A modular formal semantics for Ptolemy

STAVROS TRIPAKIS, CHRISTOS STERGIOU, CHRIS SHAVER and EDWARD A. LEE

Mathematical Structures in Computer Science / Volume 23 / Special Issue 04 / August 2013, pp 834 - 881
DOI: 10.1017/S0960129512000278, Published online: 08 July 2013

Link to this article: http://journals.cambridge.org/abstract_S0960129512000278

How to cite this article:
STAVROS TRIPAKIS, CHRISTOS STERGIOU, CHRIS SHAVER and EDWARD A. LEE (2013). A modular formal semantics for Ptolemy. Mathematical Structures in Computer Science, 23, pp 834-881 doi:10.1017/S0960129512000278

Math. Struct. in Comp. Science (2013), *vol.* 23, *pp.* 834–881.
doi:10.1017/S0960129512000278

A modular formal semantics for Ptolemy[†]

STAVROS TRIPAKIS, CHRISTOS STERGIOU,
CHRIS SHAVER and EDWARD A. LEE

University of California, Berkeley, California, U.S.A.
Email: {stavros;chster;shaver;eal}@eecs.berkeley.edu

Received 23 August 2012

Ptolemy[‡] is an open-source and extensible modelling 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 that 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 the state with the passage of time). Composite actors are obtained using composition operators that in Ptolemy are called directors. Different directors realise different models of computation. In this paper, we formally define the directors for the following models of computation: synchronous-reactive, discrete event, continuous time, process networks and modal models.

1. Introduction

Modelling 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, analysing its behaviour and asking 'what-if' questions. Therefore, having a model of the system before actually building the system allows us to make design decisions based on the results of the analysis. On the other hand, having a model of an existing system allows us to subject the model to experimentation that for some reason the physical system cannot be subjected to, such as cost, size or time scales. Such experimentation can

[†] This work was 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) (which is one of six research centres funded under the Focus Center Research Program), a Semiconductor Research Corporation program and the following companies: Bosch, National Instruments, Thales and Toyota.

[‡] In this paper, 'Ptolemy' refers specifically to Ptolemy II – see http://ptolemy.org.

influence decisions such as whether to make adjustments to the system and for future system evolution.

1.1. *Model composition*

Building large complex systems is not a trivial task. This task is often accompanied by the task of building large complex models, which is itself non-trivial. One of the main difficulties of the modelling task comes from the fact that a large system cannot be modelled in a *monolithic* way. That is, rather than develop a single model that captures the entire system, we develop many smaller models each modelling a part of the system. These sub-models then 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 flavours:

— 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 that allow us to compose models to form a larger model[†].
— 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[‡] model can be written in a block-diagram notation, and so can a SysML[§] 'block definition diagram'. The fact that the two models share a similar notation, however, does not imply that it is easy to compose them since this composition depends strongly 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 to 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, and syntax does matter in modelling and system design. As an extreme thesis, it could be claimed that since every model executable in a computer can be encoded as a Turing machine, Turing machines are the ultimate unifying modelling language! But this language is, of course, not very useful.

  Another practical problem with composing semantically homogeneous models that are not written in the same language or syntax is tool support. The individual models

[†] Even in this simplest case, composition may not be entirely straightforward. This is because the 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 (that is, non-composite) model. Indeed, many languages are not compositional in this sense – see, for instance, Lublinerman *et al.* (2009) and Tripakis *et al.* (2010).

[‡] See http://www.mathworks.com/products/simulink/ for details.

[§] See http://www.omgsysml.org/ for details.

can often 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', for example, in the context of the EU project SPEEDS[†] or the earlier US project MoBIES[‡], but, unfortunately, with only limited success).

In practice, models are often *heterogeneous*, in one or more of the senses mentioned above. That is, they may have a different syntax or semantics, or both. Heterogeneous models arise naturally because different parts of the system have inherently distinct properties, and therefore require different types of model. For instance, it is natural to capture the dynamics of a car using a continuous-time model, but a computerised controller is more naturally described 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, it may be easier to describe in Lustre or Simulink, but 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 the fact that different models are often built by different groups of people with different traditions or processes.

1.2. *Models of computation*

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[§]. Thus, an MoC can be viewed as providing a solution to the model composition problem for homogeneous models. There are a number of existing modelling languages, which realise different MoCs. Many of these languages are gaining acceptance in industry in so-called *model-based design* methodologies. Examples include UML/SysML, Matlab/Simulink/Stateflow, AADL, Modelica and LabVIEW. These types of language are raising the level of abstraction in system design by offering mechanisms to capture concurrency, interaction and time behaviour, all of which are essential aspects of modern systems. Moreover, verification and code generation tools exist for many of these languages, allowing us to go beyond simple modelling and simulation, and facilitating the process of going from high-level models to low-level implementations.

However, despite these advances, these languages offer little or no support for heterogeneity. There is currently no universally accepted solution for heterogeneous modelling.

1.3. *Ptolemy*

The modelling and simulation tool Ptolemy[¶] has formed the basis of 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*

[†] See http://www.speeds.eu.com/ for details.
[‡] See http://w3.isis.vanderbilt.edu/Projects/mobies/ for details.
[§] The term *model of concurrency and communication* (MoCC) is often used instead.
[¶] See http://ptolemy.eecs.berkeley.edu/ for details.

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 (for example, by interleaving, in lock-step or in some other order) and the exact manner in which they communicate (for example, through message passing or shared variables) are not fixed but are defined by an MoC, which is also called a *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 designed to be easily extensible: new domains (that is, new directors) and new actors can be added with relatively little effort.

Currently, Ptolemy supports a number of MoCs and their 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, where we use 'unified' to 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 from its implementation, that is, 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 the conformance of the implementation with the semantics of the language is always a tricky one. Despite this inherent difficulty, a formal semantics is desirable for many reasons, which we will not repeat here as they have been well argued elsewhere (see, for example, Floyd (1967) and Dijkstra (1976)), except to say that we view *conciseness* and *readability* as two of the most important – 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 frequently used subset. Apart from this, 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 through FIFO queues.
— MM capture control using state machines or hierarchical state machines (Harel 1987; André 1996).

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 the set of variables X.

We believe that the approach proposed in this paper is not limited to these domains, and could be extended to other MoCs. For instance, SDF can be seen as a static subclass of KPN and therefore could be captured semantically as such[†].

Ptolemy uses a graphical syntax, with *hierarchy* being the fundamental modularity mechanism at the syntactic level. This means that a model is essentially a tree of submodels. The leaves of the tree correspond to *atomic* actors, which may be 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, and so on, 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 characterises all actors, viewing them as extended, timed state machines. These machines are similar to the *abstract state machines* of Gurevich (1993). They are characterised 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 Mealy state machines (Kohavi 1978). Our machines also include a *deadline* function D and a *time-update* function T, which capture timed actors and the effect that the passage of time has on the state. A summary of the standardised actor interface is given in Figure 1 for reference purposes, and a detailed description of the interface is given in Section 4.

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

[†] An implementation would typically distinguish SDF from PN for reasons of efficiency since SDF allows specialised algorithms for scheduling and analysis.

	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.

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 summarised 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 since the Java code has interdependencies that extend beyond the particular director class and require an examination of extensive parts of the code.

Our semantics, while not directly reflecting the implementation (which would mean formalising the Java code!), aims to come as close to the implementation as possible while maintaining a high-level view that allows us to achieve conciseness and readability. In particular, our formalisation captures the salient features of the implementation, and, in particular, the fire/postfire interface of actors, which is called the *abstract semantics* (Eker *et al.* 2003).

