Preprint, to appear in Math. Struct. in Comp. Sc. (CUP)

Under consideration for publication in Math. Struct. in Comp. Science

A Modular Formal Semantics for Ptolemy[1] [2]

Stavros Tripakis, Christos Stergiou, Chris Shaver and Edward A. Lee

University of California, Berkeley

{stavros,chster,shaver,eal}@eecs.berkeley.edu

Received 23 August 2012

Ptolemy is an open-source and extensible modeling and simulation framework. It offers heterogeneous modeling capabilities by allowing different models of computation, both untimed and timed, to be composed hierarchically in an arbitrary fashion. This paper proposes a formal semantics for Ptolemy which is modular, in the sense that atomic actors and their compositions are treated in a unified way. In particular, all actors conform to an executable interface that contains four functions: fire (produce outputs given current state and inputs), postfire (update state instantaneously), deadline (how much time the actor is willing to let elapse) and time-update (update state with passage of time). Composite actors are obtained from composition operators that in Ptolemy are called directors. Different directors realize different models of computation. This paper defines formally the directors for the following models of computation: Synchronous-Reactive, Discrete Event, Continuous Time, Process Networks, and Modal Models.

Contents

[1] Ptolemy in this document refers to Ptolemy II, see http://ptolemy.org. If you are reading this document on screen (vs. on paper) and you have a network connection, then you can click on the figures showing Ptolemy models to view, execute and experiment with those models online. There is no need to pre-install Ptolemy or any other software.

[2] This work is partially supported by NSF project *ExCAPE: Expeditions in Computer Augmented Program Engineering*, and by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET) and #0931843 (ActionWebs), the U.S. Army Research Office (ARO #W911NF-11-2-0038), the Air Force Research Lab (AFRL), the Multiscale Systems Center (MuSyC), one of six research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program, and the following companies: Bosch, National Instruments, Thales, and Toyota.

1. Introduction

Modeling has always been an essential component of system design. Building models of systems before or even after building the systems themselves is beneficial for a number of reasons. The model provides a means for experimenting with a virtual version of the system, analyzing its behavior, and asking "what-if" questions. Therefore, having a model of the system before actually building the system allows to make design decisions based on the results of the analysis. On the other hand, having a model of an existing system allows to subject the model to experimentation that the physical system cannot be subjected to, for various reasons (cost, size, time scales, etc.). Such experimentation can influence decisions such as adjustments to be made to the system and future system evolution.

Building large and complex systems is not a trivial task. This task is often accompanied by the task of building large and complex models, which is itself non-trivial. One of the main difficulties of the modeling task comes from the fact that a large system cannot be modeled in a *monolithic* way. That is, instead of developing a single model that captures the entire system, one develops many smaller models, for parts of the system. These sub-models need to be combined somehow into a single model. We refer to this problem as the problem of *model composition*.

Model composition may be easier (but by no means easy!) when the models to be composed are of the same nature, or *homogeneous*. Homogeneity comes in different flavors:

— Homogeneity may be *linguistic* in the sense that the models to be composed are written in the same language. In this case, the language typically provides some composition operators which allow to compose these models and form a larger model.[1]

— Homogeneity may be *syntactic*, meaning that the models, even though they may be written in different languages, share the same syntax or have similar syntaxes. For instance, a Simulink [2] model can be written in a block diagram notation, and so can a SysML "block definition diagram".[3] The fact that the two models share a similar notation, however, does not imply

[1] Even in this simplest case, composition may not be entirely straightforward. This is because existence of composition operators does not ensure that the language is *compositional*, in the sense that an arbitrary composition of models can be represented as an atomic (i.e., non-composite) model. Indeed, many languages are not compositional in this sense, for instance, see (Lublinerman et al., 2009; Tripakis et al., 2010).

[2] `http://www.mathworks.com/products/simulink/`

[3] `http://www.omgsysml.org/`

that it is easy to compose them, as this composition strongly depends on the semantics of the corresponding notations, as well as on the desired semantics of the composition.

— Homogeneity may be *semantic*, meaning that the models share the same semantics, even though they may have different syntaxes. For instance, a model written in a state-machine notation may have a different syntax than a model written in the synchronous language Lustre (Halbwachs et al., 1991), but they can both be given semantics in terms of sets of synchronous input-output traces. This makes it easier to compose the models semantically, but it is unclear how to do so syntactically. Syntax does matter in modeling and system design. As an extreme thesis, one may claim that every model executable in a computer could be encoded as a Turing machine, therefore Turing machines are the ultimate unifying modeling language! But this language is of course not very useful.
  Another practical problem with composing semantically homogeneous models but which are not written in the same language or syntax is tool support. Often the individual models can be handled by separate tools, but there is no tool that can handle the composition. A number of attempts have been made in the past to build tool "bridges" (e.g., in the context of the EU project SPEEDS[4] or the earlier US project MoBIES[5] unfortunately with limited success).

In practice models are often *heterogeneous*, in any of the senses mentioned above. That is, they may have different syntaxes, semantics, or both. Heterogeneous models arise naturally because different parts of the system have inherently distinct properties, and therefore require different types of models. For instance, the dynamics of a car is natural to capture using a continuous-time model, whereas a computerized controller is more natural to describe in discrete-time. If the controller is implemented in hardware (say, as a synchronous digital circuit) or as a single read-compute-write software control loop, then it may be easier to describe in Lustre or Simulink, whereas if it is implemented as a set of concurrent threads, it may be easier to capture as a Kahn Process Network (Kahn, 1974). Another reason for heterogeneity is also the fact that different models are often built by different groups of people, with different traditions or processes.

The term *model of computation* (MoC) can be defined as the set of rules used to obtain a semantically well-defined composite model from a set of sub-models.[6] Thus, a MoC can be seen as providing a solution to the model composition problem for homogeneous models. 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 system design, by offering mechanisms to capture concurrency, interaction, and time behavior, all of which are essential aspects of modern systems. 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, the above languages offer little or no support for heterogeneity. Currently, no universally accepted solution exists for heterogeneous modeling.

[4] `http://www.speeds.eu.com/`
[5] `http://w3.isis.vanderbilt.edu/Projects/mobies/`
[6] Often the term *model of concurrency and communication* (MoCC) is used instead.

The modeling and simulation tool Ptolemy [7] has been a pioneering, long-term, and on-going effort to provide a solution to the model composition problem in the presence of heterogeneity (Eker et al., 2003; Lee, 2010). 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) and the exact manner in which they communicate (e.g., through message passing or shared variables) are not fixed but are defined by an MoC, also called *domain*, in Ptolemy terminology. Each domain is implemented in the tool by a *director*, which coordinates the execution of a set of actors as well as their communication. Ptolemy is written in Java, and it is open-source and free. It is also architected to be easily extensible: new domains (i.e., new directors) and new actors can be added with relatively small effort.

Currently, Ptolemy supports a number of MoCs and corresponding domains, including *synchronous data flow* (SDF), *synchronous reactive* (SR), *discrete event* (DE), *process networks* (PN), *continuous time* (CT), *extended state machines* (ESM), and *modal models* (MM). A rich body of literature presents formal semantics for all of these MoCs (see Section 2 for references). However, no unified formal semantics of Ptolemy has been provided so far. By "unified" we mean a semantics that can encompass more than one, and in principle all, the domains implemented in Ptolemy.

Ultimately, the semantics of a tool like Ptolemy is derived by its implementation, i.e., by "what the simulator does". This is the case for every tool that implements a language, even one with a formal semantics, since the question of conformance of the implementation to the semantics of the language is always a tricky one. Despite this inherent difficulty, a formal semantics is desirable to have, for many reasons that we will not repeat here as these have been well argued before (e.g., in (Floyd, 1967; Dijkstra, 1976)). Suffice it to say that we view *conciseness* and *readability* as two of the most important reasons. It is much easier to read and understand a few pages of formalism than many thousands of lines of Java code.

In this paper we propose a formal semantics for Ptolemy that unifies a number of domains, in particular, SR, DE, CT, PN and MM. These domains have been chosen as they represent a significant subset of Ptolemy, as well as the most often used subset. Apart from that, they also represent significantly heterogeneous models of computation. SR has a synchronous, "untimed" (or "logical-time") semantics akin to that of the *synchronous languages* (Benveniste and Berry, 1991; Halbwachs et al., 1991; Benveniste et al., 2003). DE has a timed semantics based on streams of timed events (Yates, 1993; Lee, 1999). CT approximates continuous-time semantics using numerical solvers for differential equations. PN is based on Kahn Process Networks (KPN) (Kahn, 1974), which model asynchronous concurrent processes communicating via FIFO queues. And MM capture control with state machines or hierarchical state machines (Harel, 1987; André, 1996). We believe that the approach proposed in this paper is not limited to the above domains and could be extended to other MoCs as well. For instance, SDF can be seen as a static subclass of KPN and therefore could be captured semantically as such.[8]

Ptolemy uses a graphical syntax, with *hierarchy* being the fundamental modularity mechanism

[7] http://ptolemy.eecs.berkeley.edu/.
[8] An implementation would typically distinguish SDF from PN for reasons of efficiency, since SDF admits specialized algorithms for scheduling and analysis.

at the syntactic level. This means that a model is essentially a tree of submodels. The leaves of the tree correspond to *atomic* actors, available in the Ptolemy library of predefined actors or written in Java by users. The internal nodes of the tree correspond to *composite* actors, which are formed by composing other actors using the graphical syntax into an *actor diagram* (see Figures 3, 5, etc., for examples).

Our semantics is designed to mirror this syntactic modularity mechanism. The semantics is *modular* in the sense that it treats composite actors and atomic actors in a unified way. This is achieved by identifying a unique formal "interface" (or "signature") that characterizes all actors, viewing them as extended, timed state machines. These machines are similar to the *abstract state machines* of (Gurevich, 1993). They are characterized by a *fire* function F that produces outputs based on the state and the inputs, and a *postfire* function P that updates the state based on the same information. These are the standard functions found in state machines of type Mealy (Kohavi, 1978). Our machines also include a *deadline* function D and a *time-update* function T, which capture timed actors and the effect that passage of time has on the state. A summary of the standardized actor interface is given in Figure 1 for reference purposes. A detailed description of the interface is given in Section 4.

Spaces

space	*description*
$\widehat{I}$	Input valuations
$\widehat{O}$	Output valuations
$\widehat{S}$	State valuations
$s_0 \in \widehat{S}$	Initial state

Interface

function	*type*	*description*
F	$\widehat{S} \times \widehat{I} \to \widehat{O}$	Form outputs from inputs and state.
P	$\widehat{S} \times \widehat{I} \to \widehat{S}$	Update state from inputs and state.
D	$\widehat{S} \times \widehat{I} \to \mathbb{R}_+^\infty$	Return firing deadline.
T	$\widehat{S} \times \widehat{I} \times \mathbb{R}_+ \to \widehat{S}$	Update state over a given time.

Fig. 1. Actor interface: $\widehat{X}$ denotes the set of all valuations over set of variables X.

Directors are viewed as *composition operators*: they take as input an actor diagram and return a new actor as output. The returned actor is a composite actor, but obeys the same interface as atomic actors and is therefore indistinguishable from the latter. In this way, we can define the semantics of hierarchical Ptolemy models of arbitrary depth and domain combinations. In particular, for each director we will show how the functions F, P, D, T of the composite actor are defined in terms of the same functions of its sub-actors. The main ideas behind these definitions are summarized in Figure 2, for reference purposes. Detailed descriptions of directors are provided in Section 6.

Modularity greatly enhances conciseness and readability, because in order to understand the semantics of a particular MoC it suffices to understand the semantics of the corresponding director. This is often difficult to achieve by looking at the implementation of the director, as the Java code has interdependencies that extend beyond the particular director class and require looking at extensive parts of the code.

Our semantics, although not directly reflecting the implementation (this would mean formalizing the Java code!) aims to come as close to the implementation as possible while maintaining a high-level view that allows to achieve conciseness and readability. In particular, our formalization

	fire F	*postfire P*	*deadline D*	*time-update T*
SR	least fixpoint on flat CPO	P of components	∞ or fixed step	state unchanged
DE	same as SR	same as SR	min D of components	T of components
CT	same as SR	same as SR	step determined by ODE solver	state updated by ODE solver
PN	least fixpoint on stream CPO	same as SR	∞ or fixed step	state unchanged
MM	possibly F of source refinement, and transition action	possibly P of source refinement, and transition action	D of (target) refinement	T of (target) refinement

Fig. 2. Main ideas of composite actor definitions for the different Ptolemy domains.

captures the salient features of the implementation, and in particular the fire/postfire interface of actors, called the *abstract semantics* (Eker et al., 2003).

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 briefly reviews the visual syntax of Ptolemy through an example. In Section 4 we provide a formal semantics for actors. Section 5 formalizes actor diagrams. Section 6 formalizes directors for the SR, DE, CT, PN and MM domains. Section 7 concludes the paper.

2. Related Work

The semantics of Ptolemy II has been described previously in the literature in a number of papers. These papers differ from the current paper in the following ways.

— In some papers the semantics is presented in an informal or incomplete manner. For instance: (Eker et al., 2003) discusses the principles of Ptolemy's abstract semantics and domain polymorphism, but does not provide a formal semantics. It also limits its discussion to a restricted actor interface that contains only fire and postfire, which are not sufficient to cover timed actors. (Lee and Zheng, 2007) discusses the implementation principles and commonalities of SR, DE and CT, including a discussion of the fireAt method for timed actors. However, this paper does not provide a formal semantics either, nor a complete implementation policy. A recent informal discussion can also be found in (Goderis et al., 2009).

— In other papers, formal semantics is presented for individual domains of Ptolemy. (Edwards and Lee, 2003) presents the basis for the SR semantics. Different variants of formal semantics for DE are presented in (Lee, 1999; Liu et al., 2006; Cataldo et al., 2006; Liu and Lee, 2008; Bae et al., 2010), the last paper being the closest to the actual Ptolemy implementation. Continuous-time and hybrid systems are considered in (Liu and Lee, 2003; Lee and Zheng, 2005). (Lee and Tripakis, 2010) presents a formal semantics for modal models (an informal description can be found in (Lee, 2009)).

— Some papers present formal semantics that unify more than one models of computation. In particular, a denotational semantics is proposed in (Lee and Sangiovanni-Vincentelli, 1998) where processes are seen as relations between signals, where a signal is a set of *tagged* events (events with timestamps in some abstract time domain). Only a single composition operator, essentially based on intersection, is provided. It is difficult to see how this can capture different domains which, as mentioned above, are viewed as different composition operators in this paper.

Another denotational semantics is proposed in (Liu and Lee, 2008), based on fixpoints on CPOs with a prefix order. This allows this semantics to be applied directly to a number of domains that can be naturally described using CPOs, in particular SR (Edwards and Lee, 2003) and PN (Kahn, 1974). The authors also show how to incorporate timed systems (e.g., DE) in the same framework. A similar denotational semantics is proposed in (Benveniste et al., 2009) with the difference that a special "absent" value is not used.[9] It is unclear how this work can be extended to include other MoCs, in particular continuous-time and modal models.

Finally, an abstract framework reminiscent of trace theory (Dill, 1988) is provided in (Burch et al., 2001). The latter can be seen as a "meta-framework" under which the heterogeneous composition of specific MoCs can be formulated, but does not include such formulations for the MoCs considered in this paper.

An additional concern with some (although by no means all) of the above papers is also how close the formal semantics is to the actual tool implementation. This is particularly a concern with papers that present denotational semantics. Although we do not pretend in this paper to present a semantics that exactly captures the tool implementation, we believe our semantics is much closer to the implementation than in previous works.

Aside from the above literature, most of which focuses on the Ptolemy tool in particular, a rich body of research is concerned with the semantics of the individual MoCs considered in this paper. Our work builds upon all this previous work, our focus being to develop a composable semantics that integrates multiple models of computation. We next list the relations of our work to the previous work for each of the MoCs considered in this paper.

