Modal Models in Ptolemy

Edward A. Lee and Stavros Tripakis
University of California, Berkeley
eal@eecs.berkeley.edu, stavros@eecs.berkeley.edu

Abstract

Ptolemy is an open-source and extensible modeling and simulation framework. It offers heterogeneous modeling capabilities by allowing different models of computation to be composed hierarchically in an arbitrary fashion. This paper describes modal models, which allow to hierarchically compose finite-state machines with other models of computation, both untimed and timed. The semantics of modal models in Ptolemy are defined in a modular manner.

Keywords Hierarchy, State machines, Modes, Heterogeneity, Modularity, Modeling, Semantics, Simulation, Cyber-physical systems.

1. Introduction

Cyber-physical systems (CPS) consist of digital computers interacting among themselves and with physical processes. CPS applications are emerging at high rates today in many domains, including energy, environment, healthcare, transportation, etc.

Designing CPS is a non-trivial task, as these systems manifest non-trivial dynamics, complex interactions, dynamic behavior, and a large number of components. The design complexity is increased by the inherent heterogeneity in modeling such systems: parts of the system are digital, others are analog; parts are timed, others untimed; parts are discrete-time, others are continuous-time; parts are synchronous, others are asynchronous; and so on. This inherent heterogeneity implies a need for heterogeneous modeling. By the latter we mean a method and associated tools, that provide designers with a way of combining different models of computation, in an unambiguous way, in a single model of a system. A model of computation (MoC) here refers to a language or class of languages with a common syntax and semantics. Different MoCs realize different modeling paradigms, each being more or less suitable for capturing different parts of the system.

A number of modeling languages exist today, realizing different MoCs. Many of these languages are gaining acceptance in the industry, in so-called model-based design methodologies. Examples are UML/SysML, Matlab/Simulink/Stateflow, AADL, Modelica, LabVIEW, and others. These types of languages are raising the level of abstraction in designing CPS, by offering mechanisms to capture concurrency, interaction, and time behavior, all of which are essential concepts in CPS. Moreover, verification and code generation tools exist for many of these languages, allowing to go beyond simple modeling and simulation, and facilitating the process of going from high-level models to low-level implementations.

Despite these advances, however, no universally accepted solution exists for heterogeneous modeling. In fact, integration of modeling languages and tools is still a common theme in many research or industrial projects, as well as products (e.g., co-simulation environments) despite the fact that such solutions are often cumbersome to use and unsatisfactory, at best.

One of the longest efforts attacking the heterogeneous modeling problem is the Ptolemy project [15, 25]. Ptolemy follows the actor-oriented paradigm, where a system consists of a set of actors, which can be seen as processes executing concurrently and communicating using some mechanism. In Ptolemy, the exact manner in which actors execute (e.g., by interleaving, in lock-step, or in some other order) is very different, which makes them suitable for different classes of applications.

Copyright is held by the author/owner(s). The proceedings are published by Linköping University Electronic Press. Proceedings available at: http://www.ep.liu.se/ecp/047/

We should emphasize the importance of syntax, in addition to semantics, in choosing a MoC. State machines, for example, can be given a discrete-time semantics, and so can a synchronous language such as Lustre [19]. Even though the two have the same semantics, their syntax (in the broad sense) is very different, which makes them suitable for different classes of applications.
nder) and the exact manner in which they communicate (e.g., through message passing or shared variables) are not fixed once and for all: they are defined by an MoC. The implementation of an MoC in Ptolemy is called a domain and is realized by a director, which semantically is a composition operator. Currently, Ptolemy includes a number of different domains and corresponding directors, including, finite-state machines (FSM), synchronous data flow (SDF) [26], synchronous reactive (SR) [11, 19, 14], and discrete event (DE) [7, 33, 9, 37, 23, 20, 8].

Among the important characteristics of Ptolemy is that (a) it is open-source (and free); and (b) it is architected to be easily extensible. Thanks to these features, new domains and new actors are being added to the tool by different groups, depending on their specific interests.

Another essential feature of Ptolemy is that domains can be combined hierarchically, in an arbitrary fashion. For example, the model in Figure 1 combines SDF with hierarchical FSMs; the model in Figure 3 combines DE with FSMs. This is the fundamental mechanism that Ptolemy provides to deal with the heterogeneous modeling problem. It allows designers to build models where different parts are described in different MoCs, in a well-structured manner [15].

In this paper, we are particularly interested in one aspect of heterogeneous modeling in Ptolemy, namely, modal models. Modal models are hierarchical models where the top level model consists of an FSM, the states of which are refined into other models (possibly from different domains). Modal models are suitable for a number of applications. They are especially useful in describing event-driven and modal behavior, where the system’s operation changes dynamically by switching among a finite set of modes. Such changes may be triggered by user inputs, sensor data, hardware failures, or other types of events, and are essential in fault management, adaptivity, and reconfigurability (see, for instance, [36, 35]). A modal model is an explicit representation of this type of behaviors and the rules that govern transitions between behaviors.

