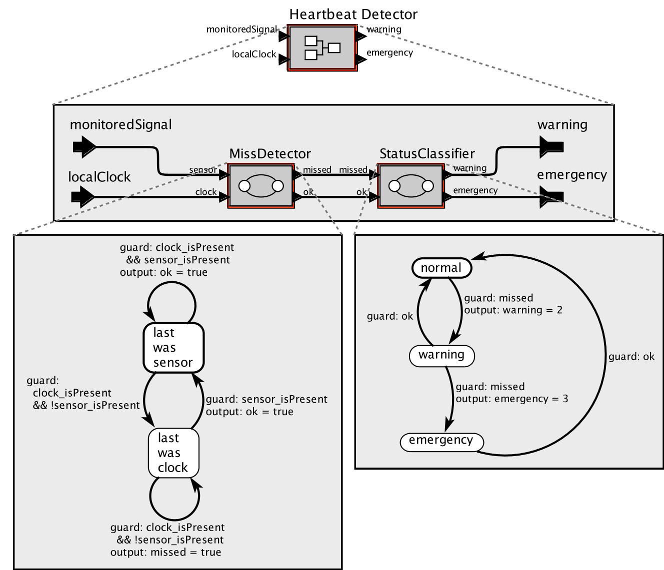


Fig. 9. Heartbeat detector that raises alarms

The time stamps 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 in the way it uses to send *warning* and *emergency* events. As shown in Figures 7 and 8, the Generator/Turbine Model sends (time-stamped) sensor readings over the network to the Local Control component. These sensor events are shown with "x" symbols in Figure 8. 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 9 and illustrates another use of the

modal model temporal semantics of section III. 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 this message will be delivered to this component in time-stamp order, *even when the messages and their time stamps originate on a remote platform elsewhere in the network*. This MissDetector component issues a missed event if two successive local clock events arrive without an intervening sensor event. The missed event will have the same time stamp 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 consecutive 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.

If the generated code correctly performs a comparison between timestamp and physical time, as explained in section II-C, it is guaranteed that the implementation will behave exactly like the simulation, given the same time-stamped inputs. Moreover, it is easy to integrate a simulation model of the plant, thus evaluating total system design choices well before system integration.

A detailed discussion of the design issues illustrated in this example for an actual commercial power plant control system is found in [6]. In the following section, we discuss other PTIDES applications such as power supply shutdown sequencing. In many distributed systems such as high speed printing presses, when an emergency shutdown signal is received, one cannot simply turn off power throughout the system. Instead, a carefully orchestrated shutdown sequence needs to be performed. During this sequence, different parts of the system will have different timing relationships with the primary shutdown signal. As presented below, this relationship is easily captured in the timed semantics of PTIDES.

B. 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.

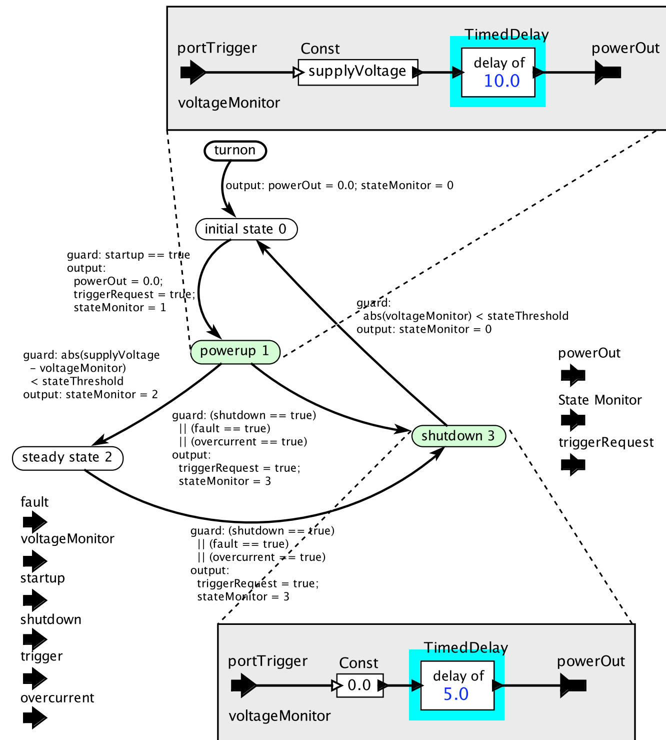
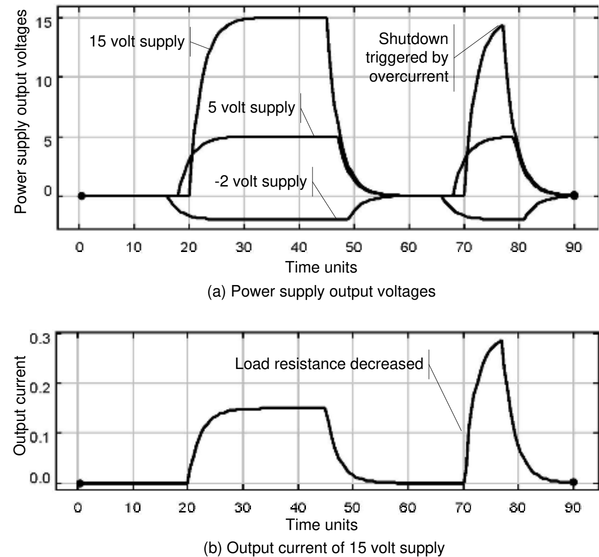