The semantics of SR is strongly related to those of the synchronous languages (Benveniste and Berry, 1991; Halbwachs et al., 1991; Benveniste et al., 2003), and in particular the idea of *constructive* semantics of Esterel (Malik, 1994; Berry, 1996; Shiple et al., 1996). These ideas have been adapted in (Edwards and Lee, 2003) for a block-diagram notation, which is also the one used in Ptolemy in the case of SR. Our semantics follows the one of (Edwards and Lee, 2003) and extends it to "open" systems, in the sense that a composite block can have inputs. The theory of fixpoints of Scott-continuous functions on CPOs (complete partial orders) is used to give an unambiguous meaning to models with feedback loops. Feedback loops may result in causality cycles, but these are resolved by adding a special "bottom" value $\perp$ representing an unknown value. As a result, the set of values becomes a "flat" CPO with $\perp$ being the smallest element and all other values being incomparable. A monotonic function in this CPO is guaranteed to have a unique least fixpoint, and this is defined to be the semantics of a model.

[9] This results in some loss of expressiveness, attested by the fact that the Adder actor cannot be specified in a satisfactory manner.

The semantics of PN is based on Kahn Process Networks (Kahn, 1974). This semantics is also given in terms of the least fixpoint of a continuous function on a CPO, however, the CPO is different here than the one used in SR. In SR, inputs and outputs are individual values, whereas in PN they are *streams*, i.e., finite or infinite sequences of values. Streams are ordered with the prefix order, and the empty sequence is the minimal element of the corresponding CPO. The stream CPO is not flat, in fact, it has infinite height (since finite streams can be of arbitrary length). As a result, monotonicity of functions does not generally imply Scott-continuity. Scott-continuity is a reasonable assumption to make, however, and it is satisfied by actors in practice.

The PN semantics is a denotational semantics. The stream CPO has infinite height, therefore, the least fixpoint may not be reachable in a finite number of iterations. In fact, problems such as deciding whether in a given PN model the length of a produced stream is finite are undecidable (Buck, 1993). Algorithmic ways of executing a PN model that satisfy different properties are provided in (Lee and Parks, 1995) and (Geilen and Basten, 2003). The semantics of PN is unified with the semantics of dataflow models in (Lee and Matsikoudis, 2009). Reactive process networks which extend process networks with event-based control are defined in (Geilen and Basten, 2004).

The semantics of DE has been an old topic of discussion (Reed and Roscoe, 1988; Yates, 1993), which is also related to fixpoint semantics based on metric spaces or CPOs (Arnold and Nivat, 1980; Baier and Majster-Cederbaum, 1994). The DE domain is related to other models of computation that have a dense-time semantics. An example is timed automata (Alur and Dill, 1994). A timed automaton has a finite number of clocks, whereas in DE a separate clock may be required for each token, which makes the number of clocks a-priori unbounded. For this reason, DE models are not directly representable as timed automata. They could be representable as some form of timed Petri nets (Sifakis, 1977) but to our knowledge, this link has not been explored yet.

SR, PN and DE have semantical similarities that have been explored and exploited in the literature (Broy and Stolen, 2001; Liu and Lee, 2008; Benveniste et al., 2009). Particularly relevant is the work on FOCUS (Broy and Stolen, 2001) which offers a general framework for specifying systems based on stream-processing elements. FOCUS can capture both untimed and timed systems, as well as asynchronous and synchronous systems. It provides formal refinement relations and guarantees of compositionality.

The semantics of CT is based on numerical methods for solving differential equations. Combinations of CT models with discrete logic (e.g., modal models) result in models similar to hybrid systems (Manna and Pnueli, 1992). The faithful reproduction of the semantics of such systems by a computer, for instance, by simulation, is a difficult problem and still an active area of research (e.g., see (Zhu et al., 2010)). (Liu and Lee, 2003) uses the notion of an *ideal solver*, which can solve a set of differential equations exactly, provided the equations satisfy a Lipschitz condition over a given time interval. This is not as far-fetched as it might sound, because closed form expressions can sometimes be given for the solution over the intervals of continuous behavior. Even when we do not have closed form solutions, for many special cases, numerical solutions yield exact answers (using appropriate solvers). But even in cases where the solution must be approximated, it is valuable to separate the issue of approximate ODE solutions from the other semantic issues (such as determinacy of the model). Hence, the idealization remains useful.

Modal models are based on hierarchical state machines, various versions of which have been studied in the literature or are available as commercial products, including Statecharts (Harel, 1987), SyncCharts (André, 1996), Stateflow from the Mathworks, Safe State Machines from Esterel Technologies (André, 2003) (SSMs are based on SyncCharts), and UML state machines. A variety of different semantics has been proposed for Statecharts for instance, see (Beeck, 1994; Eshuis, 2009). Some of these semantics are synchronous in nature. SyncCharts also uses a synchronous semantics. An alternative for incorporating mode switching into synchronous languages is presented in (Maraninchi and Rémond, 2003). The semantics of Stateflow are based on the "run-to-completion" principle which is not really synchronous, although it can be approximated by a synchronous model (Scaife et al., 2004). Operational and denotational semantics for Stateflow are presented in (Hamon and Rushby, 2004; Hamon, 2005). Timed versions of Statecharts and UML have been proposed in (Damm et al., 1998; Graf et al., 2006). In Ptolemy modal models the hierarchy is not restricted to contain only state machines or concurrent state machines (built with AND states). Also, contrary to Statecharts, SyncCharts and Stateflow, Ptolemy modal models do not use *broadcast* events for communication. Instead, communication is done via *ports*, as in the block-diagram based notation of Ptolemy.

A number of other modeling frameworks exist that provide mechanisms for mixing MoCs. An early systematic approach to such mixed models was realized in Ptolemy Classic (Buck et al., 1994). The Metropolis environment (Balarin et al., 2003; Goessler and Sangiovanni-Vincentelli, 2002) focuses on modeling both function and architecture, as well as the mapping of the former to the latter. Metropolis includes the concepts of constraints and quantity managers that are used to constrain the behaviors of a model and annotate them with quantities such as time, energy or other metrics. The Generic Modeling Environment (GME) (Karsai, 1995; Nordstrom et al., 1999; Ledeczi et al., 2001) uses metamodeling techniques to create domain-specific modeling and program synthesis environments. BIP (Basu et al., 2006; Bliudze and Sifakis, 2008a; Bozga et al., 2009) models are built by composing behavioral components with n-ary rendezvous based interactions and then restricting those interactions using priorities. An important problem that researchers working on BIP have tackled is that of *glue expressiveness*, namely, what is the relative expressive power of two modeling formalisms with same sets of basic components but different composition operators (Bliudze and Sifakis, 2008b). Specifying interaction as a first-class citizen is also at the heart of the Reo model of concurrency (Arbab, 2004). In Reo, complex interaction protocols (*connectors*) can be formed by combining simpler protocols (*channels*), such as bounded/unbounded and lossless/lossy versions of FIFO queues. Composition is performed by creating channels and connecting their end-points in a graph-oriented manner using operators such as join or split. Glue expressiveness has been studied in the context of Reo as well: (Arbab, 2004) shows examples of how protocols that can be expressed as regular expressions over I/O operations can also be captured by Reo connectors composed of five primitive channels. The ModHel'X environment (Hardebolle et al., 2007; Boulanger et al., 2011) shares a number of concepts with Ptolemy, such as hierchical composition of MoCs, and emphasizes the use of interface blocks that perform "semantic adaptation" between heterogeneous models. For instance, when embedding an SDF model within a DE model, an interface block can be used to add timestamps to the typically untimed outputs of SDF. ForSyDe (Jantsch, 2003; Sander and Jantsch, 2004) provides a set of libraries for capturing heterogeneous MoCs based on the functional programming language Haskell. ForSyDe includes different model transformations that are used to refine an abstract specification model into a detailed implementation model, which can be translated into a target implementation language. SystemC is capable of realizing multiple

MoCs with a discrete-event simulation flavor (Patel and Shukla, 2004; Herrera and Villar, 2006). An interesting way of expressing the semantics of a MoC is given by "42" (Maraninchi and Bhouhadiba, 2007), which integrates with an application model a specification of a customized MoC. A mechanism to create domain-polymorphic components similar in spirit to Ptolemy is proposed in (Feredj et al., 2009). Close in spirit to this paper are also the works (Bliudze and Krob, 2009; Aiguier et al., 2011), whose goal is to provide a sound semantical framework for heterogeneous systems, in particular with respect to the integration of systems operating at different time dimensions and scales. In (Bliudze and Krob, 2009) a system (in our terms, actor) is captured as a kind of timed Turing machine, where non-standard analysis is used to represent continuous time via *infinitesimals*. In (Aiguier et al., 2011) a system is captured as a kind of timed Mealy machine. The interface of these machines contains two functions, an output function and a state-update function, as in a Mealy machine, except that time is an additional input argument to both functions. Composition in this framework is achieved by three operators: parallel composition, feedback and abstraction. Therefore, in the approach of (Aiguier et al., 2011), the different MoCs are not realized by the composition operators (as is the case with Ptolemy).

Perhaps the work most closely related to our paper is (Denckla and Mosterman, 2008). There, the authors present two types of semantics for a block-diagram language with hierarchy: a stream-based semantics where blocks are viewed as functions from streams to streams; and a state-based semantics where each block is represented by an initial state and a kind of 'step' function which, given the current input and state, returns the current output and an 'implicit output'. For discrete systems the latter is interpreted as 'next state', whereas for continuous systems it is interpreted as the time derivative of the state. A solver is then used to transform continuous systems to discrete systems. The state-based semantics of (Denckla and Mosterman, 2008) is closely related to the semantics we present in this paper. However, the (single) step function used there is different from our 4-function actor interface. In particular, their step function does not appear to be able to explicitly manipulate time (e.g., to specify a deadline).

Comparing the above frameworks among themselves as well as with Ptolemy, stating precisely the strengths and weaknesses of each, is a difficult task. This is partly because each of the above projects pursues slightly different goals, ranging from "pure" modeling and simulation, to verification, to design-space exploration, mapping and implementation. Ptolemy focuses on modeling and simulation, and leverages external tools (e.g., model-checkers) and code-generators for other tasks (e.g., verification).

In terms of expressiveness, many frameworks are equivalent in the sense of being Turing-complete. However, other types of expressiveness may be more appropriate in the context of heterogeneous modeling, such as glue expressiveness (Bliudze and Sifakis, 2008b). A formal comparison of the semantics of Ptolemy viewed as a kind of "glue" and compared to other glues is beyond the scope of this paper, but a worth-pursuing future research direction.

Another, perhaps a more fundamental issue is that modeling and design are ultimately creative tasks and therefore inherently subjective to human taste, experience, and other factors. Which types of designs are easier to build or more intuitive to understand in each of the above approaches? This is at least as difficult as the question: which kind of programs are easier to write in each of the existing programming languages. Most designs could in principle be captured in any of the frameworks listed above, and even in homogeneous modeling frameworks, using clever encodings. The question is how much effort is required to do so, as well as to understand the result, modify it when necessary, use it for analysis or implementation, and so on. Ptolemy strives to offer a framework which is as general as possible (integrating many MoCs), yet at the same time as intuitive as possible, so that an individual model written in, say, SDF or CT, behaves in Ptolemy as one familiar with SDF or CT would expect it to behave.

Finally, a number of component-oriented frameworks come from the fields of traditional programming and software engineering, e.g., object-oriented programming languages such as Eiffel (Meyer, 1992), component diagrams in UML and other notations, and component models such as CORBA CCM, .NET, EJB, or Fractal (Bruneton et al., 2006), to name a few. The common characteristic that these frameworks have with ours is that they also provide notions of standardized interfaces, from the level of notation, as with UML-based frameworks, to the level of execution, as with concrete implementations of frameworks like Fractal. The above frameworks have a variety of objectives, which are quite different from ours. Our main goal is to come up with the right actor interface to express the behavioral semantics of many different MoCs.

3. Ptolemy's Graphical Syntax

Before presenting the formal semantics, we give a brief overview of Ptolemy's graphical syntax, via an example, shown in Figure 3. There are 9 actors in this model:

1 The top-level actor is a composite actor, composed with the DE director. This top-level actor contains three sub-actors.
2 The `DiscreteClock` actor with period 2.5 embedded in the top-level actor.
3 The `TimedPlotter` actor embedded in the top-level actor.
4 The `ModalModel` actor embedded in the top-level actor. The `ModalModel` actor is also a composite actor, composed with the MM director.
5 The controller automaton of the `ModalModel` actor, which has two locations, `regular` and `irregular`.[10]
6 The composite actor refining state `regular` of `ModalModel`. This composite actor is composed with the DE director.
7 The composite actor refining state `irregular` of `ModalModel`. This composite actor is composed with the DE director.
8 The `DiscreteClock` actor with period 1.0 embedded in the composite actor refining `regular`.
9 The `PoissonClock` actor embedded in the composite actor refining `irregular`.

Out of these 9 actors, 4 are composite actors and 5 are atomic actors.[11] Each composite actor is associated with an actor diagram, therefore, the model also contains 4 actor diagrams.

Actors in Ptolemy have *ports*, rendered graphically as small triangles attached to the "boxes" that represent the external view of actors. In composite actors, ports appear also internally in the corresponding actor diagram. Ports can be *input* or *output*. For example, `DiscreteClock` has four input ports and one output port. `ModalModel` has one input port and one output port.

[10] For state machines and modal models, we use the term *location* instead of *state*, in order to distinguish it from the semantical concept of state (defined formally in Section 4).

[11] We classify the automaton of the modal model as an atomic actor. It could also be classified as a composite actor, composed from basic states and transitions, but this would make things more complex than necessary.

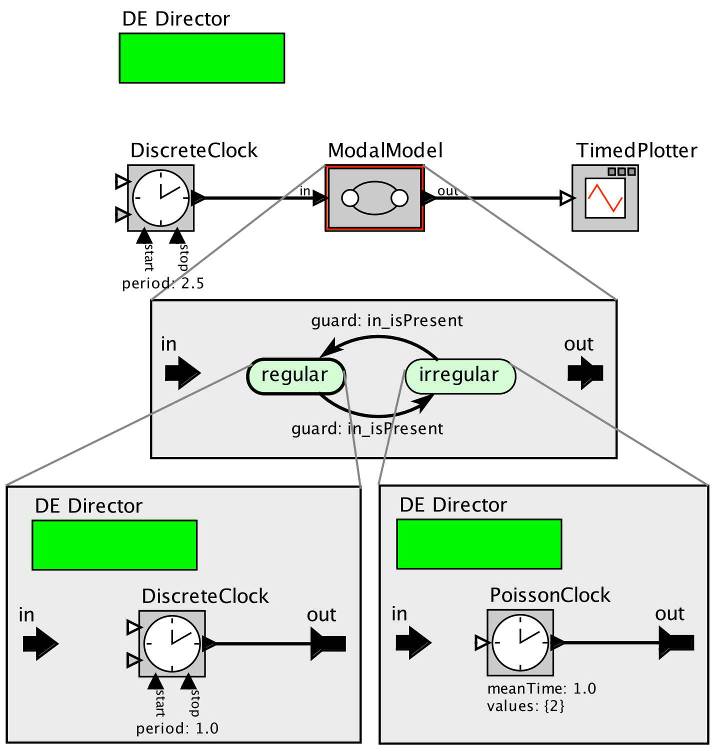


Fig. 3. A Ptolemy model.

Actors can have parameters which may be instantiated with different values at different instances of the actor. An example is the period parameter of the two `DiscreteClock` actors in the model of Figure 3. In Ptolemy parameters can be modified dynamically (during execution). In our model, we will represent such dynamically modifiable parameters as state variables of an actor (see examples of actors in Section 4). We will reserve the term *parameter* in our model for *static* parameters whose value does not change once an actor is instantiated.

State machines in Ptolemy consist of a finite set of locations, one of which is the initial location. Initial locations are indicated by a bold outline; the initial location of the state machine in Figure 3 is `regular`. 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. We omit a detailed formal definition of guards and actions, as they are standard in most modeling languages.

Transitions of Ptolemy state machines can be of different types, including *default* transitions that are to be taken when no other transition is enabled, *reset* transitions that result in the refinement of the destination location being reset to its initial state, and *preemptive* transitions (indicated by a red circle at the start of the transition) which "abort" the execution of the current refinement (see Section 6.5 for the precise semantics). Both transitions shown in Figure 3 are non-preemptive and non-reset. A preemptive transition is shown in Figure 12.

Briefly, the behavior of the model of Figure 3 is that the `ModalModel` actor switches between two modes of operation every 2.5 time units: in the `regular` mode it generates a regularly-spaced clock signal while in the `irregular` mode, it generates pseudo-randomly spaced events, as illustrated in Figure 11. See Section 6.5 for a more detailed description of the behavior of this model.