The main contribution of this paper is to provide a formal semantics of Ptolemy modal models. In the process, we also give a modular and formal framework for Ptolemy in general, which is an additional contribution. We do not formalize all the domains of Ptolemy, however, as this is beyond the scope of this work.

The paper is organized as follows. Section 2 briefly reviews the visual syntax of Ptolemy through an example. In Section 3 we provide a formalization of the abstract semantics of Ptolemy. In Section 4 we provide the formal semantics of Ptolemy modal models. In Section 5 we discuss possible alternatives and justify our choices. Section 6 discusses related work. Section 7 concludes the paper.

2. Visual Syntax

Ptolemy models are hierarchical. They can be built using a visual syntax, an example of which is given in Figure 1. This example contains, at the top level of the hierarchy, a model with five actors, Temperature Model, Bernoulli, ModalModel, SequencePlotter and SequencePlotter2. The SDF domain is used at this level, as indicated by the use of SDF Director, explained below. Temperature Model and ModalModel are composite actors: they are refined into other models, at a lower level of the hierarchy.

The refinement of ModalModel is an FSM with two locations, normal and faulty. This FSM is hierarchical: each of its locations is refined into a new FSM, as shown in the figure. In Ptolemy, FSMs use implicitly the FSM domain. This is why no director is shown in the FSM models. The FSM director is implied. Note that, although in this example the location refinements are FSMs, this need not be the case: they can be models using any of the Ptolemy domains (e.g., see example of Figure 3).

The refinement of Temperature Model is shown in Figure 2. This refinement does not specify a domain (it contains no director). In such a case, the refinement uses implicitly the same domain as its parent, that is, in this case, the SDF domain. Since this model mixes SDF and FSM, it is an example of a heterogeneous model.

The visual syntax of Ptolemy contains other elements, which we briefly describe next. For details, the reader is referred to [27, 24] and the Ptolemy documentation [1]. Each actor contains a set of ports, used for communication with other actors. Ports are explicitly shown in the internal model of a composite actor: for instance, fault is an input port and heat is an output port of ModalModel. A port may be an input, an output, both, or neither. Parameters can also be defined: for instance, heatingRate is a parameter of the top-level model of Figure 1, set initially to 0.1 (the value of parameters can be modified dynamically during execution).

FSMs in Ptolemy consist of a finite set of locations, one of which is the initial location, and some of which may be labeled as final locations. Initial locations are indicated by a bold outline; the initial locations of the FSMs in Figure 1 are normal and heating. A transition links a source location to a destination location. A transition is annotated with a guard, a number of output actions and a number of set actions. Guards are expressions written in the Ptolemy expression language. Actions are written in the Ptolemy action language. Guards of two or more outgoing transitions of the same location need not be disjoint, in which case the FSM is non-deterministic. The user can indicate this, in which case transitions are visually rendered in red. Default transitions, indicated with dashed lines, are to be taken when no other transitions are enabled, i.e., their guard is the negation of the disjunction of the guards of all other transitions outgoing from the same source location. Reset transitions, indicated with open arrowheads, result in the refinement of the destination state being reset to its initial condition. Preemptive transitions, indicated by a red circle at the start of the transition, may prevent the execution of the current state refinement, when the guard evaluates to true.

For the visual syntax, we use the term location instead of state, in order to distinguish it from the semantical concept of state (Section 3).
Figure 1. A hierarchical Ptolemy model.

Figure 2. The Temperature Model composite actor of Figure 1.

3. Ptolemy Abstract Semantics

The semantics and execution of a Ptolemy model is defined by means of so-called abstract semantics. The same mechanism is used to ensure compositionality of Ptolemy domains. Mathematically, a Ptolemy model can be viewed as an abstract state machine, with a set of states, inputs and outputs. The abstract semantics defines the transitions of this machine, that is, how its state and outputs evolve according to the inputs.

Evolution can be seen as being untimed, that is, a sequence of transitions, or timed, that is, a sequence of transitions annotated by some timing information (e.g., a time delay since the previous transition). It is interesting to note that time is mostly external to the definition of the abstract state machine, that is, the times in which transitions are taken are primarily decided by the environment of the machine. However, timed, or proactive, machines can also be defined, by providing means to impose constraints on these times. We do this using timers (see Section 3.1.2). In the absence of any such constraints, the machine is untimed, or reactive.

From an implementation point of view, the abstract semantics is essentially a set of methods (in the object-oriented programming sense) that every actor in a model implements. Composite actors also implement these methods, through their director. By implementing these methods in different ways, the various types of Ptolemy directors realize different models of computation. In the Java implementation of Ptolemy the abstract semantic methods form a Java interface that actor and director classes implement. This interface includes the following methods:

- initialize: it defines the initial state of the machine.
- fire: it computes the outputs at a given point in time, based on the inputs and state at that point in time.
- postfire: it updates the state.

A formalization of the abstract semantics of Ptolemy is provided next.