1.4. *Structure of the paper*

The rest of this paper is organised as follows:

— We discuss related work in Section 2.
— Section 3 uses an example to give a brief review of Ptolemy's visual syntax.
— Section 4 gives a formal semantics for actors.
— Section 5 formalises actor diagrams.

— Section 6 formalises directors for the SR, DE, CT, PN and MM domains.
— Finally, we present our conclusions and some suggestions for future work in Section 7.

2. Related work

The semantics of Ptolemy II has been described previously in a number of papers, which 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 only contains 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, but it does not provide a formal semantics either, or 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 Ptolemy domains:
  – Edwards and Lee (2003) presents the basis for the SR semantics.
  – Various formal semantics for DE are presented in Lee (1999), Liu *et al.* (2006), Cataldo *et al.* (2006), Liu and Lee (2008) and Bae *et al.* (2012), the last of these being the closest to the actual Ptolemy implementation.
  – Continuous-time and hybrid systems are considered in Liu and Lee (2003) and 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 model of computation. In particular:
  – Lee and Sangiovanni-Vincentelli (1998) proposed a denotational semantics where processes are viewed 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. However, it is difficult to see how this can capture different domains, which, as mentioned above, are viewed as different composition operators in this paper.
  – Liu and Lee (2008) proposed another denotational semantics, which is based on fixpoints on CPOs with a prefix order. This allows the 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 (for example, DE) in the same framework.

Benveniste *et al.* (2009) proposed a similar denotational semantics, but without the use of a special 'absent' value[†].
It is not clear how this work can be extended to include other MoCs, in particular, continuous-time and modal models.

– Finally, Burch *et al.* (2001) provides an abstract framework reminiscent of trace theory (Dill 1988), which can be viewed as a 'meta-framework' under which the heterogeneous composition of specific MoCs can be formulated. However, it 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 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 captures the tool implementation exactly, we believe our semantics is much closer to the implementation than previous work.

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.

2.1. *Relation to previous work on specific MoCs*

We will now consider how our work is related to previous work for each of the MoCs considered in this paper.

2.1.1. *SR.* 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, to the *constructive* semantics of Esterel (Malik 1994; Berry 1996; Shiple *et al.* 1996). These ideas were adapted in Edwards and Lee (2003) for a block-diagram notation, which is also the notation used in Ptolemy for SR. Our semantics follows that 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.

2.1.2. *PN.* 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, but it is a different CPO from the one used in SR. In SR, inputs and outputs are individual

[†] This results in some loss of expressiveness, which is attested to by the fact that the Adder actor cannot be specified satisfactorily.

values, but in PN they are *streams*, that is, 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, the monotonicity of functions does not generally imply Scott-continuity, though Scott-continuity is a reasonable assumption to make, and in practice is satisfied by actors.

The PN semantics is a denotational semantics. The stream CPO has infinite height, so 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).

2.1.3. *DE.* The semantics of DE has been a topic of discussion for many years (Reed and Roscoe 1988; Yates 1993), and 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, such as timed automata (Alur and Dill 1994). A timed automaton has a finite number of clocks, but 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 as far as we are aware, 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, and provides formal refinement relations and guarantees of compositionality.

2.1.4. *CT.* The semantics of CT is based on numerical methods for solving differential equations. Combinations of CT models with discrete logic (for example, 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 is still an active area of research (see Zhu *et al.* (2010), for example). 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 behaviour. Even when we do not have closed form solutions, numerical solutions yield exact answers (using appropriate solvers) for many special cases. 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 the determinacy of the model). Hence, the idealisation remains useful.

2.1.5. *MM*. Modal models are based on hierarchical state machines, various versions of which have been studied in the literature or are available as commercial products. The latter include: 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 number of different semantics have been proposed for Statecharts – see, for instance, Beeck (1994) and 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 is based on the 'run-to-completion' principle, which is not really synchronous, though 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) and Hamon (2005), and timed versions of Statecharts and UML have been proposed in Damm *et al.* (1998) and Graf *et al.* (2006). In Ptolemy modal models, the hierarchy is not restricted to contain just state machines or concurrent state machines (built with AND states). Also, unlike Statecharts, SyncCharts and Stateflow, Ptolemy modal models do not use *broadcast* events for communication, but use *ports*, as in the block-diagram based notation of Ptolemy.

2.2. *Other modelling frameworks*

A number of other modelling frameworks provide mechanisms for mixing MoCs:

— An early systematic approach to such mixed models was realised in Ptolemy Classic (Buck *et al.* 1994).
— The Metropolis (Balarin *et al.* 2003; Goessler and Sangiovanni-Vincentelli 2002) environment focuses on modelling both function and architecture, and on mapping the former to the latter. Metropolis includes the concepts of constraints and quantity managers, which are used to constrain the behaviours of a model and annotate them with quantities such as time and energy, or other metrics.
— The Generic Modeling Environment (GME) (Karsai 1995; Nordstrom *et al.* 1999; Ledeczi *et al.* 2001) uses metamodelling techniques to create domain-specific modelling and program synthesis environments.
— BIP (Basu *et al.* 2006; Bliudze and Sifakis 2008a; Bozga *et al.* 2009) models are built by composing behavioural 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, the relative expressive power of two modelling formalisms with the 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 also been studied in the context of Reo: 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 the hierchical composition of MoCs, and emphasises the use of interface blocks that perform 'semantic adaptation' between heterogeneous models. For example, 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 Haskell functional programming language. ForSyDe includes different model transformations, which are used to refine an abstract specification model into a detailed implementation model, which can be translated into a target implementation language.
— SystemC can be used to realise multiple MoCs with a discrete-event simulation flavour (Patel and Shukla 2004; Herrera and Villar 2006).
— '42' (Maraninchi and Bhouhadiba 2007), which integrates an application model with a specification of a customised MoC, provides an interesting way of expressing the semantics of an MoC.
— Feredj *et al.* (2009) proposes a mechanism for creating domain-polymorphic components similar in spirit to Ptolemy.
— Bliudze and Krob (2009) and Aiguier *et al.* (2011) are also close in spirit to the current paper. Their 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, an actor) is captured as a kind of timed Turing machine, where non-standard analysis is used to represent continuous time through *infinitesimals*.
  – In Aiguier *et al.* (2011), a system is captured as a kind of timed Mealy machine. As in a Mealy machine, the interface of these machines contains two functions, an output function and a state-update function, but in this case there is an additional input argument to both functions. Composition in this framework is achieved by three operators: parallel composition, feedback and abstraction. Therefore, in Aiguier *et al.*'s approach, the different MoCs are not realised by the composition operators, unlike the case with Ptolemy.
— Arguably the work most closely related to the current paper is that of Denckla and Mosterman (2008), which presents 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 the 'next state', while for continuous systems, it is interpreted as the time derivative of the state. A solver is then used to transform continuous systems into discrete systems. The state-based semantics of Denckla and Mosterman (2008) is closely related to the semantics we present in the current paper, though 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 manipulate time explicitly (for example, to specify a deadline).

When comparing the above frameworks with each other and with Ptolemy, it is difficult to come up with a precise statement of their relative strengths and weaknesses. This is partly because they each pursue slightly different goals, ranging from 'pure' modelling and simulation, through verification, to design-space exploration, mapping and implementation. Ptolemy focuses on modelling and simulation, and leverages external tools (for example, model-checkers) and code-generators for other tasks (for example, verification).

