A Time-Centric Model for Cyber-Physical Applications

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

University of California at Berkeley, Berkeley, CA, 94720, USA
{eidson,eal,matic,sseshia,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 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. We show the use of this environment in the design of interesting practical real-time systems.

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 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. 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 adaptations of traditional computer science concepts to distributed control problems make the assumption of the oracle t. For example, in [21] consensus problems from computer science are translated into control systems formulations. 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 [12] can be used to approximate a uniform model of time, but the analysis of the dynamics has to include the imperfections.

Although 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. This

enables safety or quality of service verification, and thus helps with integration and scalability of designed systems. 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. However, the multiplication of modeling languages raises the question of mutual consistency and interoperability. This is mainly why the OMG consortium extended UML with a profile called MARTE (Modeling and Analysis of Real-Time and Embedded Systems) [23]. Another trend is the acceptance of synchronous-reactive languages, particularly SCADE [1], in safety critical applications. 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. Also related to our work are component frameworks based on formal verification methods, like the BIP framework [2], but they mostly focus on compositional verification of properties such as deadlock freedom. BIP relies on priorities to model scheduling policies and, as far as we know, has not been used to address modeling and design problems for components with explicit timing requirements.

To ensure proper real-time interaction between the dynamics of the controller and the dynamics of the controlled physical system, programmers of embedded systems typically use platform-specific system timers. However, design of the system should be independent of implementation details, in order to allow for portability of the design. In [26] we presented a programming model called PTIDES (*programming temporally-integrated distributed embedded systems*) that addresses this problem by relying on a suitable abstraction of time. With PTIDES, application programmers specify the interaction between the control program and the physical dynamics in the system model, without the knowledge of details such as timers. Paper [28] studies the semantic properties of an execution model that permits out of order processing of events without sacrificing determinacy and without requiring backtracking.

The goal of this work is to demonstrate the usefulness of PTIDES for time-critical CPS applications. We first explain how design with PTIDES results in deterministic processing of events. Then we illustrate how to specify timed reactions to events in PTIDES models. This results in traces from model simulation and execution of automatically generated code being identical. In order to account for different modes of operation, modal models have been widely used in embedded system design [8]. Here, we show the use of modal models within the context of a timed environment, i.e., we illustrate timed mode transitions and operations in modes at certain time instants.

This paper is organized as follows. First, section 2 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. At the top level, this environment uses the PTIDES [26] extension to the Ptolemy II simulation framework [7] as a coordination language for the design of distributed real-time embedded systems. Section 3 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 example in section 4, which shows the use of this environment and particularly the ability to explicitly address timing in the design of interesting practical real-time systems. We conclude in section 5.

2 Design Environment

2.1 PTIDES Workflow

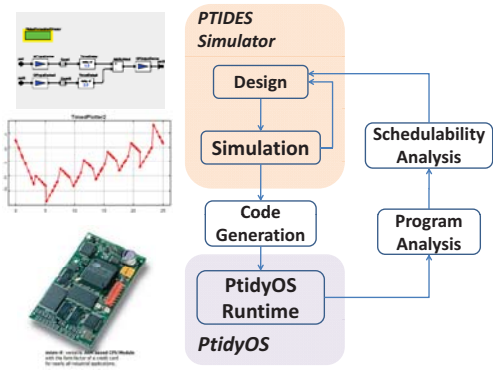

Fig. 1. PTIDES Code Generation Workflow

Fig. 1 shows our envisioned workflow, from modeling to code generation to implementation. The proposed PTIDES design environment is an extension of the Ptolemy II framework which supports modeling, simulation, and design of systems using mixed models of computation. 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, each actor can be annotated with execution time, and with several implemented scheduling schemes simulation can be performed to confirm whether real-time constraints can be met for a given set of inputs.