3 For the purposes of this discussion, we omit some methods, e.g., prefire, etc. We also ignore implementation of communication between actors. See [1] for details.
3.1 A formalization of abstract semantics of actors

3.1.1 Untimed actors

An untimed actor is formalized as a tuple

\[(S, s_0, I, O, F, P)\]

The actor has a set of states \(S\), a set of input values \(I\), and a set of output values \(O\). The \text{initialize} method of the actor is formalized as an initial state \(s_0 \in S\). The \text{fire} and \text{postfire} methods of the actor are formalized as two functions \(F\) and \(P\), respectively, of the following type:

\[F : S \times I \times \mathbb{N} \rightarrow O\]
\[P : S \times I \times \mathbb{N} \rightarrow S\]

That is, \(F\) returns an output value \(y \in O\), given a state \(s \in S\), an input value \(x \in I\), and an index \(j \in \mathbb{N}\); \(P\) returns a new state, given a state, an input value and an index. The index is used to model non-determinism, and can be seen as a “dummy” input.\(^4\)

We require that \(F\) and \(P\) be total in \(S\) and \(I\). \(F\) and \(P\) may be partial in \(\mathbb{N}\) (i.e., in their index argument), however, we require that for any \(s \in S\) and \(x \in I\), there exists at least one \(j \in \mathbb{N}\) such that both \(F(s, x, j)\) and \(P(s, x, j)\) are defined. If there is a unique such \(j\) for all \(s \in S\) and \(x \in I\) then the actor is deterministic. In that case we omit index \(j\) and write simply \(F(s, x)\) and \(P(s, x)\).

The semantics of an untimed actor can be then defined as a set of sequences of transitions, of the form:

\[s_0 \xrightarrow{s_1} s_1 \xrightarrow{s_2} \cdots \]

such that for all \(i = 0, 1, \ldots\), we have \(s_i \in S\), \(x_i \in I\), \(y_i \in O\), and there exists \(j_i \in \mathbb{N}\) such that

\[y_i = F(s_i, x_i, j_i) \quad (1)\]
\[s_{i+1} = P(s_i, x_i, j_i) \quad (2)\]

Note that, exactly what the sets \(S, I\) and \(O\) are, and exactly how the functions \(F\) and \(P\) are defined, is a property of a given actor: it is in this sense that this semantics is abstract. Different actors will have different instantiations of this abstract semantics. Also note that the above elements essentially define a non-deterministic Mealy machine, where \(F\) is the output function of the machine, and \(P\) the state update function. This machine is not necessarily finite-state. The input and output domains may also be infinite.

Examples of untimed actors

A simple untimed actor is the \text{Gain} actor which produces at its output a value \(k \cdot x\) for every value \(x\) appearing at its input. \(k\) is a parameter of the actor. This actor is deterministic. It has a single state, and therefore a trivial (constant) update function \(P\). Its \(F\) function is defined simply by: \(F(x) = k \cdot x\).

\(^4\) Since \(j \in \mathbb{N}\), we can model unbounded, but enumerable, non-determinism. The reader may wonder why we do not model the pair \(F, P\) simply as a single function with type \(S \times I \rightarrow 2^{O \times \mathbb{N}}\), that is, taking a state and an input and returning a set of state, output pairs. The reason is that we want to decouple output and update functions, which allows to give semantics of modal models in a modular manner: see Section 4.2. Once the \(F\) and \(P\) functions have been decoupled, it is necessary for some bookkeeping in order to keep track of non-deterministic choices, that must be consistent among the two functions. This role is played by the index \(j \in \mathbb{N}\).

3.1.2 Timed actors

The semantics of timed actors extend those of untimed actors with time. In particular, timed actors have special state variables, called \textit{timers}, that measure time. Our timers are dense-time variables (they take values in the set of non-negative reals, \(\mathbb{R}_{\geq 0}\)) inspired by the model of [13]. A difference with [13] is that in our case timers can be created or destroyed dynamically, and the set of timers that are active at any given time is not necessarily bounded. Also, our timers can be suspended and resumed, which is not the case with the timers of [13]. Timers are set to some initial value when they are created, and then run downwords (i.e., decrease as time elapses) until they reach zero, at which point they expire. A timer can be suspended which means it is “frozen” and ceases to decrease with time. It can then be resumed. Suspended timers are also called inactive, otherwise they are active.

In the case of timed actors, the sets \(I\) and \(O\) often contain the special value \(e\) denoting \textit{absence} of a signal at the corresponding point in time. We will use this value in the examples that follow.

Consider a timed actor with \text{fire} and \text{postfire} functions \(F\) and \(P\). The semantics of this actor can be defined as a set of sequences of timed transitions of the form:

\[s_0 \xrightarrow{s_1} s_1 \xrightarrow{s_2} \cdots \]

such that there exists a sequence of indices \(j_0, j_1, \ldots \in \mathbb{N}\), and for all \(i = 0, 1, \ldots\), we have \(s_i \in S\), \(x_i \in I\), \(y_i \in O\), \(d_i \in T\), and

\[y_i = F(s_i, x_i, j_i) \quad (3)\]
\[s_{i+1} = P(s_i \oplus d_i, x_i, j_i) \quad (4)\]
\[d_i \leq \min\{c | c \text{ an active timer in } s_i\} \quad (5)\]

\(d_i \in \mathbb{R}_{\geq 0}\) denotes the time elapsed at state \(s_i\). \(s_i \oplus d_i\) denotes the operation which consists in decrementing all active timers in \(s_i\) by \(d_i\). Condition (5) ensures that this operation will not result in some timers becoming negative, i.e., that no timer expiration is “missed”. This condition therefore “forces” the environment of the actor to fire the actor at least at those instants when its timers are set to expire. Note that the actor could also be fired at other instants as well, for example, whenever an external event is received. The actor itself does not, and cannot, specify those other instants, because they are generally context-dependent.