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

Another, and possibly more fundamental, issue is that modelling and design are ultimately creative tasks and thus judgements about them are inherently subjective, with contributions from human taste, experience and other factors. For example, we might want to know which of the above approaches produces designs that are easier to build or more intuitive to understand? This is at least as difficult to answer as the question asking what kind of programs are easier to write in each of the existing programming languages? In principle, with clever encodings, most designs could be captured in any of the frameworks listed above, and even in homogeneous modelling frameworks. The question is how much effort is required to do so, and then to understand the result, modify it when necessary, use it for analysis or implementation, and so on. Ptolemy strives to offer a framework that is as general as possible (integrating many MoCs), but at the same time as intuitive as possible so that an individual model written in, say, SDF or CT, behaves in Ptolemy in a way that someone familiar with SDF or CT would expect it to behave.

Finally, a number of component-oriented frameworks have been developed in the fields of traditional programming and software engineering, for example, 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. A common characteristic that these frameworks share with ours is that they provide notions of standardised interfaces, starting at the level of notation, as with UML-based frameworks, up to the level of execution, as with concrete implementations of frameworks like Fractal. However, these frameworks have

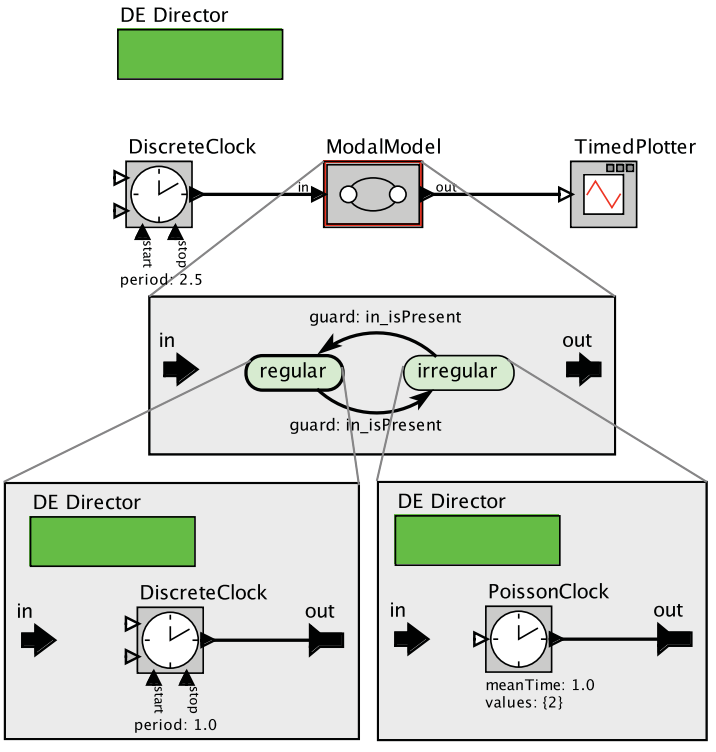

Fig. 3. (Colour online) A Ptolemy model.

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 behavioural semantics of many different MoCs.

3. Ptolemy's graphical syntax

Before presenting the formal semantics later in the paper, in this section we will use the example shown in Figure 3 to give a brief overview of Ptolemy's graphical syntax. There are 9 actors in this model:

(1) The top-level actor.
This is a composite actor, which is composed with the DE director. This top-level actor contains three sub-actors.
(2) The `DiscreteClock` actor.
This has a period 2.5 and is embedded in the top-level actor.
(3) The `TimedPlotter` actor.
This is also embedded in the top-level actor.
(4) The `ModalModel` actor.
This is also embedded in the top-level actor, and is itself a composite actor, which is composed with the MM director.

(5) The controller automaton of the `ModalModel` actor.
    This has two locations, `regular` and `irregular`[†].
(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.
    This has a period of 1.0 and is embedded in the composite actor refining `regular`.
(9) The `PoissonClock` actor.
    This is embedded in the composite actor refining `irregular`.

Of these 9 actors, 4 are composite and 5 are atomic[‡]. Each composite actor is associated with an actor diagram, so the model also contains 4 actor diagrams.

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

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 values do 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 identified by a bold outline: for example, 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, and actions are written in the Ptolemy action language. We will omit a detailed formal definition of guards and actions since they are standard in most modelling languages.

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

[†] We use the term *location* for state machines and modal models rather than *state* to distinguish it from the semantical concept of state, which we will define formally in Section 4.

[‡] 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 unnecessarily complex.

Stated briefly, the behaviour 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 behaviour 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 $\mathscr{U}$. A *valuation* (or *assignment*) over S is a function $x : S \to \mathscr{U}$ that assigns to each variable $v \in S$ some value $x(v) \in \mathscr{U}$. The set of all assignments over S is denoted by $\widehat{S}$. 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 $x \restriction_{S'}$ 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 $\alpha \in \mathscr{U}$, 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 $\alpha_1, \alpha_2 \in \mathscr{U}$, 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 variable called a *timer*. 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 $\mathscr{U}$ are $\perp$, 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 will 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), while in the latter case, the signal has the 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 will 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*[†], 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:

$$F : \widehat{S} \times \widehat{I} \to \widehat{O} \tag{2}$$

$$P : \widehat{S} \times \widehat{I} \to \widehat{S} \tag{3}$$

$$D : \widehat{S} \times \widehat{I} \to \mathbb{R}_+^\infty \tag{4}$$

$$T : \widehat{S} \times \widehat{I} \times \mathbb{R}_+ \to \widehat{S} \tag{5}$$

We assume that I, O, S are pair-wise disjoint, that is, $I \cap O = I \cap S = O \cap S = \varnothing$. We use the terms *input*, *output*, *state* to mean valuations over I, O, S, respectively. For example, $x : I \to \mathscr{U}$ is an input, $y : O \to \mathscr{U}$ is an output and $s : S \to \mathscr{U}$ 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, that is, a set with a single element that we will denote by $*$. Even if all its sets of variables are finite, an actor need not be *finite-state*, since its *state space*, that is, $\widehat{S}$, 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 for modelling the semantics of *timed* actors. Their role should become clear when we explain timed behaviours below.

4.3. *Actor behaviours*

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

$$s_0 \xrightarrow{x_0/y_0} s'_0 \xrightarrow{x'_0/d_0} s_1 \xrightarrow{x_1/y_1} s'_1 \xrightarrow{x'_1/d_1} s_2 \xrightarrow{x_2/y_2} s'_2 \xrightarrow{x'_2/d_2} \cdots \tag{6}$$

[†] Input and output variables are called *ports* in Ptolemy.

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

$$y_i = F(s_i, x_i) \tag{7}$$
$$s_i' = P(s_i, x_i) \tag{8}$$
$$d_i \leqslant D(s_i', x_i') \tag{9}$$
$$s_{i+1} = T(s_i', x_i', d_i). \tag{10}$$

The intuition is as follows. Suppose that at some point in time, say $t \in \mathbb{R}_+$, we have A is at state s_i. The environment provides input x_i to A, and A instantaneously produces output y_i using its F 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 has 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 behaviours, 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, for example, 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. However, if required, we can model non-deterministic actors as deterministic actors with extra input variables.