Fig. 10. Power supply controller FSM



(a) Power supply output voltages

(b) Output current of 15 volt supply

Fig. 11. Power supply system outputs

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 time stamp in the message enables the recipients to meet the timing requirements independent of network and local delay and jitter, provided causality is not violated.

An example is illustrated in Figures 10 and 11.

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. In many cases system specifications require that in the event of such a failure that other equipment in the system be shut down in a specific order and with specific time constraints relative to the time of the failure event. 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. Indeed it is common practice to implement the failure response by means of dedicated circuits and cables between components to avoid introducing complicating software into the system.

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

The modal model of Figure 10 illustrates a typical design for a controller that implements a typical system. This design illustrates another use of the modal model of section III and the modification of model time stamps using the delay actor 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 during selected state transitions as shown and serves to generate an execution cycle of the modal model refinements.

The key inputs for the failure response mechanism 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 III 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 10, in which case the output can be preempted by a mode change as discussed, or outside the modal model, 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 11. Figure 11 (a) shows the actual output voltages from the 15, 5, and -2 volt power supplies in the system. Figure 11 (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 11 (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.

V. Conclusion

This paper has described modeling techniques for several important aspects of CPS design and deployment, specifically focusing on the PTIDES model for distributed real-time systems and on modal models for multi-mode system behavior. The timed semantics of PTIDES allows us to specify the interaction between the control program and the physical dynamics in the system model, largely independent of underlying hardware details. Because of this independence, PTIDES models are 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.

Of course, in any real system, these constraints may be violated due to unanticipated events or system faults. Hence, although PTIDES removes a great deal of uncertainty, it does not eliminate the need to make systems adaptive. By combining PTIDES with modal models, we have illustrated timed mode transitions, which can be used to build in adaptive behaviors. For example, modal models enable time-based detection of missing signals, which could be due to system faults, and mode changes to adapt to those faults.

In order to deploy PTIDES, certain requirements must be met. On a distributed platform, clocks must be synchronized so that there is a known bound on the clock error. That is, they cannot have unbounded drift. If the bound on the error is large, then the latency from a sensor on one platform to an actuator on another will be increased. This tradeoff between latency and clock synchronization precision is quantified by PTIDES. In addition, networks must have bounded latencies, and the bounds must be known. The safe-to-process analysis

of PTIDES gives us a rigorous way to evaluate the tradeoff between sensor-to-actuator latencies and network latencies. In particular, this analysis gives us a precise measure of the cost of network variability, as measured by increased latency from sensors to actuators. Because PTIDES provides determinate semantics, *variability* in clock synchronization and network latencies has no visible effect in the physical part of a CPS. Only the bound has an effect, and that effect is a static end-to-end latency between sensors and actuators.

PTIDES can be implemented entirely in software with off-the-shelf sensors and actuators. To take full advantage of PTIDES, however, and to reduce latencies to smallest achievable, requires hardware support. Network time synchronization can be made much more precise with hardware assistance than with pure software implementations. Moreover, if sensor hardware puts time stamps onto measurements, these time stamps can be much more precise than what we would get if the time stamps are added in software. As a consequence, much tighter tolerances and lower end-to-end latencies become realizable.

Considerable work remains to be done on the PTIDES framework. For example, PTIDES relies on software components providing information about model delay that they introduce. This information is captured by causality interfaces [30], 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. Another challenge is to provide schedulability analysis for a broad class of models. This would allow for a static analysis of the deployability of a given application on a set of resources. Our prototype implementation of PTIDES is also incomplete as of this writing. The simulator supports models of distributed systems, but our code generator and runtime kernel (PtidyOS) so far only support single-platform interactions with a plant.

REFERENCES

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

[2] S. Bliudze and J. Sifakis. The algebra of connectors: structuring interaction in bip. In *EMSOFT*, pages 11–20. ACM, 2007.

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

[4] 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.

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

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

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

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

[9] 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.

[10] M. Jersak. Timing model and methodology for autosar. In *Elektronik Automotive. Special issue AUTOSAR*, 2007.