Notice that, even though the above semantics does not explicitly mention suspensions and resumptions of timers, these actions can be easily modeled as part of the inputs \(x_i\). In Proleny, these inputs are not accessible to the user, however, only to the director. This is particularly the case for hierarchical modal models, as described in Section 4.2.

\textbf{Superdense time:} It is worth noting that the delays \(d_i\) can be zero. This implies in particular that multiple output events can occur at the same real-time instant. It is convenient to model such cases using so-called \textit{superdense} time, i.e., the set \(\mathbb{R}_{\geq 0} \times \mathbb{N}\) [32, 28, 31]. Then, an output \(y\) can be seen as a signal with a superdense time axis, that is, as
a partial function from \( \mathbb{R}_{\geq 0} \times \mathbb{N} \) to a set of values. For instance, in a run of the form

\[
\begin{array}{c}
s_0 \xrightarrow{x_0 y_0 d_0} s_1 \xrightarrow{x_1 y_1 v_1} s_2 \xrightarrow{x_2 y_2 d_2} \cdots
\end{array}
\]

where \( d_0 = d_1 = 0 \) and \( d_2 > 0 \), the output signal \( y \) can be seen as a function on superdense time, such that \( y(0, 0) = y_0, y(0, 1) = y_1, y(d_2, 0) = y_2 \), and so on.

**Examples of timed actors**

First, let us consider a DiscreteClock actor that periodically emits some value \( v \) at its output, with period \( \pi \in \mathbb{R}_{\geq 0} \). Both \( v \) and \( \pi \) are parameters of the actor. The DiscreteClock actor has no inputs. It has a single state \( \pi \), and output with value \( v \) whenever an input is received: at that point, \( \pi \) is set to \( \pi \), and \( c \) is produced. Formally, a state \( F(c) = v \) if \( c = 0 \), and \( \pi \) if \( c > 0 \).

The definition of \( F \) states that an output with value \( v \) is produced when the timer \( c \) reaches zero, otherwise, the output is absent. The definition of \( P \) states that the timer is reset to \( \pi \) when it reaches zero and is left unchanged otherwise.

Next, let us consider the ConstantDelay actor, which, for every input with value \( x \) that it receives at time \( t \), it produces an output with value \( x \) at time \( t + \Delta \), where \( \Delta \in \mathbb{R}_{\geq 0} \), is a parameter of the actor. As state variables, ConstantDelay maintains a set of active timers \( C \) plus, for each \( c \in C \), a variable \( v_c \) to memorize the value that must be produced when \( c \) expires. Initially \( C \) is empty. A new timer \( c \) is added to \( C \) whenever an input is received: at that point, \( c \) is set to \( \Delta \) and \( v_c \) is set to \( x \), the value of the input. When a timer \( c \) expires it is removed from \( C \) and output with value \( v_c \) is produced. Formally, a state \( s \) of ConstantDelay is a set of triples of the form \( (c, \delta, v_c) \), where \( c \) is a timer. \( \delta \in \mathbb{R}_{\geq 0} \) is the current value of \( c \), and \( v_c \) is as explained above. The initial state is \( s_0 = \emptyset \). The \( F \) and \( P \) functions of ConstantDelay can be defined as follows (again we omit \( j \) because of determinism):

\[
\begin{align*}
F(s, x) &= \begin{cases} 
v_c & \text{if } \exists (c, 0, v_c) \in s \\
\varepsilon & \text{otherwise} \end{cases} \\
P(s, x) &= \begin{cases} 
(s \setminus \{ (c, 0, v_c) \}) \cup Q & \text{if } \exists (c, 0, v_c) \in s \\
s \cup Q & \text{otherwise} \end{cases} \\
Q &= \begin{cases} 
\{(c', \Delta, x)\} & \text{if } x \neq \varepsilon \text{ and } \exists (c', \delta, v_{c'}) \in s \\
\emptyset & \text{otherwise} \end{cases}
\end{align*}
\]

Note that a “fresh” timer \( c' \) is created only when the input \( x \) is not absent, as defined by \( Q \).

**3.1.3 Untimed actors as a special case of timed actors**

As expected, an untimed actor can be seen as a special case of a timed actor, with no timers. Because of this, untimed actors can also be given semantics in terms of sequences of timed transitions. In this case, Condition (4) reduces to Condition (2), and Condition (5) is trivially satisfied with the convention that the minimum of an empty set is infinity. This means that the time instants when untimed actors are fired are entirely determined by the context in which these actors are embedded.