4.4. *Actor classification and special cases*

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

— A is called a *source* if it has no input variables, that is, $I = \varnothing$.
— A is called a *sink* if it has no output variables, that is, $O = \varnothing$.
— A is said to be *stateless* if it has no state variables, that is, $S = \varnothing$.
— A is said to be *untimed* if D is a constant function that always returns infinity, that is, for any state s and input x, $D(s, x) = \infty$. Otherwise A is said to be *timed*.
— A is said to be *delay-independent* if T leaves the state unchanged, that is, for any state s, input x, and delay d, $T(s, x, d) = s$. Otherwise A is said to be *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 is simplified so we can simplify our notation accordingly. For instance, if A is a source, then $\widehat{I}$ is a singleton and 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 can similarly simplify the 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, the functions P and T are trivial: they are the constant functions that return the unique element $*$, so there is no need to specify them.

4.5. *Examples of atomic actors*

4.5.1. *Constant.* Intuitively, the Constant actor parameterised by some value $\alpha \in \mathscr{U}$ produces the constant value α every time it is fired. This actor can be modelled as follows:

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

where

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

F returns the valuation $y \in \widehat{\{o\}}$ that assigns α to the unique output variable o.

Const_α is a source, and 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 modelled as follows:

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

where

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

for any input x.

F returns the valuation $y \in \widehat{\{o\}}$ that assigns to the unique output variable o the value $x(v)$ of the unique input variable v at the input valuation x.

Id is stateless, untimed and delay-independent.

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

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

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 $y \in \widehat{\{o\}}$ that 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 (that is, $\bot$) 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.* Intuitively, the And actor produces a value that corresponds to the logical-and (conjunction) of its inputs every time it is fired. This actor can be modelled as follows:

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

where

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

$$D(x) = \infty \tag{21}$$

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.* Intuitively, the Memory actor† parameterised by some initial value $\alpha \in \mathscr{U}$ 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 modelled as follows:

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

where

$$s_0 = \{m \mapsto \alpha\} \tag{23}$$

$$F(s, x) = \{o \mapsto s(m)\} \tag{24}$$

$$P(s, x) = \{m \mapsto x(v)\} \tag{25}$$

$$D(s, x) = \infty \tag{26}$$

$$T(s, x, d) = s \tag{27}$$

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

Mem_α is untimed and delay-independent.

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

4.5.6. *Discrete Clock.* Intuitively, the Discrete Clock actor parameterised by some value $\alpha \in \mathscr{U}$ and some period $\pi \in \mathbb{R}_+$ produces an event with value α every π time units. This actor can be modelled as follows:

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

where

$$s_0 = \{c \mapsto 0\} \tag{29}$$

$$F(s) = \begin{cases} \{o \mapsto \alpha\} & \text{if } s(c) = 0 \\ \{o \mapsto \mathsf{absent}\} & \text{otherwise} \end{cases} \tag{30}$$

$$P(s) = \begin{cases} \{c \mapsto \pi\} & \text{if } s(c) = 0 \\ \{c \mapsto s(c)\} & \text{otherwise} \end{cases} \tag{31}$$

$$D(s) = s(c) \tag{32}$$

$$T(s,d) = \{c \mapsto \big(s(c) - d\big)\} \tag{33}$$

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

$\mathsf{Clk}_{\alpha,\pi}$ 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, its value is left unchanged, as denoted by the mapping $c \mapsto s(c)$. In the above formalisation, the timer is initialised to 0, which means it is initially expired. An alternative could be to initialise 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 that 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.

Note that $d \leqslant D(s)$ implies $d \leqslant s(c)$, so the new value of the timer, $s(c) - d$, is guaranteed to be non-negative. Also note that the condition $d \leqslant D(s)$ is ensured by the rules defining actor behaviours (Section 4.3).

4.5.7. *Constant Delay.* Intuitively, the Constant Delay actor parameterised by a delay $\Delta \in \mathbb{R}_+$ delays each input event by Δ time units. This actor can be modelled 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 $(\alpha, d) \in \mathscr{U} \times [0, \Delta]$. 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:

$$s_0 = \{\mathsf{Active} \mapsto []\} \tag{35}$$

$$F(s,x) = \begin{cases} \{o \mapsto \alpha\} & \text{if } s(\mathsf{Active}) \neq [] \text{ and } \mathsf{head}(s(\mathsf{Active})) = (\alpha, 0) \\ \{o \mapsto \mathsf{absent}\} & \text{otherwise} \end{cases} \tag{36}$$

$$P(s,x) = \begin{cases} \{\mathsf{Active} \mapsto (A' \cdot (\alpha, \Delta))\} & \text{if } x(v) = \alpha \text{ and } \alpha \neq \mathsf{absent} \\ \{\mathsf{Active} \mapsto A'\} & \text{otherwise} \end{cases} \tag{37}$$

$$A' = \begin{cases} \mathsf{tail}(s(\mathsf{Active})) & \text{if } s(\mathsf{Active}) \neq [] \text{ and } \mathsf{head}(s(\mathsf{Active})) = (\alpha, 0) \\ s(\mathsf{Active}) & \text{otherwise} \end{cases} \tag{38}$$

$$D(s,x) = \begin{cases} d & \text{if } s(\mathsf{Active}) \neq [] \text{ and } \mathsf{head}(s(\mathsf{Active})) = (\alpha, d) \\ \infty & \text{otherwise} \end{cases} \tag{39}$$

$$T(s,x,d) = \{\mathsf{Active} \mapsto (s(\mathsf{Active}) \ominus d)\} \tag{40}$$

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

The intuition is as follows. s_0 initialises 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 parameterised 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 the sinusoid generated, which is updated by the time-update function. The signal generated follows the function $\alpha \sin(\omega r + \phi)$.

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

where

$$s_0 = \{r \mapsto 0\} \tag{42}$$

$$F(s) = \{o \mapsto \alpha \cdot \sin(\omega \cdot s(r) + \phi)\} \tag{43}$$

$$P(s) = s \tag{44}$$

$$D(s) = \infty \tag{45}$$

$$T(s,d) = \{r \mapsto s(r) + d\} \tag{46}$$

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 $\mathsf{Integrator}_\alpha$ parameterised by an 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 just 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 and at the same time eliminate 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 can illustrate how data actors can be modelled in our framework by first considering two *static* dataflow (also called synchronous dataflow or SDF) actors (Lee and Messerschmitt 1987).

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

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

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. Hence, the F and D functions of Up_n can be defined as follows[†]:

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

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

for any input x.

Up_n is stateless, untimed and delay-independent.

We can similarly define the Downsampling actor parameterised 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 modelled as follows:

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

[†] 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

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

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

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

for any input x.

Down_n is stateless, untimed and delay-independent.

4.5.11. *Switch and Select.* Up_n and Down_n are SDF actors. In this section we will describe Switch and Select, which are *dynamic* dataflow actors in the sense that the number of tokens they 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 modelled as follows:

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

where

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

$$D(x) = \infty \tag{55}$$

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 modelled as follows:

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

where

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

$$D(x) = \infty$$

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 these actors' input queues.

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 modelled as actors. ESMs are useful for capturing various types of behaviour, and 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 (that is, dynamic parameters) of the ESM, plus a variable ranging over the set of locations of the ESM. s_0 initialises 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[†] 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 chosen transition's output and set actions, respectively. 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 will not formalise this intuitive semantics for ESMs here since it is 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 to form composite actors. For most composite actors, actor diagrams follow a block-diagram notation. That is, they are formed by instantiating actors and connecting the output ports of one actor to the input ports of another actor. The top-level diagram of the model of Figure 3 follows this block-diagram notation. Another example is provided by 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 formalised in our framework.