4. Actors

4.1. *Variables, Assignments and Timers*

Let S be a set of variables (more precisely, variable names). We will assume that all variables take values in some universe of values $\mathcal{U}$. A *valuation* (or *assignment*) over S is a function $x : S \to \mathcal{U}$ that assigns to each variable $v \in S$ some value $x(v) \in \mathcal{U}$. The set of all assignments over S is denoted $\widehat{S}$. Note that if S_1 and S_2 are disjoint sets of variables, then $\widehat{S_1 \cup S_2}$ is isomorphic to $\widehat{S_1} \times \widehat{S_2}$. If $x_1 \in \widehat{S_1}$ and $x_2 \in \widehat{S_2}$ we write (x_1, x_2) for the valuation $x \in \widehat{S_1 \cup S_2}$ such that $x(v_1) = x_1(v_1)$ for all $v_1 \in S_1$ and $x(v_2) = x_2(v_2)$ for all $v_2 \in S_2$. If $S' \subseteq S$ and $x \in \widehat{S}$ then $x \upharpoonright_{S'}$ is the *restriction* (or *projection*) of x to S', that is, the valuation $x' \in \widehat{S'}$ such that $x'(v) = x(v)$ for all $v \in S'$.

We use the following notation for valuations. If $x \in \widehat{S}$, $v \in S$, and $\alpha \in \mathcal{U}$, then $\{x \mid v \mapsto \alpha\}$ denotes the new valuation x' obtained from x by setting v to α and leaving other variables unchanged. A new valuation is denoted by listing the assignments for all variables in S. For example, if $S = \{v_1, v_2\}$ and $\alpha_1, \alpha_2 \in \mathcal{U}$, then $\{v_1 \mapsto \alpha_1, v_2 \mapsto \alpha_2\}$ denotes the valuation $x \in \widehat{S}$ such that $x(v_1) = \alpha_1$ and $x(v_2) = \alpha_2$.

We will often use a special type of variables called *timers*. Timers are implicitly typed to take values in $\mathbb{R}_+$, the set of non-negative real numbers. We use $\mathbb{R}_+^\infty$ to denote the set $\mathbb{R}_+ \cup \{\infty\}$, where ∞ denotes (positive) infinity.

Two special values in $\mathcal{U}$ are $\bot$, representing "bottom" or "unknown", and absent, representing the "absence" of a signal at a particular point in time. Unknown values are useful when defining the semantics of diagrams of actors that contain feedback loops, as the fixpoint of some function. We define such fixpoint semantics for SR, DE and CT (see Sections 6.1, 6.2 and 6.3). Absent values are useful in models with discrete events, where at any given time either an event occurs or it does not: in the former case, the corresponding signal is *present* (and assumes some value), whereas in the latter case, the signal has value absent. Note that the concepts of absent and unknown are very different. A signal that takes absent values is perfectly legal in a model. However, a signal that is sometimes unknown corresponds to a "bad", ambiguous model. In the rest of the paper we present concrete examples of actors and models that manipulate these values.

4.2. *Actors*

An actor is a tuple

$$A = (I, O, S, s_0, F, P, D, T) \tag{1}$$

where I is a set of *input variables*, O is a set of *output variables*,[12] S is a set of *state variables*, $s_0 \in \widehat{S}$ is a valuation over S representing the *initial state*, and F, P, D, T are total functions with the following types:

$$\begin{aligned} F &: \widehat{S} \times \widehat{I} \to \widehat{O} && (2) \\ P &: \widehat{S} \times \widehat{I} \to \widehat{S} && (3) \\ D &: \widehat{S} \times \widehat{I} \to \mathbb{R}_+^\infty && (4) \\ T &: \widehat{S} \times \widehat{I} \times \mathbb{R}_+ \to \widehat{S} && (5) \end{aligned}$$

We assume that I, O, S are pair-wise disjoint, i.e., $I \cap O = I \cap S = O \cap S = \emptyset$. We use the terms *input*, *output*, *state* to mean valuations over I, O, S, respectively. For example, $x : I \to \mathcal{U}$ is an input, $y : O \to \mathcal{U}$ is an output, and $s : S \to \mathcal{U}$ is a state.

Note that any of the sets of variables I, O, S may be empty or infinite. By convention, the set of valuations over an empty set of variables is a singleton, i.e., a set with a single element that we will denote $*$. Even if all its sets of variables are finite, an actor need not be *finite-state*, since its *state space*, i.e., $\widehat{S}$, can still be infinite. This is because the domains of variables can be infinite. Similarly, the input and output spaces can be infinite.

F, P, D and T are called the *fire, postfire, deadline* and *time-update* functions of A, respectively. F and P are similar to the output and transition functions of a state machine. F produces an output given a state and an input. P produces a new state, given the same information as F. D returns a *deadline*, indicating how much time the actor is willing to let elapse. T updates the state given information on the actual delay chosen by the environment. Delays and deadlines are useful to model the semantics of *timed* actors. Their role should become clear when we explain timed behaviors below.

4.3. *Actor Behaviors*

An actor $A = (I, O, S, s_0, F, P, D, T)$ defines a set of behaviors. Our model of behaviors is inspired from the semantic models of timed or hybrid automata (Alur et al., 1995). A *timed behavior of* A is a sequence

$$s_0 \xrightarrow{x_0/y_0} s_0' \xrightarrow{x_0'/d_0} s_1 \xrightarrow{x_1/y_1} s_1' \xrightarrow{x_1'/d_1} s_2 \xrightarrow{x_2/y_2} s_2' \xrightarrow{x_2'/d_2} \cdots \tag{6}$$

where for all $i \in \mathbb{N}$, $s_i, s_i' \in \widehat{S}$, $d_i \in \mathbb{R}_+$, $x_i \in \widehat{I}$, $y_i \in \widehat{O}$, and

$$\begin{aligned} y_i &= F(s_i, x_i) && (7) \\ s_i' &= P(s_i, x_i) && (8) \\ d_i &\leq D(s_i', x_i') && (9) \\ s_{i+1} &= T(s_i', x_i', d_i) && (10) \end{aligned}$$

The intuition is as follows. Suppose that at some point in time, say $t \in \mathbb{R}_+$, A is at state s_i. The environment provides input x_i to A and A instantaneously produces output y_i using its F

[12] In Ptolemy terminology, the term *ports* is used for input and output variables.

function, and moves to state s'_i using its P function. The environment then proposes to advance time and "asks" A whether it has any restrictions on the amount of time that may elapse. A "replies" by returning a deadline $D(s'_i, x'_i)$ on the amount of time that may elapse. To compute this deadline, A may in general use input value x'_i which is provided by the environment. This value can be viewed as an estimate of the environment of the value of input variables during the next interval of time. Next, the environment chooses to advance time by some concrete delay $d_i \in \mathbb{R}_+$, making sure that d_i does not violate the deadline provided by A. Finally, the environment notifies A that it advanced time by d_i and A updates its state to s_{i+1} accordingly, using its T function. The new time is $t + d_i$ and execution repeats from then on in the same fashion.

It is worth noting that in our model of actors and behaviors, the "interesting points in time" are determined by the environment, and not the actor. In fact, the actor has no explicit notion of time (although it can measure time by using state variables, e.g., timers). However, the actor can impose constraints on the advancement of time using deadlines.

The fact that F, P, D, T are functions makes our actors deterministic. This is done for reasons of simplicity, and because our main focus is heterogeneity. Non-deterministic actors could be modeled, however, as deterministic actors using extra input variables.

4.4. *Actor Classification and Special Cases*

Consider an actor $A = (I, O, S, s_0, F, P, D, T)$.

— A is called a *source* if it has no input variables, i.e., $I = \emptyset$.
— A is called a *sink* if it has no output variables, i.e., $O = \emptyset$.
— A is called *stateless* if it has no state variables, i.e., $S = \emptyset$.
— A is called *untimed* if D is a constant function that always returns infinity, i.e., for any state s and input x, $D(s, x) = \infty$. Otherwise A is called *timed*.
— A is called *delay-independent* if T leaves the state unchanged, i.e., for any state s, input x, and delay d, $T(s, x, d) = s$. Otherwise A is called *delay-dependent*.
— A is called a *dataflow* actor if its input and output variables range over streams.

When a set of variables is empty or a function is independent of one or more of its parameters, the type of the function simplifies, and we accordingly simplify our notation. For instance, if A is a source then $\widehat{I}$ is a singleton, therefore, F does not depend on the input. Because of this, we can assume that F is a function with a simpler type $F : \widehat{S} \to \widehat{O}$ and we can write $F(s)$, for state s. We similarly simplify notation for other special cases of actors. For example, for a stateless actor, we write $F(x)$, for input x. For an actor that is both stateless and a source, we write $F()$. Also note that for a stateless actor functions P and T are trivial: they are the constant functions that return the unique element $*$, therefore they need not be specified.

4.5. *Examples of Atomic Actors*

4.5.1. *Constant:* The Constant actor parameterized by some value $\alpha \in \mathcal{U}$ is an actor that intuitively produces the constant value α every time it is fired. This actor can be modeled as follows:

$$\mathsf{Const}_\alpha = (\emptyset, \{o\}, \emptyset, *, F, P, D, T) \tag{11}$$

where

$$F() = \{o \mapsto \alpha\} \text{ and } D() = \infty \tag{12}$$

F returns the valuation $y \in \widehat{\{o\}}$ which assigns α to the unique output variable o. Const_α is a source, and it is also stateless, untimed, and delay-independent.

4.5.2. *Identity:* The Identity actor simply "copies" its input to its output. This actor can be modeled as follows:

$$\mathsf{Id} = (\{v\}, \{o\}, \emptyset, *, F, P, D, T) \tag{13}$$

where

$$F(x) = \{o \mapsto x(v)\} \text{ and } D(x) = \infty \tag{14}$$

for any input x.

F returns the valuation $y \in \widehat{\{o\}}$ which assigns to the unique output variable o the value $x(v)$ of the unique input variable v at the input valuation x. Id is a stateless, untimed, and delay-independent actor.

4.5.3. *Adder:* The Adder actor intuitively produces a value that corresponds to the sum of its inputs every time it is fired. This actor can be modeled as follows:

$$\mathsf{Add} = (\{v_1, v_2\}, \{o\}, \emptyset, *, F, P, D, T) \tag{15}$$

where

$$F(x) = \begin{cases} \{o \mapsto (x(v_1) \oplus x(v_2))\} & \text{if } x(v_1) \neq \bot \text{ and } x(v_2) \neq \bot \\ \{o \mapsto \bot\} & \text{otherwise} \end{cases} \tag{16}$$

$$u_1 \oplus u_2 = \begin{cases} u_1 + u_2 & \text{if } u_1 \neq \mathsf{absent} \text{ and } u_2 \neq \mathsf{absent} \\ u_1 & \text{if } u_1 \neq \mathsf{absent} \text{ and } u_2 = \mathsf{absent} \\ u_2 & \text{if } u_1 = \mathsf{absent} \text{ and } u_2 \neq \mathsf{absent} \\ \mathsf{absent} & \text{if } u_1 = \mathsf{absent} \text{ and } u_2 = \mathsf{absent} \end{cases} \tag{17}$$

$$D(x) = \infty \tag{18}$$

for any input x. That is, F returns the valuation $y \in \widehat{\{o\}}$ which assigns to the unique output variable o the sum $x(v_1) + x(v_2)$ of the values of the two input variables v_1 and v_2, provided none of these values are unknown (i.e., $\bot$), or absent. If any of the input values is unknown, the output is unknown. If one of the inputs is absent, the output is equal to the other input. If both inputs are absent, the output is absent.

Add is stateless, untimed and delay-independent.

4.5.4. *Logical-and:* The And actor intuitively produces a value that corresponds to the logical-and (conjunction) of its inputs every time it is fired. This actor can be modeled as follows:

$$\mathsf{And} = (\{v_1, v_2\}, \{o\}, \emptyset, *, F, P, D, T) \tag{19}$$

where

$$\begin{aligned} F(x) &= \begin{cases} \{o \mapsto \mathit{false}\} & \text{if } x(v_1) = \mathit{false} \text{ or } x(v_2) = \mathit{false} \\ \{o \mapsto \mathit{true}\} & \text{if } x(v_1) = \mathit{true} \text{ and } x(v_2) = \mathit{true}, \text{ otherwise:} \\ \{o \mapsto x(v_1)\} & \text{if } x(v_2) = \mathsf{absent} \\ \{o \mapsto x(v_2)\} & \text{if } x(v_1) = \mathsf{absent} \\ \{o \mapsto \bot\} & \text{otherwise} \end{cases} & (20) \\ D(x) &= \infty & (21) \end{aligned}$$

for any input x.

Note that And is *non-strict* in the sense that it returns *false* if one of its inputs is *false*, even if the other input is unknown or absent. This can be useful to “break” input-output dependencies in feedback loops, as in the model shown in Section 6.1. On the other hand, if one input is *true* and the other input is unknown, the result is unknown. Finally, if one input is *true* and the other is absent, the result is *true*.

And is stateless, untimed and delay-independent.

4.5.5. *Memory:* The Memory actor[13] parameterized by some initial value $\alpha \in \mathcal{U}$ intuitively stores a value in a slot of memory. It returns the value as an output every time it fires and updates it every time it postfires. This actor can be modeled as follows:

$$\mathsf{Mem}_\alpha = (\{v\}, \{o\}, \{m\}, s_0, F, P, D, T) \tag{22}$$

where

$$\begin{aligned} s_0 &= \{m \mapsto \alpha\} & (23) \\ F(s,x) &= \{o \mapsto s(m)\} & (24) \\ P(s,x) &= \{m \mapsto x(v)\} & (25) \\ D(s,x) &= \infty & (26) \\ T(s,x,d) &= s & (27) \end{aligned}$$

for any state s, input x and delay $d \in \mathbb{R}_+$. Mem_α is untimed and delay-independent.

4.5.6. *Discrete Clock:* The Discrete Clock actor parameterized by some value $\alpha \in \mathcal{U}$ and some period $\pi \in \mathbb{R}_+$ intuitively produces an event with value α every π time units. This actor can be modeled as follows:

$$\mathsf{Clk}_{\alpha,\pi} = (\emptyset, \{o\}, \{c\}, s_0, F, P, D, T) \tag{28}$$

[13] This actor is called `Sample Delay` in Ptolemy. We prefer to use the term `Memory` to distinguish it from the `Constant Delay` actor.

where

$$
\begin{aligned}
s_0 &= \{c \mapsto 0\} && (29)\\
F(s) &= \begin{cases} \{o \mapsto \alpha\} & \text{if } s(c) = 0 \\ \{o \mapsto \mathsf{absent}\} & \text{otherwise} \end{cases} && (30)\\
P(s) &= \begin{cases} \{c \mapsto \pi\} & \text{if } s(c) = 0 \\ \{c \mapsto s(c)\} & \text{otherwise} \end{cases} && (31)\\
D(s) &= s(c) && (32)\\
T(s,d) &= \{c \mapsto \big(s(c) - d\big)\} && (33)
\end{aligned}
$$

for any state s, and delay $d \in \mathbb{R}_+$ such that $d \leq D(s)$.

$\mathsf{Clk}_{\alpha,\pi}$ has a single state variable c, which is a timer. When the actor is fired, it produces an output event with value α if its timer has *expired*, that is, has reached the value 0. Otherwise, it produces an output with the special value absent representing the fact that the output is absent at this point in time. The state update of $\mathsf{Clk}_{\alpha,\pi}$ works as follows. When the timer c reaches the value 0 it is reset to π, so that it counts a new period. If c is not yet 0, then its value is left unchanged, as denoted by the mapping $c \mapsto s(c)$. In the above formalization, the timer is initialized to 0, which means it is initially expired. An alternative could be to initialize the timer to π which would imply that the timer does not produce a value until the first π time units have elapsed.

$\mathsf{Clk}_{\alpha,\pi}$ is timed: its deadline function D returns $s(c)$, the current value of c. This imposes constraints on the environment which calls the functions of $\mathsf{Clk}_{\alpha,\pi}$ on the times at which these functions may be called (Section 4.3). $\mathsf{Clk}_{\alpha,\pi}$ is delay-dependent: its time-update function T decrements the timer by the amount of time d that the environment chooses to let elapse. Notice that $d \leq D(s)$ implies $d \leq s(c)$, therefore, the new value of the timer, $s(c) - d$, is guaranteed to be non-negative. Also note that the condition $d \leq D(s)$ is ensured by the rules defining actor behaviors (Section 4.3).