The PTIDES design environment leverages the Ptolemy II code generation framework, and allows a programmer to generate target-specific implementations from the PTIDES model once she is satisfied with the design. The generated executable includes a lightweight real-time operating system (RTOS) which we call PtidyOS. Its real-time scheduler implements PTIDES semantics and therefore preserves the timing specifications present in the top level PTIDES design. Like TinyOS [17], PtidyOS is a set of C libraries that glues together application code, which then runs on bare-iron. Currently, our code generation framework supports a Luminary Micro board as our target platform. Once implemented in PtidyOS, platform specific worst-case-execution times need to be extracted through program analysis, and schedulability analysis is needed to ensure the real-time requirements are met. It is important to point out, at this point of our PTIDES project, program and schedulability analysis are still under development. 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 of several applications to illustrate how explicit timing constraints can be used, but not on their PtidyOS implementations.

2.2 Model Time and Physical Time

PTIDES is based on discrete-event (DE) systems [3] [25], 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 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 [16]. The purpose of this paper is not to study its rigorous and determinate semantics. For that an interested reader is referred to [18] 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 timetamp τ at physical time $t \geq \tau$, 2) actuators receives events with timestamp τ at physical time $t \leq \tau$, and 3) network interfaces receive events with timestamp τ at physical time $t \leq \tau$. We explain these constraints in detail below.

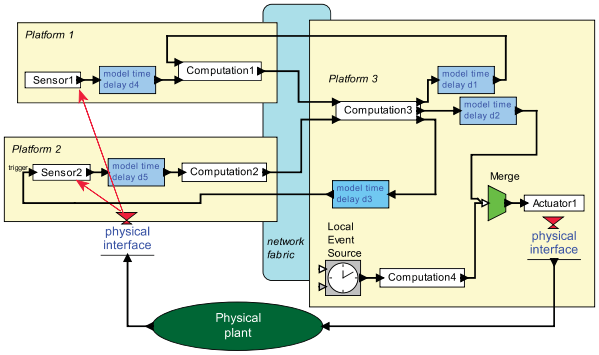

Fig. 2. Prototypical CPS

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 avoid causality violations. 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 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. Our assumption that it equals the time stamp makes the analysis in next subsections particularly simple, so for the purposes of this paper we proceed with that.

2.3 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 [19], and CAN buses [24]). 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 compromising 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 $-d_2 + n + s$). 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.

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.

3 Temporal Semantics in PTIDES

PTIDES semantics is fully described in [26] and [28], 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 [25].

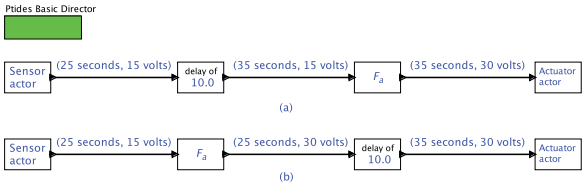

Fig. 3. Linear combination of actors

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 $F_\delta(s) : S \to S$ 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.

Modal Models.

The use of modal models is well established both in the literature, for example Statecharts [8], UML [22], and in commercial products such as Simulink/Stateflow from MathWorks [20]. Note that we use the term *modal* to describe models that extend finite-state machines by allowing states to have Ptolemy II models as refinements [14]. 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 4. Our style for modal models follows the pattern shown in Figure 4. 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.

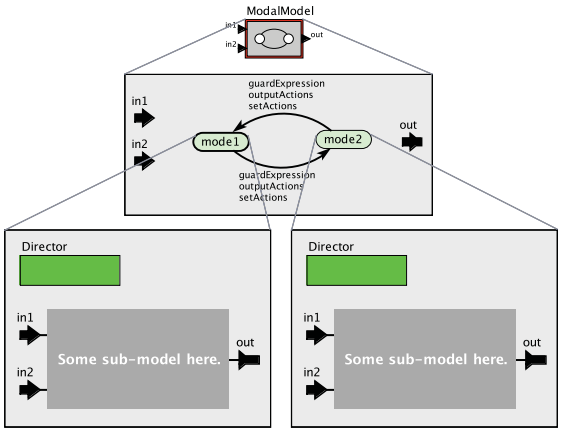

Fig. 4. General pattern of a modal model with two modes, each with its own refinement

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. While straightforward, these rules can yield surprises particularly when one or more of the refinements modify the model time of a signal.