### 3.2 Composite actors

As illustrated in Section 2, Ptolemy allows to build hierarchical models, by encapsulating a set of actors, plus a director, within a composite actor. The latter is itself an actor, thus can be further encapsulated to create new composite actors. Models of arbitrary hierarchy depths can be built this way.

A composite actor \( C \) has an abstract semantics just like any actor. How this abstract semantics is instantiated depends on: (a) the instantiation of the abstract semantics of the internal actors of \( C \); and (b) the director that \( C \) uses.

Directors can be viewed formally as composition operators: they define functions \( F \) and \( P \) of a composite actor \( C \), given defined such functions for all internal actors of \( C \).

A large number of directors are included in Ptolemy, implementing various models of computation. It is beyond the scope of this document to formalize all these directors. We informally describe two of them, namely, SR (synchronous reactive) and DE (discrete event). More information can be found in [23, 14, 29, 30, 8]. In the next section, we formalize the semantics of the FSM Director. The latter implements modal models, which is the main topic of this paper.

**Synchronous Reactive (SR):** Every time a composite actor \( C \) with an SR director is fired, the SR director repeatedly fires all actors within \( C \) until a fixpoint is reached. This fixpoint assigns values to all ports of actors of \( C \). Note that, because of interconnections between actors, some output ports are typically connected to input ports of other actors of \( C \), and therefore obtain equal values in the fixpoint. The fixpoint is defined with respect to a flat CPO, namely, the one that has a bottom element \( \bot \) representing an “undefined” or “unknown” value, and all other, “true” values, greater than \( \bot \) in the CPO order (see [14]). The fixpoint is computed by assigning initially \( \bot \) to all outputs, and then iterating in a given order the \( F \) functions of all actors of \( C \). Any execution order can be used and is guaranteed to reach the fixpoint, although some execution orders may be more efficient (i.e., may converge faster). When the fixpoint is reached, the fire() method of the SR director (and consequently, of \( C \)) returns.\(^5\) The postfire() method \( P \) of \( C \) is implemented by invoking the \( P \) methods of all internal actors of \( C \).

**Discrete Event (DE):** DE leverages the SR semantics, but extends it with time. (see [23, 8]). As is typical with DE simulators, the DE director maintains an event queue that stores events in timestamp order. Initially, the event queue is empty. When actors are initialized, some of them may post initial events to the event queue. Whenever the composite actor is fired, the earliest events are extracted from

\(^5\) The fixpoint may contain \( \bot \) values, which means the model contains feedback loops with causality cycles. In this case, the Ptolemy implementation returns a Java exception.
the event queue and presented to the actors that receive them. In contrast to standard DE simulators, Ptolemy incorporates the SR semantics for processing simultaneous events. In particular, a fixpoint is computed, starting with the extracted events at the specified ports, and $\bot$ values for all other, unknown, ports. Fire() returns when the fixpoint is found, as in the SR case. Postfire() consists in calling postfire() of internal actors, as in the SR case. During postfire(), actors may post new events to the queue.

4. Modal Model Semantics

A modal model $M$ is a special kind of composite actor. In the visual syntax, $M$ is defined by a finite-state machine $M_c$ whose locations can be refined into sub-models, as illustrated in Figure 1. In Ptolemy terminology, $M_c$ is called the controller of $M$. Each of these sub-models is itself a composite actor. Therefore, the internal actors of $M$ are the composite actors that refine the locations of $M_c$, plus $M_c$ itself. Note that a special case of modal model is an FSM actor: this is a modal model whose controller has no refinements. Another special case of modal model is a hierarchical state machine: this is a modal model whose refinements are FSM actors or are themselves hierarchical state machines.

In this section, we describe the semantics of modal models, starting with the simple case of FSM actors, and extending to the general case of modal models.

4.1 Semantics of FSM actors

FSM actors are untimed actors. For a given FSM actor $M$, its set of states is the set of all possible valuations of the state variables of $M$. The set of state variables includes all parameters of $M$ (which in Ptolemy can be changed dynamically) as well as a state variable to record the current location of $M$. A valuation is a function assigning a value to every state variable. The initial state assigns to each state variable. The initial state assigns to each state variable, namely $s_0$, all parameters to be the set of initial states, i.e.: $S = S^c \times S^d \times \cdots \times S^n$

$s_0 = (l_1, s_0^1, \ldots, s_0^n)$

Although timers are just a special kind of state variables, it is convenient to be able to refer to them specifically. Therefore, we define $C$ as the set of timers of $M$, as:

$$C = \bigcup_{i=1}^{n} C^i$$

In $s_0$, all timers except those in $C^1$ are set to their suspended state. Those in $C^1$ are set to their active state.