4.5.7. *Constant Delay:* The Constant Delay actor parameterized by a delay $\Delta \in \mathbb{R}_+$ intuitively delays each input event by Δ time units. This actor can be modeled as follows:

$$\mathsf{Del}_\Delta = (\{v\}, \{o\}, \{\mathsf{Active}\}, s_0, F, P, D, T) \tag{34}$$

where Active is a state variable: Active is a FIFO (first-in first-out) queue of tuples of the form $(\alpha, d) \in \mathcal{U} \times [0, \Delta]$. Tuple (α, d) represents the fact that the actor must "remember" to produce an event with value α in d time units. Let $[]$ denote the empty queue, $\mathsf{head}(q)$ denote the head of a queue q, and $\mathsf{tail}(q)$ its tail. Let $q \cdot e$ denote the queue obtained by appending element e at the end of q. Finally, for $d' \in \mathbb{R}_+$, let $q \ominus d'$ denote the queue obtained by replacing each element

(α, d) of q by $(\alpha, d - d')$. Then, we have:

$$\begin{aligned}
s_0 &= \{\mathsf{Active} \mapsto []\} && (35)\\
F(s,x) &= \begin{cases} \{o \mapsto \alpha\} & \text{if } s(\mathsf{Active}) \neq [] \text{ and } \mathsf{head}(s(\mathsf{Active})) = (\alpha, 0) \\ \{o \mapsto \mathsf{absent}\} & \text{otherwise} \end{cases} && (36)\\
P(s,x) &= \begin{cases} \{\mathsf{Active} \mapsto \big(A' \cdot (\alpha, \Delta)\big)\} & \text{if } x(v) = \alpha \text{ and } \alpha \neq \mathsf{absent} \\ \{\mathsf{Active} \mapsto A'\} & \text{otherwise} \end{cases} && (37)\\
A' &= \begin{cases} \mathsf{tail}(s(\mathsf{Active})) & \text{if } s(\mathsf{Active}) \neq [] \text{ and } \mathsf{head}(s(\mathsf{Active})) = (\alpha, 0) \\ s(\mathsf{Active}) & \text{otherwise} \end{cases} && (38)\\
D(s,x) &= \begin{cases} d & \text{if } s(\mathsf{Active}) \neq [] \text{ and } \mathsf{head}(s(\mathsf{Active})) = (\alpha, d) \\ \infty & \text{otherwise} \end{cases} && (39)\\
T(s,x,d) &= \{\mathsf{Active} \mapsto \big(s(\mathsf{Active}) \ominus d\big)\} && (40)
\end{aligned}$$

for any state s, input x, and delay $d \in \mathbb{R}_+$ such that $d \leq D(s,x)$.

The intuition is as follows: s_0 initializes Active to the empty queue. F produces an event with value α at the output if the head of the queue reads $(\alpha, 0)$, which means it is time to produce such an event, otherwise the event with absent value is produced. P updates the queue by first removing its head in the case where a non-absent event is produced, and then appending a new element (α, Δ) at the end of the queue, if a non-absent input is received. A' is an intermediate variable denoting the queue obtained after potentially removing the head of Active.

The deadline function D returns the delay of the head of Active if the queue is non-empty, and ∞ otherwise. Note that the fact that the delay Δ is constant ensures that elements in Active are always ordered with respect to their delay field. This ensures that the head of the queue has the smallest delay. The time-update function T decrements all delays in the queue by the delay d' chosen by the environment.

Del_Δ is timed and delay-dependent.

4.5.8. *Sinusoid:* The sinusoid actor generates a continuous sinusoidal signal parameterized by frequency $\omega \in \mathbb{R}$, amplitude $\alpha \in \mathbb{R}$, and phase offset $\phi \in \mathbb{R}$. The sinusoid actor has a single state variable r representing the current phase of sinusoid generated, which is updated by the time-update function. The signal generated follows the function $\alpha \sin(\omega r + \phi)$.

$$\mathsf{Sin}_{\omega,\alpha,\phi} = (\emptyset, \{o\}, \{r\}, s_0, F, P, D, T) \tag{41}$$

where

$$\begin{aligned}
s_0 &= \{r \mapsto 0\} && (42)\\
F(s) &= \{o \mapsto \alpha \cdot \sin(\omega \cdot s(r) + \phi)\} && (43)\\
P(s) &= s && (44)\\
D(s) &= \infty && (45)\\
T(s,d) &= \{r \mapsto s(r) + d\} && (46)
\end{aligned}$$

for any state s and delay $d \in \mathbb{R}_+$.

$\mathsf{Sin}_{\omega,\alpha,\phi}$ is a source, untimed but delay-dependent actor.

4.5.9. *Integrator:* The Integrator actor $\mathsf{Integrator}_\alpha$, parameterized by initial value $\alpha \in \mathbb{R}_+$, is used by the CT director (Section 6.3) to perform integration. $\mathsf{Integrator}_\alpha$ is in fact identical to Mem_α. However, we use a special name as a syntactic mechanism that permits the CT director to identify the integrator actors in a given model.

It may appear surprising that Integrators are nothing else but memories, but this is the case for numerical solvers of Runge-Kutta type (see Section 6.3). Integrators would be less trivial under a different solver, for instance, a fixed-step Euler solver. In that case, Integrators can be defined to perform the integration, at the same time eliminating the need for a special CT director: in the case of Euler integration, the DE director is sufficient.

4.5.10. *Upsampling and Downsampling:* Ptolemy's library of atomic actors contains a number of actors that can be called *dataflow* actors, in the sense that they operate on *streams* of values. A stream is a finite or infinite sequence of values. The elements of a stream are called *tokens*. Dataflow actors are useful in domains such as PN (Process Networks) where communication happens using FIFO (first-in, first-out) queues of unbounded size. We illustrate how data actors can be modeled in our framework by considering first two *static* dataflow (also called synchronous dataflow or SDF) actors (Lee and Messerschmitt, 1987).

The Upsampling actor parameterized by a constant $k \in \mathbb{N}$ copies each token it receives at its input n times at its output. This actor can be modeled as follows:

$$\mathsf{Up}_n = (\{p\}, \{q\}, \emptyset, *, F, P, D, T) \tag{47}$$

where the values of variables p, q are streams.

If α is a value, α^n denotes the stream of length n consisting of n consecutive copies of α. Then, the F and D functions of Up_n can be defined as follows:[14]

$$F(x) = \begin{cases} \{q \mapsto \mathsf{head}(x(p))^n \cdot F(x')\} & \text{if } x(p) \neq [],\ \text{where } x'(p) = \mathsf{tail}(x(p)) \\ \{q \mapsto []\} & \text{otherwise} \end{cases} \tag{48}$$

$$D(x) = \infty \tag{49}$$

for any input x. Up_n is stateless, untimed and delay-independent.

We can similarly define the Downsampling actor parameterized by a constant $n \in \mathbb{N}$, which repeatedly consumes a stream of n consecutive tokens at its input and returns the first of these tokens at its output. This actor can be modeled as follows:

$$\mathsf{Down}_n = (\{p\}, \{q\}, \emptyset, *, F, P, D, T) \tag{50}$$

Let $|\rho|$ denote the length of a stream ρ. If ρ is infinite then $|\rho| = \infty$. If $|\rho| \geq k$ then let $\rho(k..)$ denote the substring obtained from ρ by removing the first k elements of ρ. Then:

$$F(x) = \begin{cases} \{q \mapsto \mathsf{head}(x(p)) \cdot F(x')\} & \text{if } |x(p)| \geq n,\ \text{where } x'(p) = (x(p))(n..) \\ \{q \mapsto []\} & \text{otherwise} \end{cases} \tag{51}$$

$$D(x) = \infty \tag{52}$$

for any input x. Down_n is stateless, untimed and delay-independent.

[14] we use a recursive definition for F as it is easier to express, and can also be directly mapped into an implementation in a functional programming language such as Haskell

4.5.11. *Switch and Select:* Up_n and Down_n are SDF actors. We now describe Switch and Select which are *dynamic* dataflow actors, in the sense that the number of tokens the consume or produce is not fixed and can vary from one firing to the next (Buck, 1993).

The Switch actor has a data input queue, a control input queue, and two data output queues. The control input queue carries boolean tokens. Switch uses each control token to choose in which of the two output queues to "route" the token it receives from the (single) input queue. This actor can be modeled as follows:

$$\mathsf{Switch} = (\{p, p_c\}, \{q_1, q_2\}, \emptyset, *, F, P, D, T) \tag{53}$$

where:

$$\begin{aligned} F(x) &= \begin{cases} \{(q_1,q_2) \mapsto \big(\mathsf{head}(x(p)), []\big) \cdot F(x')\}, \\ \ \text{if } x(p) \neq [] \text{ and } x(p_c) \neq [] \text{ and } \mathsf{head}(x(p_c)) = \mathit{true}, \\ \quad \text{where } x'(p) = \mathsf{tail}(x(p)) \text{ and } x'(p_c) = \mathsf{tail}(x(p_c)) \\ \{(q_1,q_2) \mapsto \big([], \mathsf{head}(x(p))\big) \cdot F(x')\}, \\ \ \text{if } x(p) \neq [] \text{ and } x(p_c) \neq [] \text{ and } \mathsf{head}(x(p_c)) = \mathit{false}, \\ \quad \text{where } x'(p) = \mathsf{tail}(x(p)) \text{ and } x'(p_c) = \mathsf{tail}(x(p_c)) \\ \{(q_1,q_2) \mapsto ([], [])\}, \qquad\qquad \text{otherwise} \end{cases} & (54) \\ D(x) &= \infty & (55) \end{aligned}$$

for any input x.

The Select actor is in some sense the "dual" of Switch. Select has two data input queues, a control input queue and a data output queue. It uses each control token to select which of the two input queues to read from, and copies the corresponding input token to its output queue. This actor can be modeled as follows:

$$\mathsf{Select} = (\{p_1, p_2, p_c\}, \{q\}, \emptyset, *, F, P, D, T) \tag{56}$$

where:

$$F(x) = \begin{cases} \{q \mapsto \mathsf{head}(x(p_1)) \cdot F(x')\} & \text{if } x(p_c) \neq [] \text{ and } \mathsf{head}(x(p_c)) = \mathit{true} \text{ and } x(p_1) \neq [], \\ & \quad \text{where } x'(p_c) = \mathsf{tail}(x(p_c)), x'(p_1) = \mathsf{tail}(x(p_1)), x'(p_2) = x(p_2), \\ \{q \mapsto \mathsf{head}(x(p_2)) \cdot F(x')\} & \text{if } x(p_c) \neq [] \text{ and } \mathsf{head}(x(p_c)) = \mathit{false} \text{ and } x(p_2) \neq [], \\ & \quad \text{where } x'(p_c) = \mathsf{tail}(x(p_c)), x'(p_2) = \mathsf{tail}(x(p_2)), x'(p_1) = x(p_1), \\ \{q \mapsto []\} & \text{otherwise} \end{cases}$$
$$D(x) = \infty$$

for any input x.

The output variables n, k_c, k_1 and k_2 of Switch and Select capture the number of tokens consumed at a given firing from each of the input queues of these actors.

Both Switch and Select are stateless, untimed and delay-independent actors.

4.5.12. *Extended State Machines:* User-defined *extended state machines* (ESMs) can be naturally modeled as actors. ESMs are useful to capture various types of behaviors, and they are also part of the syntax used to describe modal models: a modal model is an ESM whose locations are *refined* into other composite actors (see Section 5.2).

An ESM naturally defines an actor $A = (I, O, S, s_0, F, P, D, T)$ as follows. I is the set of input ports and O the set of output ports that the ESM may use. S includes all state variables (i.e., dynamic parameters) of the ESM, plus a variable ranging over the set of locations of the ESM. s_0 initializes the parameters to their default value provided by the user, and the location variable to the initial location, also specified by the user. To compute F and P, an outgoing transition from the current location specified by s is chosen[15] such that the transition is *enabled*, that is, its guard evaluates to $true$, on s and x. F and P are then defined by the output and set actions, respectively, of the chosen transition. Output variables that are not mentioned in the output action are implicitly set to absent, while state variables not mentioned in the set action are implicitly left unchanged. If no transition is enabled, this corresponds to a self-loop transition which leaves the state unmodified and sets all outputs to absent. D always returns ∞ and T leaves the state unchanged. Thus, A is untimed and delay-independent.

We do not formalize the above intuitive semantics of ESMs, as they are standard in the literature.

5. Actor Diagrams

As mentioned earlier, actors in Ptolemy can be connected to form *diagrams*, which can in turn be encapsulated into composite actors. For most composite actors, actor diagrams follow a block-diagram notation. That is, they are formed by instantiating actors and connecting output ports of an actor into input ports of another actor. The top-level diagram of the model of Figure 3 follows this block-diagram notation. Another example is the model of Figure 5. A different type of actor diagram is used in the case of modal models, where the composite actor is defined by an ESM whose locations contain other diagrams. An example is the `ModalModel` actor shown in Figure 3. In this section we show how these two types of actor diagrams are formalized in our framework.

5.1. *Block Diagrams*

We will formalize the structure of the first type of composite actors, namely, block-diagram composite actors, as a set of actors where variables that have the same name are implicitly connected. More precisely, we define a *block diagram* to be a set of actors $H = \{A_1, ..., A_n\}$, with $A_i = (I_i, O_i, S_i, s_{0,i}, F_i, P_i, D_i, T_i)$, such that

— All I_i are pair-wise disjoint, i.e., for $i, j \in \{1, ..., n\}$, if $i \neq j$ then $I_i \cap I_j = \emptyset$.
— All O_i are pair-wise disjoint.
— All S_i are pair-wise disjoint.

When multiple actors are instances of the same actor, the above disjointness requirements are achieved by appropriate renaming. For example, if H contains two Const actors, say, A_3 and A_5, their output variables are renamed o_3 and o_5. On the other hand, we allow two sets I_i and O_j not to be disjoint, so that if $v \in I_i \cap O_j$ then this means that the output variable v of actor A_j is connected to the input variable with the same name of actor A_i.

[15] In Ptolemy, if more than one outgoing transitions are enabled then one is chosen non-deterministically. As stated earlier we consider here only deterministic actors for simplicity. This results in no loss of expressiveness as non-deterministic actors can be modeled as deterministic actors using extra input variables.

Fan-out, where an output variable o of some actor is connected to the input variables $v_1, v_2, ...$ of multiple actors, is modeled by adding an explicit actor Fanout with input o and outputs $v_1, v_2, ...$ Fanout is a stateless actor that "copies" its input to all its outputs every time it fires.

Feedback loops can be formed by connecting an output variable of an actor to one of its input variables. However, since in order to avoid confusion we assume that input and output variables are disjoint, we model feedback by including an additional Id actor in the connection.

5.2. *Modal Model Diagrams*

The structure of the second type of composite actors, namely, modal-models, is formalized as a *modal-model diagram*, namely, a set of actors $M = \{A_c, A_1, ..., A_n\}$, with $A_c = (I_c, O_c, S_c, s_{0,c}, F_c, P_c, D_c, T_c)$, and $A_i = (I_i, O_i, S_i, s_{0,i}, F_i, P_i, D_i, T_i)$, such that

— A_c is an ESM actor, called the *controller* of the modal model. A_c must have exactly n locations denoted $\ell_1, ..., \ell_n$. Location ℓ_i of A_c is refined into actor A_i.
— All actors in M have the same sets of input and output variables, that is, $I_c = I_1 = \cdots = I_n$ and $O_c = O_1 = \cdots = O_n$.
— All S_i are pair-wise disjoint.

In Ptolemy it is possible that some locations have no refinement. We model this as follows. Let ℓ_i be such a location. We then define A_i to be the stateless, untimed and delay-independent actor with input variables I_c and output variables O_c such that F_i assigns all output variables to absent.

6. Directors

In Ptolemy, models of computation (also called *domains*) are implemented by *directors*. Directors "tell actors what to do". In particular, they choose when to call the different functions of the actor interface and they also manage the data exchanges between the actors. Therefore, directors implement the model of concurrency and communication.