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

5.1. *Block diagrams*

We will formalise 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, that is, for $i, j \in \{1, ..., n\}$, if $i \neq j$, then $I_i \cap I_j = \varnothing$.
— 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 $v \in I_i \cap O_j$, this means that the output variable v of actor A_j is connected to the input variable with the same name in actor A_i.

Where an output variable o of some actor is connected to the input variables $v_1, v_2, ...$ of multiple actors, the *fan-out* is modelled 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 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 formalised 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, which are denoted by $\ell_1, ..., \ell_n$. 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 formalise directors as *composition operators*. Specifically, a director takes as its 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, and can thus be used in further compositions.

Most of the rest of the paper is taken up in defining the directors for various Ptolemy domains.

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 modelling 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*. The *SR composite actor* is then 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, that is, 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[†].

We will now 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

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

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

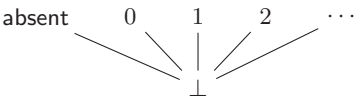

Fig. 4. The flat CPO used to ensure the existence of a unique least fixpoint that defines the semantics of SR, DE and CT.

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

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

Recall that $(x, y)\ \restriction_{I_j}$ 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, that is, F_j. This function takes as its inputs the local state s_j of A_j and the local input of A_j, which is precisely $(x, y)\ \restriction_{I_j}$.

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 by the following:

(1) 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 $\mathscr{U}$ being greater than $\perp$ in the CPO order: this CPO is illustrated in Figure 4.
(2) The fact that monotonicity implies continuity for flat CPOs.
(3) 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, $x^* \leqslant x$ with respect to the order $\leqslant$ of the CPO. Let $y^*_{s,x}$ 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

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

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

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 $\lim_{n\to\infty} \widetilde{F}^n_{s,x}(\perp)$, where

$$\widetilde{F}^0_{s,x}(\perp) = \perp$$
$$\widetilde{F}^{n+1}_{s,x}(a) = \widetilde{F}_{s,x}(\widetilde{F}^n_{s,x}(a)) \text{ for all } a.$$

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 once: from $\perp$ to some value other than $\perp$. Since the number of variables is finite, the total number of possible changes is also finite. 'Cleverer' methods for computing the fixpoint have been studied, for instance, in Edwards and Lee (2003). In the special case where the

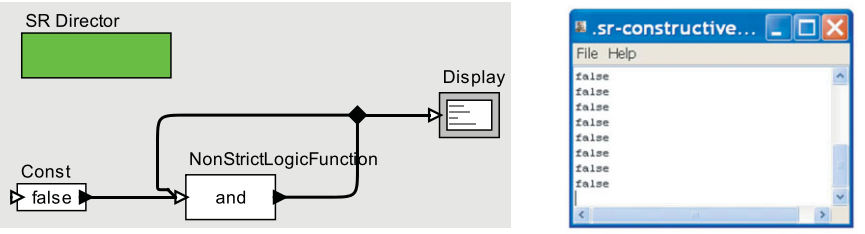

Fig. 5. (Colour online) An SR model (left) and its output (right).

block diagram H is acyclic (that is, 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 be variables to which $y^*_{s,x}$ assigns either absent or $\perp$. As mentioned earlier, absent is a perfectly legal value, just like other values (booleans, integers, and so on), but they are especially useful in defining actors in the DE domain. On the other hand, a fixpoint that contains $\perp$ values would normally indicate an erroneous model whose semantics are ambiguous – an example will be discussed later.

We still need to define functions D and T. There are various options for defining D. One option is to treat SR(H) as an untimed actor, in which case

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

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

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

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 that the state is left unchanged:

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

Note that SR(H) is delay-independent by definition.

A simple Ptolemy model that uses the SR director is shown in Figure 5. This model contains four actors: Const_{false}, And, Display and Fanout. The Fanout actor is denoted by the small black rhombus. The Display actor merely serves to output the results of the Ptolemy simulation on the screen and does not influence the behaviour 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). This is indeed an ambiguous model, which could be viewed 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 the constructive semantics of synchronous languages like Esterel.

6.2. *Discrete event (DE)*

The notion of time in SR is not quantitative in the sense that synchronous reactions are simply ordered as a sequence, and do not necessarily correspond to any 'real' notion of time (for example, in an implementation, the time that elapses between successive reactions may vary dynamically). The DE domain aims to capture 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 behaviour, 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 $d \in \mathbb{R}_+$.

Figure 6 shows a simple Ptolemy model that uses the DE director. This model contains four actors: $\mathsf{Clk}_{0.5,0.6}$, $\mathsf{Clk}_{0.3,1}$, Add and TimedPlotter[†]. The TimedPlotter actor just serves to display 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 on the right of Figure 6. Note 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 (for example, 0.6, 1, 1.2, and so on), only one clock produces an event, which the Add actor reproduces at its output.

[†] Clock actors in Ptolemy have inputs, but we will ignore them for the purposes of this paper.

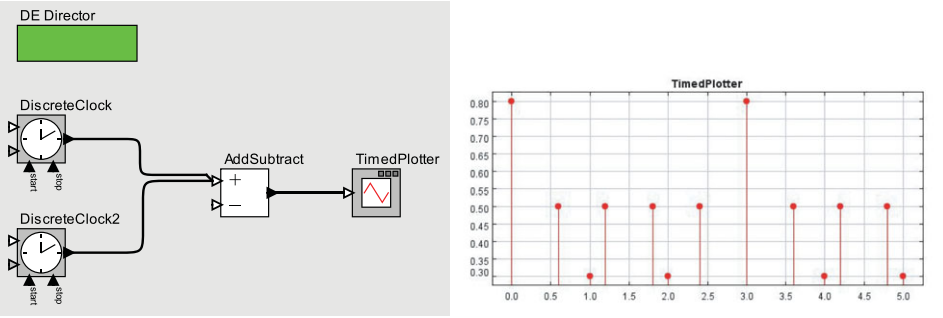

Fig. 6. (Colour online) A DE model (left) and its output (right).

6.3. *Continuous time (CT)*

The CT domain is used for modelling and simulating a broad class of continuous-time systems. This includes systems modelled 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 property that for any actor A_i that has an output that is connected to the input of an integrator, A_i must only produce numerical values. In particular, it must never produce absent on that output. This requirement is intended to characterise a 'continuous' signal in the essentially discrete computation framework underlying 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'.

The *CT composite actor* with parameters *initStepSize*, *maxStepSize* and *errorTolerance* is then 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 $\mathsf{SR}(H)$, and S and s_0 are defined by

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

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

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

Functions F and P of the composite CT actor do not use or 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, these 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, that is, $\mathsf{Ix} = \{i \mid A_i \text{ is an integrator}\}$.
— $I_\int$ denotes the set of all input variables of all integrators in H, that is, $I_\int = \bigcup_{i \in \mathsf{Ix}} I_i$.
— The state variable of an integrator actor A_i is denoted m_i and its input variable is denoted v_i.
— DE.D and DE.T refer, respectively, to the D and T functions of $\mathsf{DE}(H)$, so DE.$D(s, x)$ is the value obtained by evaluating the right-hand side of (69).
— Assignment is denoted by := 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;

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);
  $s' := \text{DE}.T(s, x, d)$;
  for $i \in \mathsf{Ix}$ **do**
    $s' := \{s' \mid m_i \mapsto P_i(s \restriction_{\{m_i\}}, z \restriction_{\{v_i\}})\}$;
    stepPrediction$(i) := d \cdot \left(\text{errorTolerance}/\text{LTEs(i)}\right)^{1/3}$;
  end for
  $s'(\text{stepSize}) := \min\{\text{maxStepSize}, \text{stepPrediction}(i) \mid i \in \mathsf{Ix}\}$;
  return s';

Roughly speaking, the Runge–Kutta procedure 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 the function $\tilde{F}$ 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, that is, 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 then 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.