It remains to define functions $F$ and $P$ of $M$. Consider a state $s \in S$ and an input $x \in I$. Let $s = (s_c, s_1, \ldots, s_n)$ be the vector of component states of $M_c, M_1, \ldots, M_n$, respectively. Suppose the location of $M_c$ at $s_c$ is $l_i$. Let $J \subseteq \mathbb{N}$ be the set of indices $j$ for which $F^j(s_c, x, j)$ and $P^j(s_c, x, j)$ are defined. We distinguish cases:

1. There are no outgoing transitions of $M_c$ from location $l_i$ that are enabled at $s_c$. Then, for $j \in J$, we define $F(s, x, j) = F^j(s_c, x, j)$, $P(s, x, j) = (s_c, s'_1, \ldots, s'_n)$, where:
(a) \( s'_j = P^i(s_i, x, j) \);
(b) for all \( m = 1, \ldots, n \) with \( m \neq i \), we have \( s'_m = s_m \).

2. There exist \( k \geq 1 \) preemptive outgoing transitions from \( l_i \) that are enabled at \( s \) and \( x \). Suppose, without loss of generality, that the \( j \)-th such transition goes from location \( l_i \) to location \( l_j \), for \( j = 1, \ldots, k \), and denote its output action and set action by \( \alpha_j \) and \( \beta_j \), respectively. Then, for \( j = 1, \ldots, k \), we define \( F(s, x, j) = y_j \) and \( P(s, x, j) = (s'_j, x'_j, \ldots, s''_j) \), where:
(a) \( y_j \) is obtained from \( \alpha_j \), as in the FSM actor semantics;
(b) \( x'_j \) is obtained from \( l_j \) and \( \beta_j \) as in the FSM actor semantics;
(c) if the \( j \)-th transition is not a reset transition then \( s'_j \) is identical to \( s_j \), except that all suspended timers in \( C^j \) are resumed; if the \( j \)-th transition is a reset transition then \( s'_j \) is the initial state of \( M_j \); \( s'_j = s''_j \); (note that the timers of \( M_j \), if any, are also re-initialized in the case of a reset transition);
(d) \( s'_j \) is identical to \( s_i \), except that all timers in \( C^i \) are suspended;
(e) for all \( m = 1, \ldots, n \) with \( m \neq j \) and \( m \neq i \), we have \( s'_m = s_m \).

3. There are no preemptive outgoing transitions from \( l_i \) that are enabled at \( s \) and \( x \), but there exist \( k \geq 1 \) non-preemptive outgoing transitions from \( l_i \) that are enabled at \( s \) and \( x \). Let \( j_1 = 1, \ldots, k \) and suppose that the \( j_1 \)-th such transition goes from \( l_i \) to \( l_j \) and has output and set actions \( \alpha_{j_1} \) and \( \beta_{j_1} \). Let \( j_2 \) range in \( J \). Then, for \( j = j_1 \cdot j_2 \), we define \( F(s, x, j) = y_j \) and \( P(s, x, j) = (s'_j, x'_j, \ldots, s''_j) \), where:
(a) \( y_j \) is obtained by applying the output action \( \alpha_{j_1} \) to \( F^i(s_i, x, j_2) \), that is, to the output produced by \( M_i \) for non-determinism index \( j_2 \);
(b) \( x'_j \) is obtained as in Case 2b.
(c) \( s'_j \) is obtained as in Case 2c.
(d) \( s'_j \) is obtained by applying the set action \( \beta_{j_1} \) to \( P^i(s_i, x, j_2) \) and suspending all timers in \( C^i \);
(e) for all \( m = 1, \ldots, n \) with \( m \neq j \) and \( m \neq i \), we have \( s'_m = s_m \).

Item 1 treats the case where no transition of the controller is enabled: in this case, the modal model \( M \) behaves (i.e., fires and postfires) like its current refinement \( M_i \). Item 2 treats the case where preemptive transitions of the controller are enabled, possibly in addition to non-preemptive transitions. In this case the preemptive transitions preempt the firing and postfiring of \( M_i \), and only the outputs produced by the transition of the controller can be emitted. Item 3 treats the case where only non-preemptive transitions of the controller are enabled. In this case, before choosing and taking such a transition non-deterministically, we must fire (again, non-deterministically in general) the current refinement \( M_i \).

Examples As a first example, consider the ModalModel actor of Figure 1. The controller of ModalModel is the automaton with locations labeled normal and faulty. The refinements of both these locations are FSM actors. As all refinements are untimed, ModalModel is also untimed. The refinement of faulty is a non-deterministic FSM actor, as the outgoing transitions of its heating location have both guard true. The state variables of ModalModel are the location variables of all FSM actors, plus the count parameter (the other parameters, such as heatingRate, etc., should in principle also be included in the state; however, they can be omitted since they remain invariant). A sample of the values that the \( F \) and \( P \) functions of ModalModel take is given below (because of determinism, the index parameter \( j \) is omitted):

\[
F((\text{normal}, \text{heating}, \text{cooling}, 10), (22, \text{faulty})) = -0.05
\]
\[
P((\text{normal}, \text{heating}, \text{cooling}, 10), (22, \text{faulty})) = (\text{normal}, \text{cooling}, 10)
\]
\[
F((\text{normal}, \text{heating}, \text{cooling}, 10), (22, \text{faulty})) = -0.05
\]
\[
P((\text{normal}, \text{heating}, \text{cooling}, 10), (22, \text{faulty})) = (\text{faulty}, \text{cooling}, \text{heating}, 0)
\]
The first two equations correspond to Case 1 whereas the last two equations correspond to Case 3. No preemptive or reset transitions exist in this model.