We formalize directors as *composition operators*. Specifically, a director takes as input an actor diagram (which can be either a block-diagram or a modal-model diagram) and returns an actor A as output. A is a composite actor, but has the same interface as an atomic actor, therefore, can be used in further compositions.

In the rest of the paper we define the directors for various domains of Ptolemy.

6.1. *Synchronous-Reactive (SR)*

The SR domain covers a broad class of models that use the synchronous model of computation. The latter is suitable for modeling a wide variety of systems, from digital circuits to embedded control software.

Let $H = \{A_1, ..., A_n\}$ be a block diagram, with $A_i = (I_i, O_i, S_i, s_{0,i}, F_i, P_i, D_i, T_i)$, for $i = 1, ..., n$. H can be composed using the SR director provided *every actor A_i is untimed and delay-independent*. Then, the *SR composite actor* is defined to be the actor

$$\mathsf{SR}(H) = (I, O, S, s_0, F, P, D, T)$$

where:

$$V = \bigcup_{i=1}^{n} O_i \tag{57}$$

$$I = \bigcup_{i=1}^{n} I_i \setminus V \tag{58}$$

$$O = V \setminus \bigcup_{i=1}^{n} I_i \tag{59}$$

$$S = \bigcup_{i=1}^{n} S_i \tag{60}$$

$$s_0 = (s_{0,1}, ..., s_{0,n}) \tag{61}$$

V is intermediate notation for the set of all output variables of all actors in H. The set of input variables I of the composite actor $\mathsf{SR}(H)$ is the set of all input variables of actors in H that are not connected to an output variable, i.e., that are not in V. The set of output variables O of $\mathsf{SR}(H)$ is defined to be the set of all output variables V minus those variables that are connected to an input variable.[16]

We proceed to define the functions F, P, D, T of $\mathsf{SR}(H)$. Let $s = (s_1, ..., s_n) \in \widehat{S}$ be a state and $x \in \widehat{I}$ be an input of $\mathsf{SR}(H)$, where $s_i \in \widehat{S_i}$, for $i = 1, ..., n$. We define the function

$$\widetilde{F}_{s,x} : \widehat{V} \to \widehat{V} \tag{62}$$

such that, for $y \in \widehat{V}$ and $o \in O_j$, for some $j \in \{1, ..., n\}$, we have

$$\big(\widetilde{F}_{s,x}(y)\big)(o) = F_j\big(s_j, (x,y) \upharpoonright_{I_j}\big)(o) \tag{63}$$

Recall that $(x,y) \upharpoonright_{I_j}$ denotes the restriction of (x,y) to variables in I_j and (x,y) is the combined valuation composed of x and y. The above definition states that, to compute the value of a given output variable o of A_j, function $\widetilde{F}_{s,x}$ uses the fire function of A_j, i.e., F_j. This function takes as inputs the local state s_j of A_j, and the local input of A_j, which is precisely $(x,y) \upharpoonright_{I_j}$.

The semantics of $\mathsf{SR}(H)$ is defined provided $\widetilde{F}_{s,x}$ is a *continuous function over a certain CPO*, for any s and x. In Ptolemy this is usually ensured as follows: First, the fire function of every atomic actor is designed so that it is monotonic over the "flat" CPO that consists of the minimal element $\perp$ and all other elements in $\mathcal{U}$ being greater than $\perp$ in the CPO order: this CPO is illustrated in Figure 4. Second, the fact that monotonicity implies continuity for flat CPOs. Third, the fact that Cartesian products of CPOs are CPOs, and composing corresponding continuous functions over such CPOs yields a continuous function over the product CPO (Davey and Priestley, 2002).

A continuous function f over a CPO has a unique *least fixpoint*, that is, a unique x^* such that $f(x^*) = x^*$ and for any other fixpoint x of f, $x^* \leq x$ with respect to the order $\leq$ of the CPO. Let

[16] We could also define O to be equal to V. We opt not to do so, as we should then also introduce a *hiding* operator that removes some unnecessary outputs. The definition we use results in this hiding happening by default to all connected outputs. This results in no loss of generality, as we can always add explicit actors that copy the outputs that we do not wish to hide.

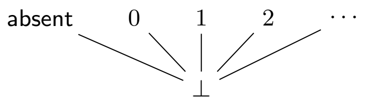


Fig. 4. The flat CPO used to ensure existence of a unique least fixpoint that defines the semantics of SR, DE and CT.

$y^*_{s,x}$ be the unique least fixpoint of $\widetilde{F}_{s,x}$. Note that $y^*_{s,x}$ is a valuation over V, that is, it assigns a value to every output variable of each actor A_i of H. We then define:

$$F(s,x) \quad = \quad y^*_{s,x} \upharpoonright_O \tag{64}$$

$$P(s,x) \quad = \quad \Big(P_1\big(s_1,(x,y^*_{s,x}) \upharpoonright_{I_1}\big),...,P_n\big(s_n,(x,y^*_{s,x}) \upharpoonright_{I_n}\big)\Big) \tag{65}$$

Note that in the case of flat CPOs the fixpoint $y^*_{s,x}$ can be computed effectively in a finite number of iterations. Indeed, $y^*_{s,x}$ is equal to the limit $\lim_{n\to\infty} \widetilde{F}^n_{s,x}(\bot)$, where $\widetilde{F}^0_{s,x}(\bot) = \bot$ and $\widetilde{F}^{n+1}_{s,x}(a) = \widetilde{F}_{s,x}(\widetilde{F}^n_{s,x}(a))$ for all a. In the case of a flat CPO this limit can be reached in a finite number of iterations because the CPO has a finite height and therefore a variable can only change value at most one time: from $\bot$ to some value other than $\bot$. Since the number of variables is finite, the total number of possible changes is also finite. More "clever" methods to compute the fixpoint have been studied, for instance, in (Edwards and Lee, 2003). In the special case where the block diagram H is acyclic (i.e., connections do not form cycles, or these cycles are "broken" by special actors such as Mem whose outputs do not depend on the current inputs) the fixpoint can be computed in a single iteration, by firing all actors in H just once, in topological order according to the diagram dependencies.

Note that the fixpoint $y^*_{s,x}$ may contain absent or unknown values, that is, there may exist variables to which $y^*_{s,x}$ assigns either absent or $\bot$. As mentioned earlier, absent is a perfectly legal value, just like other values (booleans, integers, etc.) but especially useful in defining actors in the DE domain. On the other hand, a fixpoint that contains $\bot$ values would normally indicate an erroneous model, whose semantics are ambiguous. An example will be discussed below.

It remains to define functions D and T. There are different options in defining D. One option is to treat $\mathsf{SR}(H)$ as an untimed actor, in which case

$$D(s,x) \quad = \quad \infty \qquad \text{for any state } s \text{ and input } x. \tag{66}$$

Another option is to treat $\mathsf{SR}(H)$ as a timed actor and have D return a constant value $h \in \mathbb{R}_+$ provided by the user as a parameter:

$$D(s,x) \quad = \quad h \qquad \text{for any state } s \text{ and input } x. \tag{67}$$

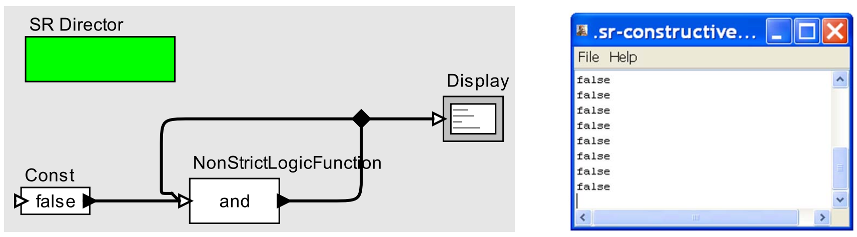
Both options are available in Ptolemy, and the user can select between the two by configuring the appropriate parameter of the SR director.

In both cases, T is defined so as to leave the state unchanged:

$$T(s,x,d) \quad = \quad s \qquad \text{for any state } s\text{, input } x \text{ and delay } d. \tag{68}$$

Note that $\mathsf{SR}(H)$ is by definition delay-independent.

A simple Ptolemy model that uses the SR director is shown in Figure 5. This model contains



```
false
false
false
false
false
false
false
false
```

Fig. 5. An SR model (left) and its output (right).

four actors: Const$_{false}$, And, `Display` and Fanout. The Fanout actor is denoted by the small black rhombus. The Display actor merely serves for outputting the results of the Ptolemy simulation on the screen, and does not influence the behavior of the rest of the model. Ignoring the Display actor, and applying the SR director to the remaining block diagram, we obtain a composite actor with no input variables and a single output variable (one of the outputs of Fan-out). The composite actor is a stateless, untimed and delay-independent actor that outputs $false$ every time it is fired. Indeed, during the computation of the fixpoint y^*, the non-strictness of the And actor (Equation 20) results in its output being computed as $false$ despite the fact that one of its inputs is $\perp$.

To see how some models may be erroneous, consider a slight modification of this model, where the Const$_{false}$ actor is replaced by a Const$_{true}$ actor producing a signal with value $true$ instead of $false$. In that case, the fixpoint results in the output being $\perp$, since the logical and of $true$ and $\perp$ is $\perp$ (Equation 20). Indeed this is an ambiguous model that could be seen as admitting both $true$ and $false$ as possible solutions. Instead, our semantics declares the output to be unknown, as is done in the case of constructive semantics of synchronous languages like Esterel.

6.2. *Discrete Event (DE)*

In SR the notion of time is not quantitative in the sense that synchronous reactions are simply ordered as a sequence, but do not necessarily correspond to any "real" notion of time (e.g., in an implementation, the time that elapses between successive reactions may vary dynamically). The DE domain aims at capturing systems where a quantitative notion of time is important. Time in DE models is continuous (the real numbers) but the dynamics of the systems that operate within this time frame are discrete. We therefore speak of discrete events. DE therefore covers a broad class of timed systems that follow this discrete behavior, including real-time control systems.

Let $H = \{A_1, ..., A_n\}$ be a block diagram, with $A_i = (I_i, O_i, S_i, s_{0,i}, F_i, P_i, D_i, T_i)$, for $i = 1, ..., n$. The *DE composite actor* is defined to be the actor

$$\mathsf{DE}(H) = (I, O, S, s_0, F, P, D, T)$$

where I, O, S, s_0, F, P are defined as in the case of $\mathsf{SR}(H)$, and D, T are defined as follows:

$$D(s,x) = \min\{D_i\big(s_i, (x, y^*_{s,x}) \restriction_{I_i}\big) \mid i = 1, ..., n\} \tag{69}$$

$$T(s,x,d) = \Big(T_1\big(s_1, (x, y^*_{s,x}) \restriction_{I_1}, d\big), ..., T_n\big(s_n, (x, y^*_{s,x}) \restriction_{I_n}, d\big)\Big) \tag{70}$$

for any state s, input x and delay $d \in \mathbb{R}_+$.

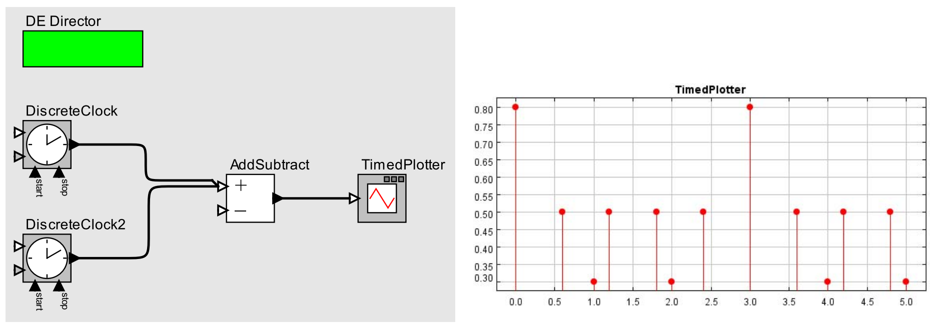


Fig. 6. A DE model (left) and its output (right).

A simple Ptolemy model that uses the DE director is shown in Figure 6. This model contains four actors: $\mathsf{Clk}_{0.5,0.6}$, $\mathsf{Clk}_{0.3,1}$, Add, and `TimedPlotter`.[17] The TimedPlotter actor serves for displaying the output of the model resulting from simulation on the screen, and can be ignored when defining the composite actor. Recall that the first parameter of the Clk actor is the output value and the second is the period. Therefore, $\mathsf{Clk}_{0.5,0.6}$ denotes a clock that fires every 0.6 time units and outputs the value 0.5 every time it fires. Similarly, $\mathsf{Clk}_{0.3,1}$ fires every 1 time unit and outputs 0.3. Ignoring the TimedPlotter actor and applying the DE director to the remaining block diagram, we obtain a composite actor with no input variables and a single output variable. The results of firing this composite actor until time 5.0 are shown to the right of Figure 6. Notice that there are instants when both clock actors produce an event (at times 0.0 and 3.0) in which case the Add actor produces an output corresponding to the sum of the two values. At other times (e.g., 0.6, 1, 1.2, and so on) only one clock produces an event, which the Add actor reproduces at its output.

6.3. *Continuous Time (CT)*

The CT domain is used for modeling and simulating the broad class of continuous-time systems. This includes systems modeled by differential equations, with applications in control, mechanics, biology, and other types of dynamical systems.

Let $H = \{A_1, ..., A_n\}$ be a block diagram, with $A_i = (I_i, O_i, S_i, s_{0,i}, F_i, P_i, D_i, T_i)$, for $i = 1, ..., n$. H may contain integrator actors as defined in Section 4.5.9. We require that H satisfies the following property: for any actor A_i that has an output which is connected to the input of an integrator, A_i must produce only numerical values. In particular, it must never produce absent on that output. This requirement aims to characterize a “continuous” signal in the essentially discrete computation framework which underlies Ptolemy (as well as most computers). This assumption is used by numerical integration algorithms which implicitly assume that a signal always has a numerical value that can be “polled”.

Then, the *CT composite actor* with parameters *initStepSize*, *maxStepSize* and *errorTolerance*

[17] Clock actors in Ptolemy have inputs that we will ignore for the purposes of this paper.

is defined to be the actor

$$\mathsf{CT}(H, \text{initStepSize}, \text{maxStepSize}, \text{errorTolerance}) = (I, O, S, s_0, F, P, D, T)$$

where I, O are defined as in the case of $\mathsf{SR}(H)$. S and s_0 are defined as follows:

$$S = \{\text{stepSize}\} \cup \bigcup_{i=1}^{n} S_i \tag{71}$$

$$s_0 = \{\text{stepSize} \mapsto \text{initStepSize}\} \cup \bigcup_{i=1}^{n} s_{0,i} \tag{72}$$

that is, the composite CT actor uses an additional state variable called *stepSize*, initialized to *initStepSize*.

Functions F and P of the composite CT actor do not use nor modify the state variable *stepSize* and are otherwise defined as in the case of $\mathsf{SR}(H)$.

The deadline and time-update functions D and T are described in pseudo-code in Procedures 1 and 2. Together with Procedure 3 the three procedures implement a Runge-Kutta 2(3) ODE solving method. Procedure *Runge-Kutta23* performs an integration step, given a state and an input valuation for the composite actor as well as a step size for the integration. The procedure returns the set of local truncation errors (LTEs) and the values at the inputs for each of the integrators after the integration step.

The following notation is used in the procedures: Ix denotes the set of indices of integrator actors in the block diagram H, i.e., $\mathsf{Ix} = \{i \mid A_i \text{ is an integrator}\}$. $I_\int$ denotes the set of all input variables of all integrators in H, i.e., $I_\int = \bigcup_{i \in \mathsf{Ix}} I_i$. The state variable of an integrator actor A_i is denoted m_i and its input variable is denoted v_i. Notation DE.D and DE.T refers to the D and T functions of $\mathsf{DE}(H)$, respectively. Thus, DE.$D(s, x)$ is the value obtained by evaluating the right-hand side of (69). Assignment is denoted := and local variables are introduced with **let**.