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

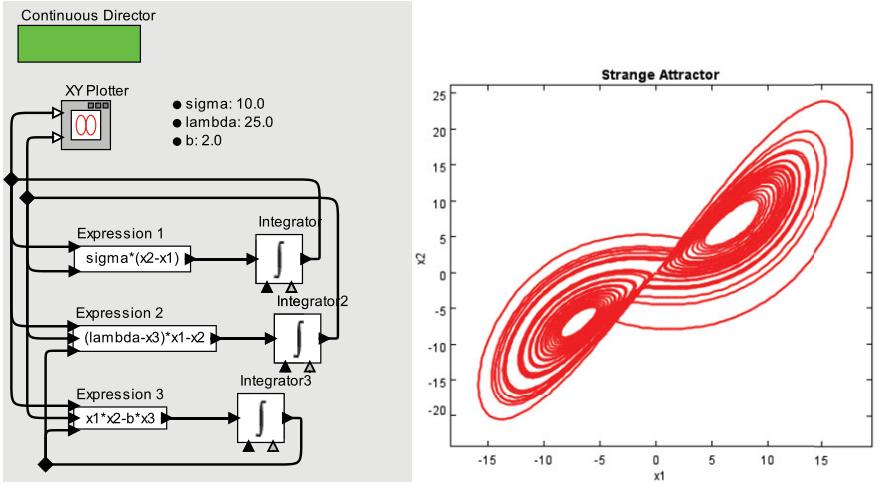

Fig. 7. (Colour online) A CT model (left) and its output (right).

Figure 7 shows an example of a CT Ptolemy model. This model captures a non-linear feedback system exhibiting chaotic behaviour, known as a Lorenz attractor. The model contains three $\mathsf{Integrator}_{\alpha}$ actors†, 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 on the right of the figure.

Figure 8 shows a model that mixes the CT and DE domains. The top-level actor of this model is a composite DE actor modelling 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 can be seen from the plot, the plant converges to that state after about 3 time units.

Most of the actors used in the model of Figure 8 were formally defined in Section 4. The `Scale` actors simply multiply their input by a constant number. The `ZeroOrderHold` actor transforms 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 the only difference being the way absent inputs are handled: in Mem, absent is handled like any other value, that is, it can be stored in the memory and produced as output, but in `ZeroOrderHold`, only non-absent values can be stored in memory. Thus, the postfire

† As with Clocks, Integrators in Ptolemy have additional inputs, but we will ignore for the purposes of this paper.

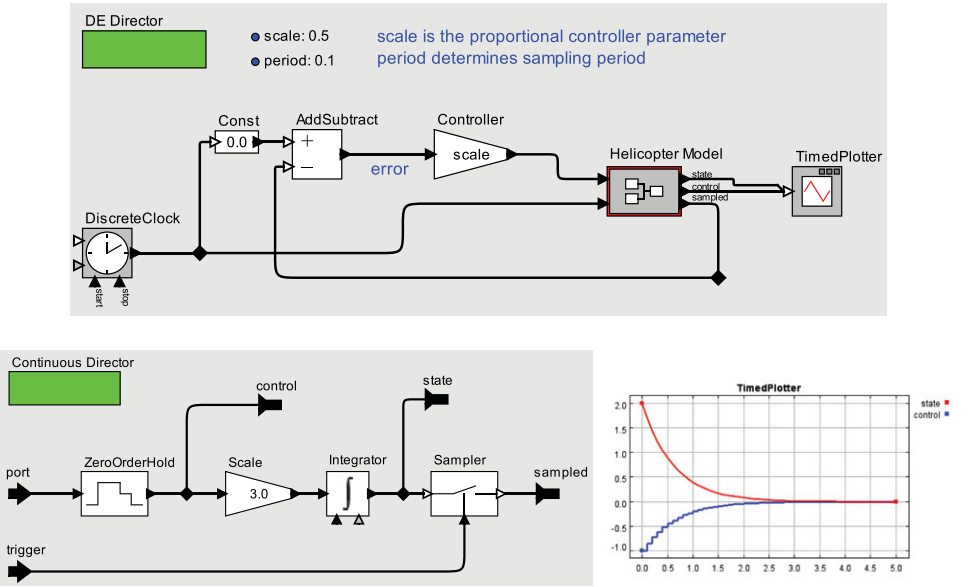

Fig. 8. (Colour online) A model mixing the DE and CT domains: top-level model (top), internals of the Helicopter Model composite actor (bottom-left) and output of the model (bottom-right).

function of `ZeroOrderHold` is as follows (contrast this with the definition in equation (25)):

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

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

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. However, in the Kahn PN model, which we follow here, the semantics is independent of the interleaving, which is a desirable feature as it allows 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 $i = 1, ..., n$. We can compose H 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. The *PN composite actor* is then 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 $\mathsf{SR}(H)$.

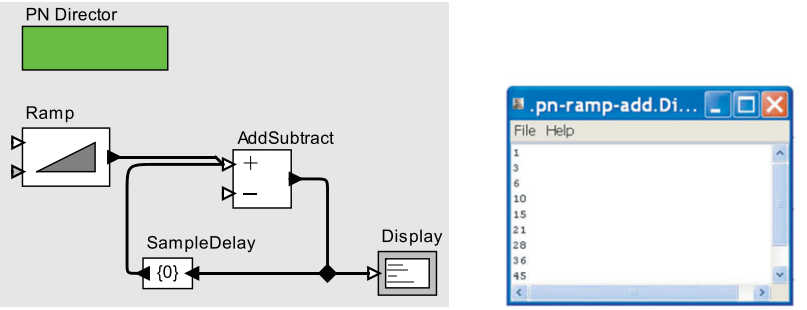

Fig. 9. (Colour online) A PN model (left) and its output (right).

The semantics for PN follows the semantics of Kahn Process Networks (Kahn 1974). Specifically, F is defined using a fixpoint semantics similar to the SR case, but with the major difference that unlike in the SR case, where the CPO is the flat CPO shown in Figure 4, 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' if and only if there exists stream ρ'' such that $\rho' = \rho \cdot \rho''$, where $\cdot$ denotes stream concatenation. Note 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 [] (that is, 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. Hence, 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 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 when embedding it into timed models. The definition of T for PN is also as in SR: 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 $[1, 2, \cdots]$ and `SampleDelay` produces as its 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†.

† 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, and so on) as well as streams. We do not provide a formal treatment of actor polymorphism since this would involve a more or less complete type theory for actors, which is beyond the scope of this paper.

6.5. *Modal models (MM)*

Modal models are hierarchical models for which 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 behaviour, where the system's operation changes dynamically by switching among a finite set of modes. Such changes may be triggered by user inputs, sensor data, hardware failures or other types of events, and are essential in fault management, adaptivity and reconfigurability (see, for instance, Sztipanovits *et al.* (1993) and Simon *et al.* (2001)). A modal model is an explicit representation of this type of behaviours, and the rules that govern transitions between behaviours.