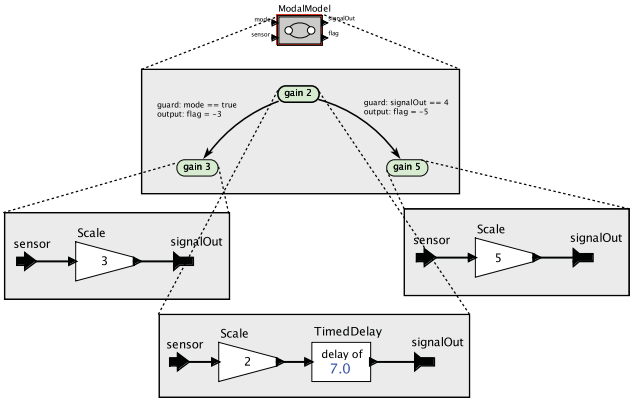

Fig. 5. Simple time-sensitive modal model

For example consider the simple modal model of Figure 5. 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.

4 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 [16]. 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 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 [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 detection of missing signals to drive mode changes in the operation of power plants.
- 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 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 at platform network outputs of sending deadlines to ensure that multi-platform feasible solutions are computable.

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 these 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.

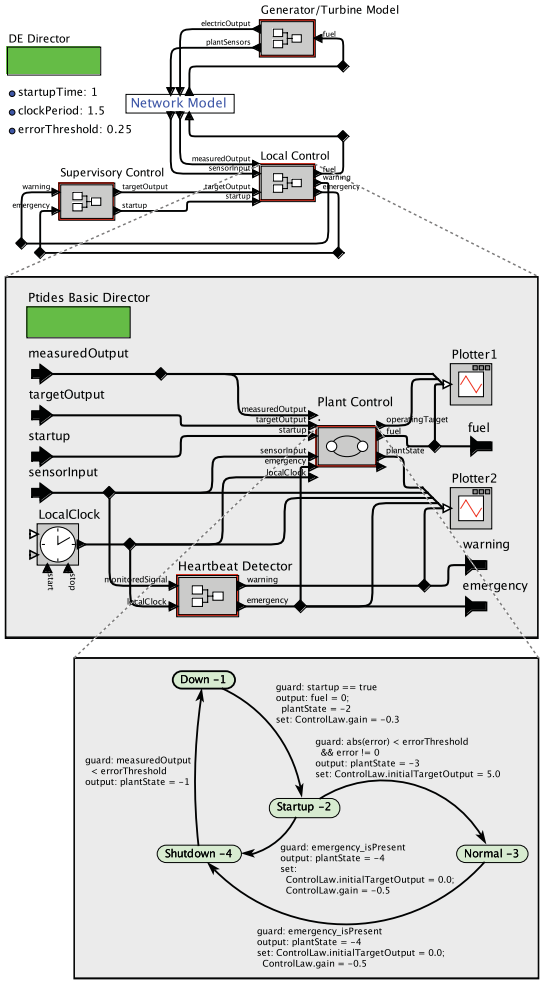

Fig. 6. Model of a small power plant

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. 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 3 . 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 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.