Procedure 1 – Function $D(s, x)$ of the CT composite actor.

```
Input: $s \in \widehat{S}, x \in \widehat{I}$
Output: $h \in \mathbb{R}_+$
  $h := \min\{s(\text{stepSize}), \text{DE}.D(s, x)\}$;
  loop
    (LTEs, _) = Runge-Kutta23($s, x, h$);
    if $\max\{\text{LTEs}(i) \mid i \in \mathsf{Ix}\} >$ errorTolerance then
      $h := h/2$;
    else
      break;
    end if
  end loop
  return h;
```

The Runge-Kutta procedure roughly works by performing two smaller integration steps at times $0.5 \cdot h$ and $0.75 \cdot h$ from the current time. Before each integration the fixpoint y^* of function

Procedure 2 – Function $T(s, x, d)$ of the CT composite actor.

```
Input: $s \in \widehat{S}, x \in \widehat{I}, d \in \mathbb{R}_+$
Output: $s' \in \widehat{S}$
  let stepPrediction : $\mathsf{Ix} \to \mathbb{R}_+$;
  (LTEs, $z$) := Runge-Kutta23($s, x, d$);
  $s' := \mathrm{DE}.T(s, x, d)$;
  for $i \in \mathsf{Ix}$ do
    $s' := \{s' \mid m_i \mapsto P_i(s \restriction_{\{m_i\}}, z \restriction_{\{v_i\}})\}$;
    $\mathrm{stepPrediction}(i) := d \cdot (\mathrm{errorTolerance}/\mathrm{LTEs(i)})^{1/3}$;
  end for
  $s'(\mathrm{stepSize}) := \min\{\mathrm{maxStepSize}, \mathrm{stepPrediction}(i) \mid i \in \mathsf{Ix}\}$;
  return $s'$;
```

$\tilde{F}$ is computed as in the SR semantics and the value of the state variable m of each integrator in the model is updated.

The deadline procedure refines the step size of the integration until the local truncation errors reported for each integrator by the Runge-Kutta procedure are all less than *errorTolerance*. In cases where there is no guarantee of convergence, i.e., that the loop of Procedure 1 will terminate, an additional parameter bounding the number of iterations may be used to enforce termination.

The time-update procedure updates the actor states in two steps. First it runs a time-update function as defined for the DE director and second it uses the *Runge-Kutta* procedure to calculate the inputs of the integrator actors at the end of an integration with a step size d, which is the amount of time to elapse. For every actor that is not an integrator the new state is equal to the result of the DE time-update function. For each integrator, the state variable m is updated to the value of its input after the integration. Finally the state variable $stepSize$ of the director is updated using a prediction from each integrator for the next integration step size value.

An example of a CT Ptolemy model is shown in Figure 7. This model captures a nonlinear feedback system that exhibits chaotic behavior, known as a Lorenz attractor. The model contains three $\mathsf{Integrator}_\alpha$ actors,[18] all with initial state $\alpha = 1.0$. The model also contains three actors of type `Expression`, which can model generic stateless input-output functions. For example, the actor Expression 1 has two inputs, x_1 and x_2, and its output is equal to $\sigma \cdot (x_2 - x_1)$, where σ is a parameter set to 10.0 in this model. The outputs of the first two integrators, plotted in 2 dimensions over time, are shown to the right of the figure.

A model that mixes the CT and DE domains is shown in Figure 8. The top-level actor of this model is a composite DE actor modeling a simple closed-loop control system. The system to be controlled is a helicopter and the controller is a simple periodic sampling proportional controller. The top-level actor contains a composite CT actor called `Helicopter Model`. The internals of this actor are shown at the bottom-left of the figure and the output of the model at the bottom-right. The $\mathsf{Const}_{0.0}$ actor specifies the desired state of the plant. As seen from the plot, the plant converges to that state after about 3 time units.

[18] As with Clocks, Integrators in Ptolemy have additional inputs that we will ignore for the purposes of this paper.

Procedure 3 – Runge-Kutta23(s_0, x, h).

```
Input: $s_0 \in \widehat{S}, x \in \widehat{I}, h \in \mathbb{R}_+$
Output: LTEs : $\mathsf{Ix} \to \mathbb{R}_+, z \in \widehat{I_\int}$
  let $k_0, k_1, k_2 \in \widehat{I_\int}$;
  let $s_1, s_2, s_3 \in \widehat{S}$;
  $k_0 := (x, y^*_{s_0,x}) \upharpoonright_{I_\int}$;
  $s_1 := s_0$;
  for $i \in \mathsf{Ix}$ do
    $s_1 := \{s_1 \mid \mathrm{m}_i \mapsto x(v_i) + \frac{1}{2} \cdot h \cdot k_0(v_i)\}$;
  end for
  $k_1 := (x, y^*_{s_1,x}) \upharpoonright_{I_\int}$;
  $s_2 := s_1$;
  for $i \in \mathsf{Ix}$ do
    $s_2 := \{s_2 \mid \mathrm{m}_i \mapsto x(v_i) + \frac{3}{4} \cdot h \cdot k_1(v_i)\}$;
  end for
  $k_2 := (x, y^*_{s_2,x}) \upharpoonright_{I_\int}$;
  $s_3 := s_2$;
  for $i \in \mathsf{Ix}$ do
    $z(v_i) := x(v_i) + h \cdot \big(\frac{2}{9}\, k_0(v_i) + \frac{1}{3}\, k_1(v_i) + \frac{4}{9}\, k_2(v_i)\big)$;
    $s_3 := \{s_3 \mid \mathrm{m}_i \mapsto z(v_i)\}$;
  end for
  $k_3 := (x, y^*_{s_3,x}) \upharpoonright_{I_\int}$;
  for $i \in \mathsf{Ix}$ do
    LTEs($i$) $:= h \cdot \big(-\frac{5}{72}\, k_0(v_i) + \frac{1}{12}\, k_1(v_i) + \frac{1}{9}\, k_2(v_i) - \frac{1}{8}\, k_3(v_i)\big)$;
  end for
  return (LTEs, $z$);
```

Most of the actors used in the model of Figure 8 have been defined formally in Section 4. The `Scale` actors simply multiply their input by a constant number. The `ZeroOrderHold` actor serves to transform the discrete-event input `port` of `Helicopter Model` to a "continuous-time" signal, that is, a signal with no absent values, as required by our CT director. `ZeroOrderHold` is essentially the same as the Mem actor (Section 4.5.5) with only difference on how absent inputs are handled. In Mem, absent is handled like any other value, i.e., it can be stored in the memory and produced as output. In `ZeroOrderHold`, only non-absent values can be stored in memory. Thus, the postfire function of `ZeroOrderHold` is as follows (contrast this to Definition (25)):

$$P(s,x) \quad = \quad \begin{cases} \{m \mapsto x(v)\} & \text{if } x(v) \neq \mathsf{absent} \\ \{m \mapsto s(m)\} & \text{otherwise} \end{cases} \tag{73}$$

The `Sampler` samples its input at discrete points in time, as provided by the input `trigger` of `Helicopter Model`, which is generated by a periodic clock actor Clk. That is, every time the `Sampler` is fired, it checks its trigger input: if the trigger is absent, the output is absent; otherwise, the output is equal to the input. `Sampler` is untimed and stateless.

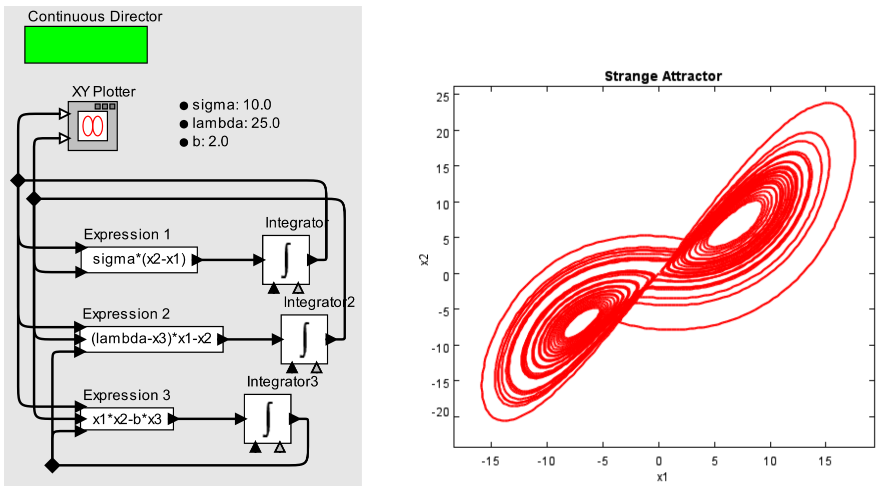


Fig. 7. A CT model (left) and its output (right).

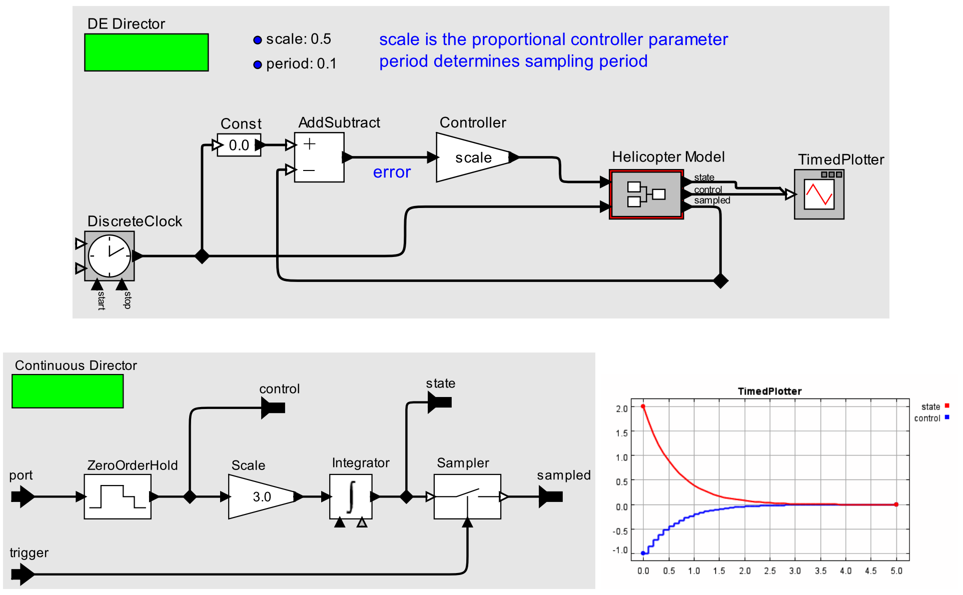


Fig. 8. A model mixing the DE and CT domains: top-level model (top), internals of the Helicopter Model composite actor (bottom-left) and output of the model (bottom-right).

6.4. *Process Networks (PN)*

The PN domain is often used to capture asynchronous distributed systems that communicate through FIFO queues. In general, in such systems the order of execution of individual actors (*interleaving*) influences the results. In the Kahn PN model, however, which we follow here, the semantics is independent from interleaving. This is a desirable feature as it offers determinism in the presence of asynchrony.

Let $H = \{A_1, ..., A_n\}$ be a block diagram, with $A_i = (I_i, O_i, S_i, s_{0,i}, F_i, P_i, D_i, T_i)$, for $i = 1, ..., n$. H can be composed using the PN director provided *every actor A_i is a dataflow, untimed and delay-independent actor*. The fact that A_i is dataflow means that variables in I_i and O_i range over streams. Then, the *PN composite actor* is defined to be the actor

$$\mathsf{PN}(H) = (I, O, S, s_0, F, P, D, T)$$

where I, O, S, s_0 are defined as in the case of $\mathsf{SR}(H)$.

The semantics for PN follows the semantics of Kahn Process Networks (Kahn, 1974). Specifically, F is defined using a fixpoint semantics similarly to the SR case, but with a major difference: in the SR case the CPO is the flat CPO shown in Figure 4, whereas in the PN case the CPO is the set of all (finite or infinite) streams ordered with the *prefix order*. Stream ρ is a prefix of stream ρ' iff there exists stream ρ'' such that $\rho' = \rho \cdot \rho''$, where $\cdot$ denotes stream concatenation. Notice that $\rho' = \rho \cdot \rho''$ implies that either ρ is finite, or it is infinite in which case $\rho' = \rho$ and $\rho'' = []$. The least element in the CPO of streams is the empty stream $[]$ (i.e., the sequence of length 0). Actors are required to be monotonic (in fact, continuous) in this CPO, essentially meaning that longer input streams can only result in longer output streams. Then, the definitions of F and P for PN can be given by the same Equations (62)–(65) used for SR, and will therefore not be repeated here. This definition ensures that continuity is preserved, that is, if all actors A_i are continuous then so is the resulting composite actor $\mathsf{PN}(H)$.

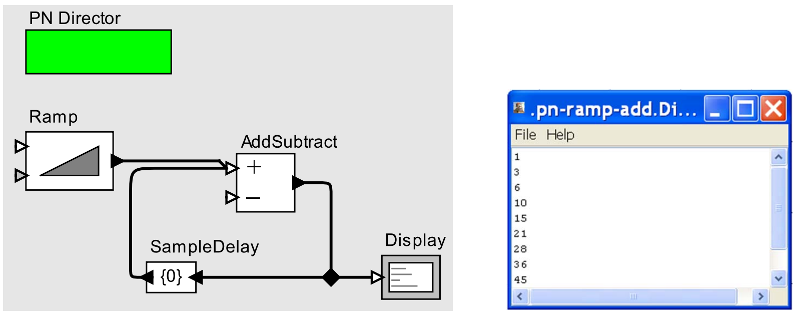
The definition of D for PN can also follow the definition for SR. PN is essentially an untimed model of computation, but, like SR, could be given timed semantics for convenience of embedding it into timed models. The definition of T for PN is also as in SR: $\mathsf{PN}(H)$ is a delay-independent actor, like its sub-actors.

A simple Ptolemy model that uses the PN director is shown in Figure 9. This model contains five actors: `Ramp`, `AddSubtract`, `SampleDelay`, `Display` and Fanout. `Ramp` produces the infinite stream $[1, 2, \cdots]$. `SampleDelay` produces as output the stream it receives as input prefixed by the element 0. For example, if it receives $[1, 2]$ it produces $[0, 1, 2]$. `AddSubtract` is the extension of Add to streams: it adds streams in an element-wise fashion, up to the length of the shorter input stream.[19]

6.5. *Modal Models (MM)*

Modal models are hierarchical models where the top level model consists of an ESM, 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

[19] In Ptolemy, many actors are *polymorphic* in the sense that they can operate under different directors and with different types of inputs and outputs. `AddSubtract` is such an actor: it can add scalars of numerical type (integers, reals, etc.) as well as streams. We do not provide a formal treatment of actor polymorphism, as this would involve a more or less complete type theory for actors, which is beyond the scope of this paper.



```
1
3
6
10
15
21
28
36
45
```

Fig. 9. A PN model (left) and its output (right).

(see, for instance, (Sztipanovits et al., 1993; Simon et al., 2000)). A modal model is an explicit representation of this type of behaviors and the rules that govern transitions between behaviors.

The syntax of modal models is captured by modal-model diagrams, as described in Section 5.2. Let $M = \{A_c, A_1, ..., A_n\}$ be such a diagram, with $A_c = (I_c, O_c, S_c, s_{0,c}, F_c, P_c, D_c, T_c)$, and $A_i = (I_i, O_i, S_i, s_{0,i}, F_i, P_i, D_i, T_i)$, for $i = 1, ..., n$. Recall that A_c is the controller of the modal model, which is an ESM as described in Section 4.5.12. Let $\{\ell_1, ..., \ell_n\}$ be the set of locations of A_c and assume that A_i is the actor refining location ℓ_i, for $i = 1, ..., n$. Recall that by definition of modal-model diagrams (enforced by Ptolemy's syntax), we have $I_c = I_1 = \cdots = I_n$ and $O_c = O_1 = \cdots = O_n$. Without loss of generality, we assume that the initial location of A_c is ℓ_1 (in Ptolemy, an ESM has a single initial location).

The *MM composite actor* is defined to be the actor

$$\mathsf{MM}(M) = (I, O, S, s_0, F, P, D, T)$$

where

$$\begin{aligned} I &= I_c = I_1 = \cdots = I_n && (74)\\ O &= O_c = O_1 = \cdots = O_n && (75)\\ S &= S_c \cup \{\mathsf{tr}\} \cup \bigcup_{i=1}^{n} S_i && (76)\\ s_0 &= (s_{0,c}, \{\mathsf{tr} \mapsto \bot\}, s_{0,1}, ..., s_{0,n}) && (77) \end{aligned}$$

Variable tr is used to "remember" which transition was taken, between the calls of F, P and D, T (this is necessary, because the state and/or inputs may have changed in the meantime, therefore also altering the enabledness of transitions). The value of tr can be one of the following:

— $\bot$: unknown, initially;
— none: no transition taken;
— $\mathsf{preemptive}(i, j)$: preemptive transition taken from ℓ_i to ℓ_j;
— $\mathsf{nonpreemptive}(i, j)$: nonpreemptive transition taken from ℓ_i to ℓ_j.