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

$$A_c = (I_c, O_c, S_c, s_{0,c}, F_c, P_c, D_c, T_c)$$
$$A_i = (I_i, O_i, S_i, s_{0,i}, F_i, P_i, D_i, T_i) \quad \text{for } i = 1, ..., n.$$

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 the 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 can 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

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

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;
— preemptive(i, j):
preemptive transition taken from ℓ_i to ℓ_j;

— nonpreemptive(i, j):
non-preemptive transition taken from ℓ_i to ℓ_j.

We will now 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 (that is, $\perp$). For example, if x is an input variable and $x = \perp$, then the guard $x \geqslant 0$ 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. We 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. We 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 (2)b.
(c) s'_j is obtained as in Case (2)c.
(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 (that is, 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 formally define 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. In particular, this is done 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 modelling hybrid systems (Manna and Pnueli 1992), the state variables in all refinements are typically the same, that is, 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 and 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.* We have now completed 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 $\perp$. This means that the behaviour 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 an SR model. The behaviour 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 the logical connectives are modelled

as 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 according to the nature of the predicate. Equality, for instance, is usually interpreted strictly so that if either operand is bottom, the predicate evaluates to logical bottom. As a consequence of these semantics for the evaluation of guards, given a particular set of inputs, each guard can be evaluated to *true*, *false* or $\perp$. Given the implicit prioritisation 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, the enabledness status of any non-preemptive transitions is irrelevant[†]. 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 for which the guards are $\perp$, 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 described in Berry (1996), the presence of a signal cannot be concluded unless it is made present by a totally defined behaviour, but if, amongst all the 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 for which the guards evaluate non-strictly to *true* or $\perp$ (not *false*). Also, let $\mathsf{must}(X)$ be the subset of transitions in X for which the guards 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.

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

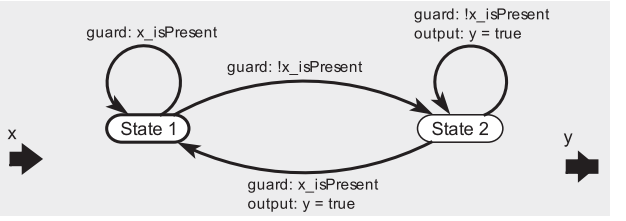

Fig. 10. A model that under the non-strict interpretation, given unknown input $x = \bot$, concludes that output $y = \mathsf{absent}$ from *State 1*, but $y = \bot$ from *State 2*.

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

if	$\mathsf{must}(T_P) = \{t_p\}$	$\mathsf{take}(t_p)$
else if	$\mathsf{can}(T_P) = \varnothing \wedge \mathsf{must}(T_N) = \{t_n\}$	$\mathsf{take}(t_n)$
else if	$\mathsf{can}(T_P) = \mathsf{can}(T_N) = \varnothing$	$\mathsf{take}(\epsilon)$
else		$\mathsf{check}(\mathsf{can}(T_P) \cup \mathsf{can}(T_N) \cup \{\epsilon\})$

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 would involve a static analysis of the action language of Ptolemy, the details of which have been omitted. Instead, we will just illustrate the behaviour of check using the following example.

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 transitions 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,

$$\mathsf{can}(T_P) = \mathsf{must}(T_P) = \mathsf{must}(T_N) = \varnothing,$$

and $\mathsf{can}(T_N)$ contains both transitions. This means that the output y must be determined by the function check applied to these two transitions. Assuming that the refinements of both locations are empty, that is, 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.

The situation is similar at *State 2*. Again,

$$\mathsf{can}(T_P) = \mathsf{must}(T_P) = \mathsf{must}(T_N) = \varnothing,$$

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 $y = \bot$, which implies that this model has ambiguous semantics. This model exemplifies

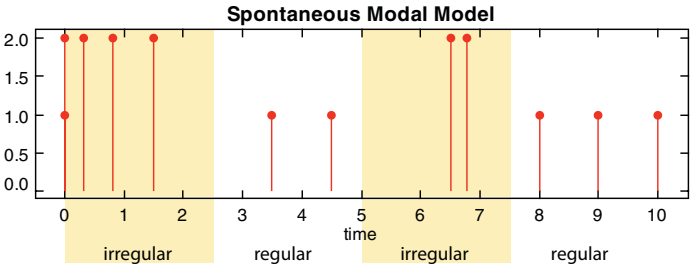

Fig. 11. (Colour online) A plot of the output from one run of the model in Figure 3.

the biasing of output values to absent, in accordance with a constructive interpretation such as that of Berry (1996).

6.5.3. *Deadline and time-update functions for modal models.* We will 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 the following cases:

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

$$D(s, x) = D_i(s_i, x) \tag{84}$$

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

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

$$D(s, x) = D_j(s_j, x) \tag{86}$$

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

(3) $\alpha = \mathsf{nonpreemptive}(i, j)$.
Then D and T are again 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 can 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 through 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 the value set to 2. The result of a typical run is plotted in Figure 11, with the background shading showing the times when it is in each of the two modes. A variant of the same

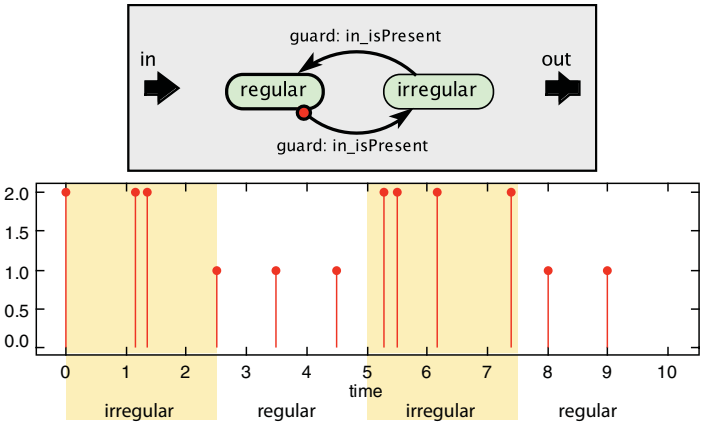

Fig. 12. (Colour online) A variant of Figure 3 where a preemptive transition prevents the initial firing of the innermost `DiscreteClock` actor of that model.

model is shown in Figure 12, the difference being that the transition from `regular` to `irregular` is now preemptive. These plots prompt a number of observations.

First note that two events are generated at time 0 in Figure 11: one 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 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 initialised, 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` (that is, of `PoissonClock`) is used to determine the deadline of the modal model. 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 in 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 (3)d). 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, that is, 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, so its timer has value 0 at time 2.5. This explains the event at that time.

7. Conclusions and future work

In this paper we have proposed a formal semantics for the heterogeneous modelling environment Ptolemy. The semantics is modular in the sense that it unifies atomic and composite actors within a single executable interface. Directors, which realise 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 viewed as 'black boxes'. This allows us to reason about a hierarchical model in a modular, actor-by-actor fashion, thus minimising the complexity that arises from cross-cutting dependencies between levels of the hierarchy.

An implementation of our framework in the Haskell functional programming language 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 modelled 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 direction for future work. Another direction is to examine the properties of the directors as composition operators, such as properties related to associativity and commutativity. We hope that this will lead to a better understanding of the properties of heterogeneous modelling in general. We also hope that this framework will serve as a starting point for a more in-depth discussion of how to unify different models of computation (for instance, what are the possible actor interfaces and what are their relative merits?) as well as how to formally represent them (for example, what would be the best language for writing directors?).

References

Aiguier, M., Golden, B. and Krob, D. (2011) Modeling of Complex Systems II: A minimalist and unified semantics for heterogeneous integrated systems. *Applied Mathematics and Computation.* **218** (16) 8039–8055.