Another example, that illustrates timed modal models, is shown in Figure 3. This model switches between two modes every 2.5 time units. In the regular mode it generates a regularly-spaced clock signal with period 1.0 (and with value 1, the default output value for DiscreteClock). In the irregular mode, it generates pseudo-randomly spaced events using a PoissonClock actor with a mean time between events set to 1.0 and value set to 2. The result of a typical run is plotted in Figure 4, with a shaded
background showing the times over which it is in the two modes. A number of observations worth making arise from this plot.

First, note that two events are generated at time 0, a first event with value 1, at superdense time (0,0), and a second event with value 2, at superdense time (0,1). The first event is produced by DiscreteClock, according to the semantic rules of Case 3a. If we had instead used a preemptive transition, as shown in Figure 5, then that first output event would not appear: this is according to the semantic rules of Case 2a and the fact that the action of the preemptive transition does not refer to the output port.

The second event is produced by PoissonClock, according to the semantic rules of Case 1. The reason for this second event is the following. When the model is initialized, a timer is set by PoissonClock to value zero: this means that this timer is to expire immediately, i.e., PoissonClock will produce an output immediately when it starts, and at random intervals thereafter. When the irregular state is entered, this timer is resumed and since it has value 0, is ready to expire. This forces a new firing of ModalModel and ultimately of PoissonClock, which produces the event at superdense time (0,1).

Another interesting observation concerns the output events with value 1 occurring at times 3.5, 4.5, 8, and so on. These events occur at times during which the model is in the regular mode. Notice that the model begins in the regular mode but spends zero time there, since it immediately transitions to the irregular mode. Hence, at time 0, the regular mode becomes inactive and the timer of DiscreteClock is suspended. Since no time has elapsed yet, the timer is equal to 1, the value of the period, at this time. When regular is re-entered at time 2.5, this timer is resumed, and expires one time unit later, i.e., at time 3.5. This explains the event at that time. Moreover, the timer is reset to 1 during postfire(), according to Case 1a. It expires again 1 time unit later, which explains the event at time 4.5. Finally, it is reset to 1 at time 4.5, suspended at time 5, and resumed at time 7.5, which explains the event at time 8.

The above examples may appear rather artificial, however, they are given mainly for purposes of illustration of the semantics. More interesting and realistic examples can be found in the open-source distribution of Ptolemy available from http://ptolemy.eecs.berkeley.edu/. Detailed descriptions of some of these examples can be found in other publications of the Ptolemy project. For modal models in particular, we refer the reader to the case studies described in [8].

5. Alternative Modal Model Patterns

It is instructive to briefly discuss alternative definitions of modal model semantics and justify our choices.

First, consider our design choice to have the refinement $M_i$ of location $l_i$ in a modal model $M$ “freeze” while the controller automaton is in a location different from $l_i$. “Freezing” here means that $M_i$ is inactive, in terms of its state which does not evolve at all. This includes in particular the timers of $M_i$, which are suspended until $l_i$ is re-entered. An alternative would be to consider all refinements “live”, but to feed the inputs of $M$ only to the currently “active” refinement, say $M_l$, and to use the outputs of $M_l$ as outputs of $M$. Let us term this alternative as the “non-freezing” semantics, for the purposes of this discussion.

One issue with the non-freezing semantics is that it is redundant from a modeling point of view. Indeed, as we show next, there exists a simple design pattern that allows the non-freezing semantics to be easily implemented in Ptolemy. Since this mechanism already exists, there would be no need to add modal models to get the same semantics. In fact, using different modeling patterns that result in the same semantics, for the purposes of this discussion.

This design pattern, which we call the switch-select pattern, is illustrated in Figure 6. There are five actors in this model, $M_1$, $M_2$, $Controller$, $BooleanSwitch$ and $BooleanSelect$. $M_1$ and $M_2$ represent the refinements of the non-freezing modal model that the pattern captures, and $Controller$ is its controller (which is assumed to have 2 locations in this example). The switch and select actors control the routing of the inputs/outputs to/from either $M_1$ or $M_2$, depending on the state that the $Controller$ is in. The latter may in turn generally depend on outputs of these actors, which is captured by the communication links between $Controller$, $M_1$ and $M_2$.

Another issue with the non-freezing semantics is that it is less modular than the freezing semantics. In the freez-
ing semantics, a subsystem (refinement of a certain location) is completely unaffected by being suspended. In the non-freezing semantics, behavior of a subsystem continues while the latter is inactive, only with absent inputs. Thus the evolution of the subsystem depends on how much time it remains inactive, for instance.

Another design choice could be to have time pass in inactive subsystems, i.e., to have their timers active, while having the rest of their state be frozen. The disadvantage of this approach is that for many components (e.g., DiscreteClock), the state is intrinsically bound to time. It is therefore hard to separate the two notions.