We proceed to define functions F, P, D, T. We first define F and P. Consider a state $s \in \widehat{S}$ and an input $x \in \widehat{I}$. Let $s = (s_c, \{\mathsf{tr} \mapsto \alpha\}, s_1, ..., s_n)$, with $s_c \in \widehat{S_c}$ and $s_i \in \widehat{S}_i$, for all $i = 1, ..., n$. Suppose the location of A_c at s_c is ℓ_i, for some $i \in \{1, ..., n\}$.

6.5.1. *Strict modal models:* We will first present a *strict* interpretation of the semantics of modal models, where we can assume that the enabledness status of all outgoing transitions from ℓ_i (that is, whether the guard of each transition evaluates to true or false) can be determined. The enabledness of a transition may not be known due to some inputs being unknown (i.e., $\perp$). E.g., if x is an input variable and $x = \perp$, then the guard $x \geq 0$ evaluates to neither *true* nor *false*, but to $\perp$. Figure 10 shows an example, which will be discussed later.

In the strict interpretation, if guards are unknown then all outputs can be also set to be $\perp$. In Section 6.5.2 we provide a non-strict interpretation where outputs can be assigned to values other than unknown, even if some inputs are unknown. This is inspired by non-strict but constructive semantics, such as those we presented for SR, or those used in Esterel.

For the strict interpretation, we distinguish the following cases:

1 There are no outgoing transitions of A_c from location l_i that are enabled at s and x. Then:

$$F(s,x) = F_i(s_i,x) \tag{78}$$
$$P(s,x) = (s_c, \{\mathsf{tr} \mapsto \mathsf{none}\}, s_1, ..., s_{i-1}, P_i(s_i,x), s_{i+1}, ..., s_n) \tag{79}$$

2 There exists a preemptive outgoing transition from ℓ_i to ℓ_j that is enabled at s and x. Denote the output action and set action of this transition by β and γ, respectively. Then:

$$F(s,x) = \beta(s,x) \tag{80}$$
$$P(s,x) = (s'_c, \{\mathsf{tr} \mapsto \mathsf{preemptive}(i,j)\}, s_1, ..., s_{j-1}, s'_j, s_{j+1}, ..., s_n) \tag{81}$$

where:

(a) $\beta(s,x)$ denotes the output obtained by applying the output action β to s and x,
(b) s'_c is obtained by applying the set action γ to s and x and setting the location of A_c to ℓ_j,
(c) s'_j is obtained by applying the set action γ to s and x.

3 There are no preemptive outgoing transitions from ℓ_i that are enabled at s and x, but there exists a non-preemptive outgoing transition from ℓ_i to ℓ_j that is enabled at s and x. Denote the output action and set action of this transition by β and γ, respectively. Then:

$$F(s,x) = \beta(s,x,F_i(s,x)) \tag{82}$$
$$P(s,x) = (s'_c, \{\mathsf{tr} \mapsto \mathsf{nonpreemptive}(i,j)\}, s_1, ..., s'_i, ..., s'_j, ..., s_n) \tag{83}$$

where:

(a) $\beta(s,x,F_i(s,x))$ denotes the output obtained by first computing the output $y = F_i(s,x)$ and then applying the output action β to y, s and x,
(b) s'_c is obtained as in Case 2b,
(c) s'_j is obtained as in Case 2c,
(d) s'_i is obtained by first computing $s''_i = P_i(s_i,x)$ and then applying the set action γ to s''_i,

Item 1 treats the case where no transition of the controller is enabled: in this case, the location of the controller remains unchanged and 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 the outputs and state updates are produced solely by the output and set actions of the transition. We will not define formally the effect of those actions, as they are standard. Note that the set action γ may *reset* some state variables of the target refinement A_j to their initial values. This is done in particular if the transition is a reset transition. Also, γ may "copy" the value of some other state variable, possibly of another refinement, to a state variable of the target refinement. In particular, when modeling hybrid systems (Manna and Pnueli, 1992), the state variables in all refinements are typically the same, i.e., copies of each other. In that case, when moving from one mode of operation to the next, the state of the system must be preserved. This can be done with the appropriate set actions.

Item 3 treats the case where only non-preemptive transitions of the controller are enabled. In this case, before taking such a transition, the current refinement M_i is fired and postfired.

Ptolemy modal models are inspired by formalisms such as Statecharts (Harel, 1987), Esterel (Berry, 1996) and SyncCharts (André, 1996), which also include concepts such as preemptive vs. non-preemptive transitions, albeit sometimes using different terminology (for instance, Esterel uses the terms *strong* and *weak abortion*, respectively).

6.5.2. *Non-strict modal models:* The above completes the description of the semantics for the case where all outgoing transitions from the current location ℓ_i have known enabledness status. In general, however, a Modal Model may be used in situations where $\widehat{I}$ and $\widehat{O}$ are in spaces that can include $\perp$. Then, the behavior of Modal Models must accomodate the non-strict evaluation of transition guards. This is the case, for instance, when a Modal Model is a composite inside of a SR model. The behavior of the Modal Model in an SR model must be defined over partially determined inputs for a constructive fixed point to be determined. When inputs involve bottom values, transition guards are lifted into standard non-strict logic where logical connectives are modeled as is described by the following truth table (the operations are symmetric):

a	b	$\neg b$	$a \wedge b$	$a \vee b$
$true$	$true$	$false$	$true$	$true$
$true$	$\perp$	$\perp$	$\perp$	$true$
$true$	$false$	$true$	$false$	$true$
$\perp$	$\perp$	$-$	$\perp$	$\perp$
$\perp$	$false$	$-$	$false$	$\perp$
$false$	$false$	$-$	$false$	$false$

The predicates themselves may vary in their definition over non-strict values depending on the nature of the predicate. Equality for instance is usually interpreted strictly, such that either operand being bottom evaluates the predicate to logical bottom. As a consequence of these semantics for the evaluation of guards, given a particular set of inputs, each guard can either be evaluated to $true$, $false$, or $\perp$. Given the implicit prioritization of preemptive transitions over non-preemptive ones, certain additional constraints can be given to predicate the enabling and disabling of some transitions.

For example, if a preemptive transition is known to be enabled, then the enabledness status of nonpreemptive transitions is irrelevant.[20] If after these additional constraints are used, only one transition can be taken, then the transition is taken in the manner described above for the case where the enabledness status of all transitions is known. Likewise, if no transition can be taken, then the current refinement is executed and the location does not change, as described above.

However, if there are some transitions the guards of which are $\bot$, and hence given more information in the input (a monotonic increase in the input domain) one of several transitions can be taken, the set of possible actions on output signals taking each of these transitions or taking none of them must be considered. In the circuit-style constructive semantics as described in (Berry, 1996), the presence of a signal cannot be concluded unless it is made present by a totally defined behavior, but if amongst all determined possible actions an output value is not declared as having a present value it can be conclusively set to absent.

To make this notion exact, let there be two sets T_P and T_N representing respectively the preemptive and non-preemptive transitions emanating from the current location ℓ_i. For a given set of transitions X, let $\mathsf{can}(X)$ denote the subset of transitions in X the guards of which evaluate non-strictly to $true$ or $\bot$ (not $false$). Also, let $\mathsf{must}(X)$ be the subset of transitions in X the guards of which evaluate to $true$. (Note that by the determinism assumption $\mathsf{must}(T_P)$ and $\mathsf{must}(T_N)$ are either singletons or empty.) Finally, let there be a special transition ϵ representing not taking any actual transition.

The algorithm to determine this non-strict behavior is as follows:

$$\begin{array}{r|c|l} \textit{if} & \mathsf{must}(T_P) = \{t_p\} & \mathsf{take}(t_p) \\ \textit{else if} & \mathsf{can}(T_P) = \emptyset \wedge \mathsf{must}(T_N) = \{t_n\} & \mathsf{take}(t_n) \\ \textit{else if} & \mathsf{can}(T_P) = \mathsf{can}(T_N) = \emptyset & \mathsf{take}(\epsilon) \\ \textit{else} & & \mathsf{check}(\mathsf{can}(T_P) \cup \mathsf{can}(T_N) \cup \{\epsilon\}) \end{array}$$

where take means to take the corresponding transition as described for the strict case (Section 6.5.1). The function check takes a set of possible transitions X and checks whether the actions of those transitions can declare present values on the outputs. An output is determined to be absent if no such declarations can be found; otherwise, the output is set to $\bot$. It is beyond the scope of this paper to provide a complete definition of check, as this involves static analysis of the action language of Ptolemy, the details of which have been omitted. Instead, we illustrate the behavior of check in the example that follows.

Figure 10 shows a simple model that can be interpreted under these non-strict semantics. At *State 1*, the guards of both non-preemptive outgoing transition express the predicates $x = \mathsf{absent}$ and $\neg(x = \mathsf{absent})$. If the input variable x is valued at $\bot$, under the non-strict logic given above both predicates will evaluate to $\bot$. Intuitively, this means either of the transitions might be enabled given more information about x. Since both predicates evaluate to $\bot$ and there are no preemptive transitions, $\mathsf{can}(T_P) = \mathsf{must}(T_P) = \mathsf{must}(T_N) = \emptyset$, and $\mathsf{can}(T_N)$ contains both transitions. This means that the output y must be determined by function check applied on these two transitions. Assuming that the refinements of both locations are empty, i.e., that this modal model is simply an ESM, check can “safely” conclude that y is absent. This is because the actions associated with both transitions emanating from *State 1* are empty.

[20] Note that the case where more than one transitions are enabled is excluded by our determinism assumption. Therefore, if one preemptive transition is known to be enabled, the rest must be disabled.

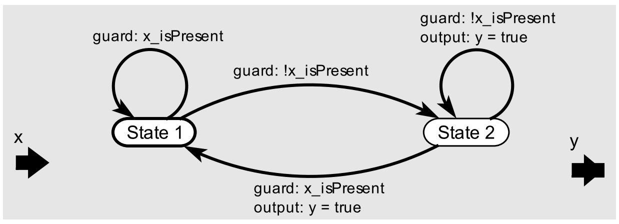


Fig. 10. A model that under the non-strict interpretation, given unknown input $x = \perp$, concludes that output $y =$ absent from *State 1*, but $y = \perp$ from *State 2*.

The situation is similar at *State 2*. Again, $\mathsf{can}(T_P) = \mathsf{must}(T_P) = \mathsf{must}(T_N) = \emptyset$, and $\mathsf{can}(T_N)$ contains both transitions emanating from *State 2*. Again, check must be used to determine the value of the output y. In this case, however, and despite the fact that y is set to *true* in both transitions emanating from *State 2*, check concludes that $y = \perp$, which implies that this model has ambiguous semantics. This model exemplifies the biasing of output values to absent, in accordance with a constructive interpretation such as the one of (Berry, 1996).

6.5.3. *Deadline and time-update functions for modal models:* We now define D and T. Consider a state $s \in \widehat{S}$, an input $x \in \widehat{I}$ and a delay $d \in \mathbb{R}_+$. Let $s = (s_c, \{\mathsf{tr} \mapsto \alpha\}, s_1, ..., s_n)$, with $s_c \in \widehat{S_c}$ and $s_i \in \widehat{S_i}$, for all $i = 1, ..., n$. We distinguish cases:

1 $\alpha =$ none. Suppose the location of A_c at s_c is ℓ_i, for some $i \in \{1, ..., n\}$. Then:

$$D(s,x) = D_i(s_i, x) \tag{84}$$
$$T(s,x,d) = (s_c, \{\mathsf{tr} \mapsto \perp\}, s_1, ..., s_{i-1}, T_i(s_i, x, d), s_{i+1}, ..., s_n) \tag{85}$$

2 $\alpha = \mathsf{preemptive}(i, j)$. Then:

$$D(s,x) = D_j(s_j, x) \tag{86}$$
$$T(s,x,d) = (s_c, \{\mathsf{tr} \mapsto \perp\}, s_1, ..., s_{j-1}, T_j(s_j, x, d), s_{j+1}, ..., s_n) \tag{87}$$

3 $\alpha = \mathsf{nonpreemptive}(i, j)$. Then D and T are defined as in (86) and (87).

In Case 1 no transition was taken during fire and postfire, and the deadline and time-update are determined by the current refinement A_i. In Case 2, where a preemptive transition was taken, the deadline and time-update are determined by the target refinement A_j. The same holds when a non-preemptive transition was taken (Case 3).

6.5.4. *Examples:* We illustrate modal models with the example of Figure 3. This is a timed modal model, in the sense that the top level actor and the refinements of the `ModalModel` actor are DE models. The `ModalModel` actor switches between two modes every 2.5 time units, according to the events it receives by the outermost `DiscreteClock`. In the `regular` mode it generates a regularly-spaced clock signal with period 1.0 and value 1. 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 11, with a shaded background showing the times over which it is in the two modes. A variant of the same model is shown in Figure 12, the difference being that the transition from `regular` to

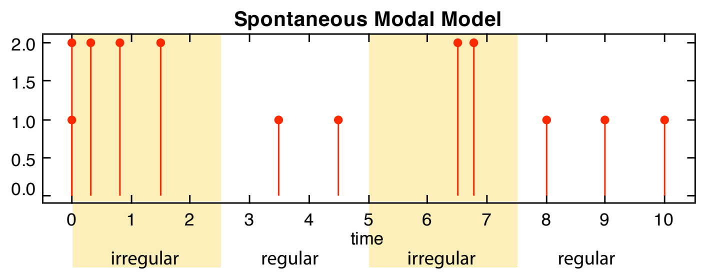


Fig. 11. A plot of the output from one run of the model in Figure 3.

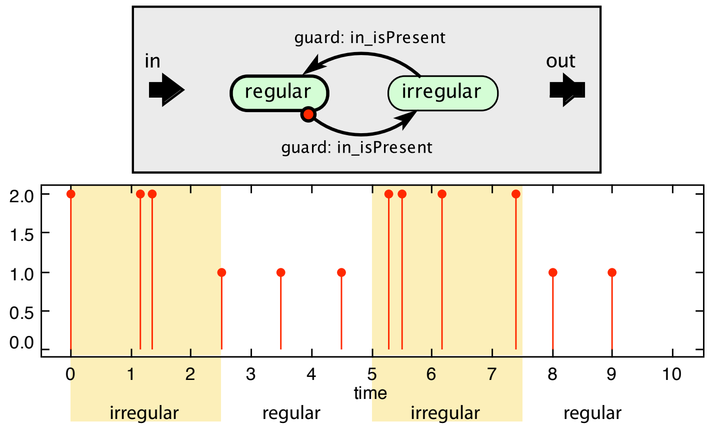


Fig. 12. A variant of Figure 3 where a preemptive transition prevents the initial firing of the innermost `DiscreteClock` actor of that model.

`irregular` is now preemptive. A number of observations worth making arise from the two plots:

First, note that two events are generated at time 0, a first event with value 1, and a second event with value 2. The first event is produced by (the innermost) `DiscreteClock`, according to the semantic rules of Case 3a. If we had instead used a preemptive transition, as shown in Figure 12, then that first output event would not appear: this is according to the semantic rules of Case 2a and the fact that the transitions of this model contain no output or set actions.

In both cases (non-preemptive and preemptive transition), the event at time 0 with value 2 is produced by `PoissonClock`, according to the fire and postfire semantic rules, Case 1. The reason why this event occurs at time 0, even in the variant of the model with the non-preemptive transition, is the rule for determining the deadline in a modal model (Cases 2 and 3). When the model is initialized, the timer of `PoissonClock` is set to zero. In both non-preemptive and preemptive cases, the `irregular` location is entered after postfiring the modal model for the first time. Therefore, the deadline function of the refinement of `irregular` (i.e., of `PoissonClock`) is used to determine the deadline of the modal modal. When the deadline function of `PoissonClock` is called at time 0, it returns 0, since the timer of `PoissonClock` is 0. This forbids time from advancing until the `PoissonClock` is fired.