Alur, R. *et al.* (1995) The algorithmic analysis of hybrid systems. *Theoretical Computer Science* **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 Ölveczky, P., Feng, T. H., Lee, E. A. and Tripakis, S. (2012) Verifying Hierarchical Ptolemy II Discrete-Event Models using Real-Time Maude. *Science of Computer Programming* **77** (12) 1235–1271.

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 06)*, IEEE Computer Society 3–12.

von der Beeck, M. (1994) A comparison of Statecharts variants. In: Langmaack, H., de Roever, W. P. and Vytopil, J. (eds.) 3rd International Symposium Formal Techniques in Real-Time and Fault-Tolerant Systems. *Springer-Verlag Lecture Notes in Computer Science* **863** 128–148.

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: Majumdar, R. and Tabuada, P. (eds.) Hybrid Systems: Computation and Control – 12th International Conference, HSCC 2009. *Springer-Verlag Lecture Notes in Computer Science* **5469** 46–60.

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. *Fundamenta Informaticae* **91** 251–274.

Bliudze, S. and Sifakis, J. (2008a) The algebra of connectors – structuring interaction in BIP. *IEEE Transactions on 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. (eds.) CONCUR 2008. *Springer-Verlag Lecture Notes in Computer Science* **5201** 508–522.

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)*, IEEE Computer Society 153–162.

Bozga, M., Sfyrla, V. and Sifakis, J. (2009) Modeling synchronous systems in BIP. In: Chakraborty, S. and Halbwachs, N. (eds.) *EMSOFT '09: Proceedings of the 9th ACM International Conference on Embedded Software*, ACM 77–86.

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

Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V. and Stefani, J.-B. (2006) The FRACTAL component model and its support in Java. *Software: Practice and Experience* **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. *International 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: *ACSD '01 Proceedings of the Second International Conference on Application of Concurrency to System Design*, IEEE Computer Society 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)*, IEEE.

Damm, W., Josko, B., Hungar, H. and Pnueli, A. (1998) A Compositional Real-time Semantics of STATEMATE Designs. In: de Roever, W.-P., Langmaack, H. and Pnueli, A. (eds.) Compositionality: The Significant Difference – Revised Lectures, International Symposium, COMPOS'97. *Springer-Verlag Lecture Notes in Computer Science* **1536** 186–238.

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

Denckla, B. and Mosterman, P. (2008) Stream- and state-based semantics of hierarchy in block diagrams. In: *Proceedings 17th IFAC World Congress* 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. *et al.* (2003) Taming heterogeneity – the Ptolemy approach. *Proceedings of the IEEE* **91** (2) 127–144.

Eshuis, R. (2009) Reconciling statechart semantics. *Science of Computer Programming* **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 *Proceedings of Symposium on Applied Mathematics* **19**, AMS 19–32.

Geilen, M. and Basten, T. (2003) Requirements on the execution of Kahn process networks. In: Degano, P. (ed.) Programming Languages and Systems: 12th European Symposium on Programming, ESOP 2003. *Springer-Verlag Lecture Notes in Computer Science* **2618** 319–334.

Geilen, M. and Basten, T. (2004) Reactive process networks. In: *EMSOFT '04: Proceedings of the 4th ACM International Conference on Embedded Software*, ACM 137–146.

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: Sangiovanni-Vincentelli, A. and Sifakis, J. (eds.) Embedded Software: Proceedings Second International Conference, EMSOFT 2002. *Springer-Verlag Lecture Notes in Computer Science* **2491** 93–107.

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

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

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*, ACM 164–172.

Hamon, G. and Rushby, J. (2004) An operational semantics for Stateflow. In: Wermelinger, M. and Margaria-Steffen, T. (eds.) Fundamental Approaches to Software Engineering: 7th International Conference, FASE 2004. *Springer-Verlag Lecture Notes in Computer Science* **2984** 229–243.

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*, IEEE Computer Society 45–54.

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: *DAC '06: Proceedings of the 43rd annual Design Automation Conference*, 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: *Proceedings of the IFIP Congress 74*, North-Holland.

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

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

Ledeczi, A. *et al.* (2001) Composing domain-specific design environments. *IEEE Computer* **34** (11) 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: Petriu, D. C., Rouquette, N. and Haugen, ∅. (eds.) Model Driven Engineering Languages and Systems, 13th International Conference, MODELS 2010: Proceedings, Part II. *Springer-Verlag Lecture Notes in Computer Science* **6395** 273–287.

Lee, E. A. and Matsikoudis, E. (2009) The semantics of dataflow with firing. In: Huet, G., Plotkin, G., Lévy, J.-J. and Bertot, Y. (eds.) *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. (eds.) Hybrid Systems: Computation and Control – 8th International Workshop, HSCC 2005. *Springer-Verlag Lecture Notes in Computer Science* **3414** 25–53.

Lee, E. A. and Zheng, H. (2007) Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In: *EMSOFT '07: Proceedings of the 7th ACM and IEEE International Conference on Embedded Software*, ACM.

Liu, J. and Lee, E. A. (2003) On the causality of mixed-signal and hybrid models. In: Maler, O. and Pnueli, A. (eds.) Hybrid Systems: Computation and Control 6th International Workshop, HSCC 2003. *Springer-Verlag Lecture Notes in Computer Science* **2623** 328–342.

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: Baier, C. and Hermanns, H. (eds.) Concurrency Theory: 17th International Conference, CONCUR 2006. *Springer-Verlag Lecture Notes in Computer Science* **4137** 1–15.

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)*, ACM 78–89.

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

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

Maraninchi, F. and Bhouhadiba, T. (2007) 42: Programmable models of computation for a component-based approach to heterogeneous embedded systems. In: *GPCE '07: Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, ACM 53–62.

Maraninchi, F. and Rémond, Y. (2003) Mode-automata: a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming* **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 *ECBS'99: Proceedings of the 1999 IEEE Conference on Engineering of Computer-Based Systems*, IEEE Computer Society 68–74.

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: Main, M., Melton, A., Mislove, M. and Schmidt, D. (eds.) Mathematical Foundations of Programming Language Semantics: Proceedings 3rd Workshop, Tulane University, New Orleans. *Springer-Verlag Lecture Notes in Computer Science* **298** 331–343.

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: *EMSOFT '04: Proceedings of the 4th ACM International Conference on Embedded Software*, ACM 259–268.

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*, North-Holland 75–93.

Simon, G., Kovácsházy, T. and Péceli, G. (2001) Transient management in reconfigurable systems. In: Robertson, P., Shrobe, H. and Laddaga, R. (eds.) Self-Adaptive Software: First International

Workshop, IWSAS 2000: Revised Papers. *Springer-Verlag Lecture Notes in Computer Science* **1936** 90–98.

Sztipanovits, J., Wilkes, D., Karsai, G., Biegl, C. and Lynd, L. (1993) The multigraph and structural adaptivity. *IEEE Transactions on Signal Processing* **41** (8) 2695–2716.

Tripakis, S., Bui, D., Geilen, M., Rodiers, B. and Lee, E. A. (2010). Compositionality in Synchronous Data Flow: Modular Code Generation from Hierarchical SDF Graphs. 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. (ed.) Proceedings of the 4th International Conference on Concurrency Theory (CONCUR). *Springer-Verlag Lecture Notes in Computer Science* **715** 384–397.

Zhu, Y. *et al.* (2010) Mathematical equations as executable models of mechanical systems. In: *ICCPS '10: Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*, ACM 1–11.