[11] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, 2003.

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

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

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

[15] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, 1998.

[16] E. A. Lee and S. Tripakis. Modal models in Ptolemy. In *3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (EOOLT)*, volume 47, pages 11–21, Oslo, Norway, 2010. Linköping University Electronic Press, Linköping University. Available from: http://chess.eecs.berkeley.edu/pubs/700.html.

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

[18] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and David C. The emergence of networking abstractions and techniques in tinyos. In *First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004)*, 2004.

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

[20] R. Makowitz and C. Temple. FlexRay-a communication network for automotive control systems. In *2006 IEEE International Workshop on Factory Communication Systems*, pages 207–212, 2006.

[21] 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.

[22] G. Papadopoulos and F. Arbab. Coordination models and languages. In M. Zelkowitz, editor, *Advances in Computers - The Engineering of Large Systems*, volume 46, pages 329–400. Academic Press, 1998.

[23] R. Wilhelm et al. The determination of worst-case execution times — overview of the methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3), 2008.

[24] S. A. Seshia and A. Rakhlin. Game-theoretic timing analysis. In *Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 575–582, 2008.

[25] S. A. Seshia and A. Rakhlin. Quantitative analysis of systems using game-theoretic learning. *ACM Transactions on Embedded Computing Systems (TECS)*, 2011. To appear.

[26] S. A. Seshia and J. Kotker. GameTime: A toolkit for timing analysis of software. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 388–392, March 2011.

[27] K. Tindell, H. Hansson, and A.J. Wellings. Analysing real-time communications: Controller area network (CAN). In *Proceedings 15th IEEE Real-Time Systems Symposium*, pages 259–265. Citeseer, 1994.

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

[29] 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.

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

[31] J. Zou, J. Auerbach, D. Bacon, and E. A. 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.

[32] J. Zou, S. Matic, E. A. Lee, T. H. 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)*, pages 77–86, San Francisco, CA, USA, 2009. IEEE.

[33] Jia Zou. From ptides to ptidyos, designing distributed real-time embedded systems. PhD Dissertation Technical Report UCB/EECS-2011-53, UC Berkeley, May 13, 2011 2011. Available from: http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-53.html.

John Eidson received his BS and MS degrees from Michigan State University and his PhD. Degree from Stanford University, all in electrical engineering. He has worked at the Central Research Laboratories at Varian Associates, the Hewlett-Packard Company, and Agilent Technologies. He has worked on a variety of projects including analytic instrumentation, electron beam lithography, and instrumentation system architectures and infrastructure. He was heavily involved in the IEEE 1451.2 and IEEE 1451.1 standards and was an active participant in the standards work of the LXI Consortium. He is the chairperson of the IEEE 1588 standards committee. He is a life fellow of the IEEE, a recipient of the 2007 Technical Award of the IEEE I&M Society, and a co-recipient of the 2007 Agilent Laboratories Barney Oliver Award for Innovation. He is currently a visiting scholar at the University of California at Berkeley.

Edward A. Lee is the Robert S. Pepper Distinguished Professor and former chair of the Electrical Engineering and Computer Sciences (EECS) department at U.C. Berkeley. His research interests center on design, modeling, and simulation of embedded, real-time computational systems. He is a director of Chess, the Berkeley Center for Hybrid and Embedded Software Systems, and is the director of the Berkeley Ptolemy project. He is co-author of six books and numerous papers. He has a BS from Yale University (1979), SM from MIT (1981), and PhD from UC Berkeley (1986). From 1979 to 1982 he was a member of technical staff at Bell Labs. He is a co-founder of BDTI, Inc., where he is currently a Senior Technical Advisor.

Slobodan Matic is a Postdoctoral Scholar with the Electrical Engineering and Computer Sciences department at U.C. Berkeley. His research interests are primarily in the area of Distributed and/or Real-Time Systems. He holds BS degree from University of Belgrade and PhD from UC Berkeley.

Sanjit A. Seshia is an Associate Professor in the Electrical Engineering and Computer Sciences (EECS) department at U.C. Berkeley. His research interests center on automated formal methods with applications to embedded systems, electronic design automation, and computer security. He is co-author of a textbook on embedded systems and numerous papers. He has a B.Tech. from IIT Bombay (1998), and an M.S. (2000) and Ph.D. (2005) from Carnegie Mellon University.

Jia Zou received a Ph.D. in 2011 from the EECS department of the University of California, Berkeley. He received his bachelor's degree from the University of Minnesota, Twin Cities in 2006, after which he joined the Center for Hybrid and Embedded Software System (CHESS) under the supervision of Professor Edward A. Lee at Berkeley. His research interest mainly focuses on the design and implementation of distributed real-time embedded systems.