Another interesting observation concerns the output events with value 1 occurring at times 3.5, 4.5, 8, and so on, in the plot of Figure 11. These events occur at times during which the model is at the `regular` mode. As explained above, the model begins in the `regular` mode but spends zero time there, since it immediately transitions to the `irregular` mode. In the non-preemptive case, the `DiscreteClock` is postfired before entering `irregular` (Case 3d). Hence, the timer of `DiscreteClock` has value 1 when `irregular` is entered. When `regular` is re-entered at time 2.5, this timer still has value 1, since it has not been updated in the meantime. It therefore expires one time unit later, i.e., at time 3.5, which explains the event at that time. Moreover, the timer is reset to 1 during postfire(). It expires again one 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.

Consider also the event with value 1 at time 2.5, in the plot of Figure 12. This event is generated when `regular` is re-entered at time 2.5. In the case of a preemptive transition, `DiscreteClock` has not been postfired at time 0, therefore, its timer has value 0 at time 2.5. This explains the event at that time.

7. Conclusions and Future Work

This paper proposes a formal semantics for the heterogeneous modeling environment Ptolemy. The semantics is modular in the sense that it unifies atomic and composite actors within a single executable interface. Directors, which realize specific models of computation, are composition operators: they take as input a diagram of actors and return a new (composite) actor. Because composite actors have the same interface as atomic actors, they can be seen as “black boxes”. This enables reasoning about a hierarchical model in a modular, actor-by-actor basis, thus minimizing the complexity that arises from cross-cutting dependencies between levels of the hierarchy.

An implementation of our framework in the functional programming language Haskell is ongoing. Prototype implementations of the SR, DE, CT and PN domains are currently available. Implementation of MM is under way. Our goal for the Haskell-based implementation is not to be a replacement for the current Java implementation of Ptolemy (the primary reason being the better performance of the latter). Instead we intend to use it mainly as a tool for validating the semantics. This validation can be performed by comparing the results of the two implementations on different models. All the examples described in this paper, with the exception of the example containing a modal model (Figure 3), have been modeled both in Ptolemy and in the Haskell implementation, and give equivalent results, modulo minor numerical discrepancies.

Adding more models of computation to the collection presented in this paper is one of the directions for future work. Another direction is to examine the properties of the directors as composition operators, for instance, properties related to associativity and commutativity. We hope that this will lead to a better understanding of the properties of heterogeneous modeling in general. We also hope that this framework will serve as a starting point for a more in-depth discussion on how to unify different models of computation (for instance, which are the possible actor interfaces and what are their relative merits?) as well as how to formally represent them (e.g., what would be the right language for writing directors?).

References

Aiguier, M., Golden, B., and Krob, D. (2011). Complex Systems Modeling II: A minimalist and unified semantics for heterogeneous integrated systems. *Applied Mathematics and Computation*. To appear.

Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T. A., Ho, P.-H., Nicollin, X., Olivero, A., Sifakis, J., and Yovine, S. (1995). The algorithmic analysis of hybrid systems. *Theor. Comput. Sci.*, 138:3–34.

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

André, C. (1996). SyncCharts: A visual representation of reactive behaviors. Technical Report RR 95–52, rev. RR (96–56), I3S.

André, C. (2003). Semantics of S.S.M (Safe State Machine). Technical report, Esterel Technologies.

Arbab, F. (2004). Reo: A channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366.

Arnold, A. and Nivat, M. (1980). Metric interpretations of infinite trees and semantics of non deterministic recursive programs. *Fundamenta Informaticae*, 11(2):181–205.

Bae, K., Csaba Olveczky, P., Feng, T. H., Lee, E. A., and Tripakis, S. (Accepted for publication, Oct 2010). Verifying Hierarchical Ptolemy II Discrete-Event Models using Real-Time Maude. *Science of Computer Programming*.

Baier, C. and Majster-Cederbaum, M. E. (1994). Denotational semantics in the CPO and metric approach. *Theoretical Computer Science*, 135(2):171–220.

Balarin, F., Hsieh, H., Lavagno, L., Passerone, C., Sangiovanni-Vincentelli, A. L., and Watanabe, Y. (2003). Metropolis: an integrated electronic system design environment. *Computer*, 36(4).

Basu, A., Bozga, M., and Sifakis, J. (2006). Modeling heterogeneous real-time components in BIP. In *International Conference on Software Engineering and Formal Methods (SEFM)*, pages 3–12, Pune.

Beeck, M. v. d. (1994). A comparison of Statecharts variants. In Langmaack, H., de Roever, W. P., and Vytopil, J., editors, *3rd Intl. Symp. Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *LNCS*, pages 128–148, Lübeck, Germany. Springer-Verlag.

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

Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Le Guernic, P., and de Simone, R. (2003). The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83.

Benveniste, A., Caspi, P., Lublinerman, R., and Tripakis, S. (2009). Actors without directors: a Kahnian view of heterogeneous systems. In *Hybrid Systems – Computation and Control (HSCC)*, volume LNCS. Springer.

Berry, G. (1996). *The Constructive Semantics of Pure Esterel*. Book Draft.

Bliudze, S. and Krob, D. (2009). Modelling of complex systems: Systems as dataflow machines. *Fundam. Inf.*, 91:251–274.

Bliudze, S. and Sifakis, J. (2008a). The algebra of connectors - structuring interaction in bip. *IEEE Trans. Computers*, 57(10):1315–1330.

Bliudze, S. and Sifakis, J. (2008b). A notion of glue expressiveness for component-based systems. In van Breugel, F. and Chechik, M., editors, *CONCUR 2008*, volume 5201 of *LNCS*, pages 508–522. Springer.

Boulanger, F., Hardebolle, C., Jacquet, C., and Marcadet, D. (2011). Semantic Adaptation for Models of Computation. In *Proceedings of ACSD 2011 (Application of Concurrency to System Design)*, pages 153–162. IEEE Computer Society.

Bozga, M., Sfyrla, V., and Sifakis, J. (2009). Modeling synchronous systems in BIP. In Chakraborty, S. and Halbwachs, N., editors, *EMSOFT*, pages 77–86. ACM.

Broy, M. and Stolen, K. (2001). *Specification and Development of Interactive Systems*, volume 62 of *Monographs in Computer Science*. Springer.

Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., and Stefani, J.-B. (2006). The FRACTAL component model and its support in Java. *Softw. Pract. Exper.*, 36:1257–1284.

Buck, J. T. (1993). *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. Ph.d. thesis, University of California, Berkeley.

Buck, J. T., Ha, S., Lee, E. A., and Messerschmitt, D. G. (1994). Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. Journal of Computer Simulation, special issue on "Simulation Software Development"*, 4:155–182.

Burch, J. R., Passerone, R., and Sangiovanni-Vincentelli, A. L. (2001). Overcoming heterophobia: Modeling concurrency in heterogeneous systems. In *International Conference on Application of Concurrency to System Design*, page 13.

Cataldo, A., Lee, E., Liu, X., Matsikoudis, E., and Zheng, H. (2006). A constructive fixed-point theorem and the feedback semantics of timed systems. In *Proceedings of the 8th International Workshop on Discrete-Event Systems (WODES'06)*.

Damm, W., Josko, B., Hungar, H., and Pnueli, A. (1998). A Compositional Real-time Semantics of STATEMATE Designs. In *Compositionality: The Significant Difference*, volume 1536 of *LNCS*, pages 186–238. Springer.

Davey, B. A. and Priestley, H. A. (2002). *Introduction to Lattices and Order*. Cambridge University Press, 2nd edition.

Denckla, B. and Mosterman, P. (2008). Stream- and state-based semantics of hierarchy in block diagrams. In *17th IFAC World Congress*, pages 7955–7960.

Dijkstra, E. W. (1976). *A Discipline of Programming*. Prentice Hall.

Dill, D. L. (1988). *Trace theory for automatic hierarchical verification of speed-independent circuits*. Ph.d., Carnegie-Mellon University.

Edwards, S. A. and Lee, E. A. (2003). The semantics and execution of a synchronous block-diagram language. *Science of Computer Programming*, 48(1).

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

Eshuis, R. (2009). Reconciling statechart semantics. *Sci. Comput. Program.*, 74(3):65–99.

Feredj, M., Boulanger, F., and Mbobi, A. M. (2009). A model of domain-polymorph component for heterogeneous system design. *The Journal of Systems and Software*, 82:112–120.

Floyd, R. W. (1967). Assigning meaning to programs. In *Proc. of Symposium on Applied Mathematics*, volume 19, pages 19–32. A.M.S.

Geilen, M. and Basten, T. (2003). Requirements on the execution of Kahn process networks. In *European Symposium on Programming Languages and Systems*, LNCS, pages 319–334. Springer.

Geilen, M. and Basten, T. (2004). Reactive process networks. In *4th ACM international conference on Embedded Software*, EMSOFT '04, pages 137–146. ACM.

Goderis, A., Brooks, C., Altintas, I., Lee, E. A., and Goble, C. (2009). Heterogeneous composition of models of computation. *Future Generation Computer Systems*, 25(5):552 – 560.

Goessler, G. and Sangiovanni-Vincentelli, A. (2002). Compositional modeling in Metropolis. In *Second International Workshop on Embedded Software (EMSOFT)*, Grenoble, France. Springer-Verlag.

Graf, S., Ober, I., and Ober, I. (2006). A real-time profile for UML. *Soft. Tools Tech. Transfer*, 8(2):113–127.

Gurevich, Y. (1993). Evolving algebras: An attempt to discover semantics. In Rozenberg, G. and Salomaa, A., editors, *Current Trends in Theoretical Computer Science*, pages 266–292. World Scientific.

Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. (1991). The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1319.

Hamon, G. (2005). A denotational semantics for stateflow. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 164–172, New York, NY, USA. ACM.

Hamon, G. and Rushby, J. (2004). An operational semantics for Stateflow. In *Fundamental Approaches to Software Engineering (FASE)*, volume 2984 of *LNCS*, pages 229–243, Barcelona, Spain. Springer.

Hardebolle, C., Boulanger, F., Marcadet, D., and Vidal-Naquet, G. (2007). A generic execution framework for models of computation. In *Proceedings of the Fourth International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, MOMPES '07, pages 45–54, Washington, DC, USA. IEEE Computer Society.

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

Herrera, F. and Villar, E. (2006). A framework for embedded system specification under different models of computation in SystemC. In *Design Automation Conference (DAC)*, San Francisco. ACM.

Jantsch, A. (2003). *Modeling Embedded Systems and SoCs - Concurrency and Time in Models of Computation*. Morgan Kaufmann.

Kahn, G. (1974). The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*. North-Holland Publishing Co.

Karsai, G. (1995). A configurable visual programming environment: A tool for domain-specific programming. *IEEE Computer*, pages 36–44.

Kohavi, Z. (1978). *Switching and finite automata theory*. McGraw-Hill.

Ledeczi, A., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J., and Karsai, G. (2001). Composing domain-specific design environments. *IEEE Computer*, pages 44–51.

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

Lee, E. A. (2009). Finite State Machines and Modal Models in Ptolemy II. Technical Report UCB/EECS-2009-151, EECS Department, University of California, Berkeley.

Lee, E. A. (2010). Disciplined heterogeneous modeling. In D.C. Petriu, N. Rouquette, O. H., editor, *Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering, Languages, and Systems (MODELS)*, pages 273–287. IEEE.

Lee, E. A. and Matsikoudis, E. (2009). The semantics of dataflow with firing. In Huet, G., Plotkin, G., Lévy, J.-J., and Bertot, Y., editors, *From Semantics to Computer Science: Essays in memory of Gilles Kahn*. Cambridge University Press.

Lee, E. A. and Messerschmitt, D. G. (1987). Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245.

Lee, E. A. and Parks, T. M. (1995). Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801.

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

Lee, E. A. and Tripakis, S. (2010). Modal Models in Ptolemy. In *EOOLT 2010 – 3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. Linköping University Electronic Press.

Lee, E. A. and Zheng, H. (2005). Operational semantics of hybrid systems. In Morari, M. and Thiele, L., editors, *Hybrid Systems: Computation and Control (HSCC)*, volume LNCS 3414, pages pp. 25–53, Zurich, Switzerland. Springer-Verlag.

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

Liu, J. and Lee, E. A. (2003). On the causality of mixed-signal and hybrid models. In *6th International Workshop on Hybrid Systems: Computation and Control (HSCC '03)*, Prague, Czech Republic.

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

Liu, X., Matsikoudis, E., and Lee, E. A. (2006). Modeling timed concurrent systems. In *CONCUR 2006 - Concurrency Theory*, volume LNCS 4137, Bonn, Germany. Springer.

Lublinerman, R., Szegedy, C., and Tripakis, S. (2009). Modular Code Generation from Synchronous Block Diagrams – Modularity vs. Code Size. In *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'09)*, pages 78–89. ACM.

Malik, S. (1994). Analysis of cyclic combinational circuits. *IEEE Trans. Computer-Aided Design*, 13(7):950–956.

Manna, Z. and Pnueli, A. (1992). Verifying hybrid systems. *Hybrid Systems*, pages 4–35.

Maraninchi, F. and Bhouhadiba, T. (2007). 42: Programmable models of computation for a component-based approach to heterogeneous embedded systems. In *6th ACM International Conference on Generative Programming and Component Engineering (GPCE)*, pages 1–3, Salzburg, Austria.

Maraninchi, F. and Rémond, Y. (2003). Mode-automata: a new domain-specific construct for the development of safe critical systems. *Sci. Comput. Program.*, 46:219–254.

Meyer, B. (1992). Applying "design by contract". *Computer*, 25(10):40–51.

Nordstrom, G., Sztipanovits, J., Karsai, G., and Ledeczi, A. (1999). Metamodeling - rapid design and evolution of domain-specific modeling environments. In *Proc. of Conf. on Engineering of Computer Based Systems (ECBS)*, pages 68–74, Nashville, Tennessee.

Patel, H. D. and Shukla, S. K. (2004). *SystemC Kernel Extensions for Heterogeneous System Modelling*. Kluwer.

Reed, G. M. and Roscoe, A. W. (1988). Metric spaces as models for real-time concurrency. In *3rd Workshop on Mathematical Foundations of Programming Language Semantics*, pages 331–343, London, UK.

Sander, I. and Jantsch, A. (2004). System modeling and transformational design refinement in ForSyDe. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, 23(1):17–32.

Scaife, N., Sofronis, C., Caspi, P., Tripakis, S., and Maraninchi, F. (2004). Defining and Translating a "Safe" Subset of Simulink/Stateflow into Lustre. In *Proceedings of the 4th ACM Intl. Conf. on Embedded Software (EMSOFT'04)*, pages 259–268. ACM.

Shiple, T., Berry, G., and Touati, H. (1996). Constructive analysis of cyclic circuits. In *European Design and Test Conference (EDTC'96)*. IEEE Computer Society.

Sifakis, J. (1977). Use of petri nets for performance evaluation. In *Measuring, modelling and evaluating computer systems*, pages 75–93. North-Holland.

Simon, G., Kovácsházy, T., and Péceli, G. (2000). Transient management in reconfigurable systems. In *IWSAS' 2000: Proc. 1st Intl. Workshop on Self-Adaptive Software*, pages 90–98, Secaucus, NJ, USA. Springer.

Sztipanovits, J., Wilkes, D., Karsai, G., Biegl, C., and Lynd, L. (1993). The multigraph and structural adaptivity. *IEEE Trans. Signal Proc.*, pages 2695–2716.

Tripakis, S., Bui, D., Geilen, M., Rodiers, B., and Lee, E. A. (Accepted for publication, Dec 2010). Compositionality in Synchronous Data Flow: Modular Code Generation from Hierarchical SDF Graphs. *ACM Transactions on Embedded Computing Systems (TECS)*. Preprint available as a technical report from http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-143.html.

Yates, R. K. (1993). Networks of real-time processes. In Best, E., editor, *Proc. of the 4th Int. Conf. on Concurrency Theory (CONCUR)*, volume LNCS 715. Springer-Verlag.

Zhu, Y., Westbrook, E., Inoue, J., Chapoutot, A., Salama, C., Peralta, M., Martin, T., Taha, W., O'Malley, M., Cartwright, R., Ames, A., and Bhattacharya, R. (2010). Mathematical equations as executable models of mechanical systems. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*, ICCPS '10, pages 1–11. ACM.