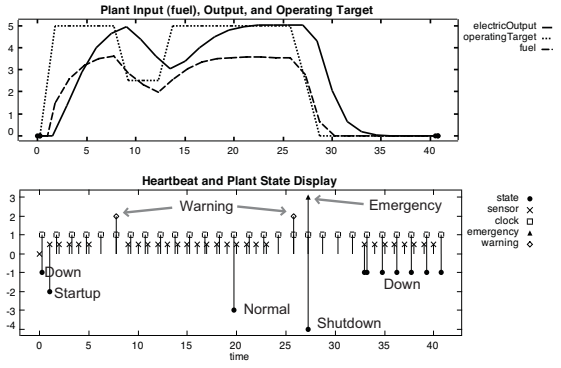

Fig. 7. Power plant output and events

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 6 and 7, the Generator/Turbine Model sends (time-stamped) sen-

sor readings over the network to the Local 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 and illustrates another use of the modal model temporal semantics of section 3. 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.

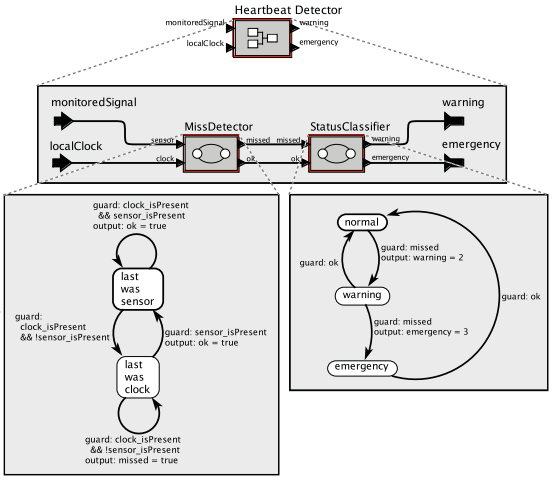

Fig. 8. Heartbeat detector that raises alarms

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.

If the generated code correctly performs a comparison between timestamp and physical time, as explained in section 2.3, 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 [5]. In an accompanying technical report [6] 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 in [6], this relationship is easily captured in the timed semantics of PTIDES.

5 Conclusion

This paper reviewed Ptolemy II enhancements for several important aspects of CPS design and deployment, namely PTIDES for distributed real-time systems, and 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, 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. By combining PTIDES with modal models, we illustrated timed mode transitions which enable time-based detection of missing signals to drive mode changes in the operation of 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 [27], 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 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.

A major component of our work will be refinement to 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 PtidyOS that can be considered as a lightweight operating system with PTIDES semantics.

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. J. C. Eidson. *Measurement, Control, and Communication Using IEEE 1588*, pages 194–200. Springer, London, 2006.
6. J. C. Eidson, E. A. Lee, S. Matic, S. A. Seshia, and J. Zou. Time-centric models for designing embedded cyber-physical systems. Technical Report UCB/EECS-2009-135, EECS Department, University of California, Berkeley, Oct 2009.

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. Finite state machines and modal models in ptolemy ii. Technical Report UCB/EECS-2009-151, EECS Department, University of California, Berkeley, Nov 2009.
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*, 17(12):1217–1229, 1998.
16. E. A. Lee and H. Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *EMSOFT*, Salzburg, Austria, 2007. ACM.
17. P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler. The emergence of networking abstractions and techniques in tinyos. In *First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004)*, 2004.
18. X. Liu and E. A. Lee. CPO semantics of timed interactive actor networks. *Theoretical Computer Science*, 409(1):110–125, 2008.
19. 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.
20. Mathworks. *Matlab*. http://www.mathworks.com/products/matlab/, 1996.
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. O.M.G. U.m.l. specification Version 1.3. *Object Management Group*, 1999.
23. OMG. Uml profile for modeling and analysis of real-time and embedded systems (marte). http://www.omgmarte.org/, 2008.
24. K. Tindell, H. Hansson, and A. Wellings. Analysing real-time communications: Controller area network (CAN). In *Proceedings 15th IEEE Real-Time Systems Symposium*, pages 259–265. Citeseer, 1994.
25. B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation*. Academic Press, 2nd edition, 2000.
26. 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.
27. Y. Zhou and E. A. Lee. Causality interfaces for actor networks. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–35, 2008.
28. 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.