Finally, it is worth mentioning the approach taken in the Simulink/Stateflow tool from the Mathworks. Simulink is a hierarchical block diagram language. Some Simulink blocks can be Stateflow models, that is, hierarchical state machines similar to Statecharts [22]. Simulink blocks, however, cannot be embedded into Stateflow as state refinements. The way to get modal behavior in Simulink/Stateflow is by connecting Stateflow outputs to enable inputs of Simulink blocks. When a block is disabled, it is frozen, as in the Ptolemy semantics. Contrary to Ptolemy, however, the output of a disabled block can still be used as it still exists: it is simply held constant while time passes.

6. Related work

A number of formalisms based on hierarchical state machines (HSMs) have been studied in the literature, including Statecharts [22], SyncCharts [5], and commercial variants such as Stateflow from the Mathworks or Safe State Machines from Esterel Technologies [6] (SSMs are based on SyncCharts). Hierarchical state machines are also one of the diagrams of UML. The main difference of Ptolemy modal models with respect to the above is that in Ptolemy modal model refinements are not restricted to state machines or concurrent state machines (built with AND states). In Ptolemy, refinements can include other domains as well, for instance, as in Figure 3. Note that AND states can still be modeled in Ptolemy, using concurrent ModalModel actors. For instance, the TemperatureModel and ModalModel actors shown in Figure 1 are concurrent: the TemperatureModel could very well be another modal model. Note that in this case, a MoC such as SR or DE must be specified, in order to define the semantics of the composition of these actors.

Even when we restrict our attention to pure HSMs with no concurrency, there are differences between the Ptolemy version and the models above. A variety of different semantics has been proposed for Statecharts for instance, see [10, 16]. Operational and denotational semantics for Stateflow are presented in [21, 20]. Implicit formal semantics of Stateflow by translation to Lustre are given in [34].

Also, contrary to Statecharts, SyncCharts and Stateflow, Ptolemy modal models do not use broadcast events for communication. Guards may refer to input events, however, these events are transmitted using explicit ports and connections, and are evaluated when the fire() or postfire() methods are called (e.g., guard in_isPresent in Figure 3 is evaluated to true or false depending on whether the value of the input is present or absent when the fire() method is called).

Another difference with the above languages is that Ptolemy modal models include both untimed (reactive) and timed (proactive) models. Timed versions of Statecharts and UML (but not general modal models) have been proposed in [12, 17].

The semantics we present are somewhat operational in nature, given by functions that produce outputs and update the state. Our semantics is also abstract, as in Abstract State Machines [18]. Most importantly, our semantics is modular, in the sense that we show how the output and state update functions of composite actors are defined given output and state update functions of sub-actors.

Formal studies of HSMs can be found in [3, 4, 2].

7. Conclusions

We presented a modular and formal framework for Ptolemy, and described the semantics of modal models, as these are implemented in Ptolemy. Modal models allow hierarchical composition of state machines with other MoCs, therefore generalizing hierarchical state machines and enriching heterogeneous modeling with modal behavior.

Existing Ptolemy models emphasize actor semantics, by having an explicit notion of inputs and outputs. This is in contrast to languages such as Modelica, which are based on undirected equations. Note that feedback loops are allowed in Ptolemy, and can be used to capture some form of equational constraints. How these loops are handled depends on the domain used. In the SR and DE domains, for instance, the equations are solved by fixpoint computations, as mentioned above. In the future we intend to study equational constraints in more depth, borrowing ideas from languages such as Modelica. One direction would be to implement a Modelica domain in Ptolemy, which would work by translating Modelica models into, essentially, CT models, and

7 It is worth pointing out that, although it is common to refer to communication in Statecharts as being “broadcast”, this is slightly misleading, since it implies that all processes receive all signals, which is not the case. A more accurate description is “name matching” since, in fact, only those processes that refer to the signal by name receive it. Name matching is as static as ports in Ptolemy, but is less modular (changing the name in one part of the model requires changing it at other places as well). It also requires more effort to identify the communication links between processes (e.g., when determining causality loops in a diagram).
then handle the latter using numerical solvers. This translation could benefit from the code generation framework available in Ptolemy [38].

Although our discussion in this paper focused on discrete-time modal models, Ptolemy currently supports continuous-time models as well, via the CT domain, which allows a number of numerical solvers to be expressed, including those that use backtracking [28, 29]. A formalization of CT using the framework developed in this paper is a topic of future work.

The semantics developed in this paper are operational. It would be interesting to study also a denotational semantics of modal models. The work reported in [20] could be beneficial in that context.

Acknowledgments

Several people contributed to the FSM and modal-model infrastructure in Ptolemy. The modal domain was created primarily by Thomas Huining Feng, Xiaojun Liu, and Haiyang Zheng. The graphical editor in Vergil for state machines was created by Stephen Neuendorffer and Hideo John Reekie. Joern Janneck contributed to the semantics for timed state machines. Christopher Brooks created the online version of the models accessible by hyperlink from this document, and has also contributed enormously to the Ptolemy software infrastructure. Other major contributors include David Hermann, Jie Liu, and Ye Zhou.

References


