Verifying Hierarchical Ptolemy II Discrete-Event Models using Real-Time Maude

Kyungmin Bae
Peter Csaba Olveczky
Thomas Huining Feng
Edward A. Lee
Stavros Tripakis

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-50
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-50.html

May 6, 2010

Verifying Hierarchical Ptolemy II Discrete-Event Models using Real-Time Maude

Kyungmin Bae
University of Illinois at Urbana-Champaign

Peter Csaba Ölveczky*
University of Oslo

Thomas Huining Feng, Edward A. Lee, and Stavros Tripakis
University of California, Berkeley

May 6, 2010

Abstract

This paper defines a real-time rewriting logic semantics for a significant subset of Ptolemy II discrete-event models. This is a challenging task, since such models combine a synchronous fixed-point semantics with hierarchical structure, explicit time, and a rich expression language. The code generation features of Ptolemy II have been leveraged to automatically synthesize a Real-Time Maude verification model from a Ptolemy II design model, and to integrate Real-Time Maude verification of the synthesized model into Ptolemy II. This enables a model-engineering process that combines the convenience of Ptolemy II DE modeling and simulation with formal verification in Real-Time Maude. We illustrate such formal verification of Ptolemy II models with three case studies.

1 Introduction

Model-based design (MBD) [1, 2, 3] emphasizes the construction of high-level models for system design. Useful models are executable, providing simulations of system functionality and/or performance during the design phases as a much less costly alternative to building prototypes and testing them. MBD typically raises the level of abstraction in system design in general, and for embedded software in particular, from low-level languages, such as C++ and Java, to high-level modeling formalisms where concepts like concurrency and time are first-class notions; this makes it feasible to design systems that would be hard or impossible to design using low-level methods. Ideally, models are translated (code generated) automatically to produce deployable software. Commercial examples of such modeling and code generation frameworks include Real-Time Workshop (from The MathWorks) and TargetLink (from dSpace), which generate code from Simulink models, LabVIEW Embedded from National Instruments, and SCADE from Esterel Technologies.

Many real-time *embedded* systems – in areas such as avionics, motor vehicles, and medical systems – are *safety-critical* systems, whose failures may cause great damage to persons and/or valuable assets. Models of such embedded systems should therefore be formally analyzed to prove safety properties or identify security vulnerabilities. Instead of requiring designers to develop models in some formal framework, a promising approach to formally verify design models is to add formal analysis capabilities to the intuitive, often graphical, *informal* modeling languages preferred by practitioners by: (i) providing a formal semantics for the informal language, (ii) leveraging the code generation features of the informal modeling framework to automatically translate an informal model to a formal model, and (iii) verifying the synthesized formal model.

For *real-time* systems, we believe that *real-time rewrite theories* [4] should be a suitable formalism in which to define the semantics of time-based modeling languages, for the following reasons:

*Corresponding author

- Real-time rewrite theories have a natural and "sound" model of timed behavior that makes them suitable as a semantic framework [4].
- The expressiveness and generality of real-time rewrite theories allow us to give a formal semantics to languages with advanced functions and data types, new communication models, arbitrary and unbounded data structures, program variables ranging over unbounded domains, and so on.
- The associated Real-Time Maude tool [5] provides a range of formal analysis capabilities, including simulation, reachability analysis, and linear temporal logic model checking. Despite the expressiveness of real-time rewriting, timed-bounded LTL properties are often decidable under mild conditions [6].

Real-time rewrite theories and Real-Time Maude have been used to define the formal semantics of – and to provide a simulator and model checker for – some real-time modeling languages, including: a timed extension of the Actor model [7], the Orc web services orchestration language [8], a language developed at DoCoMo laboratories for handset applications [9], a behavioral subset of the avionics standard AADL [10], the visual model transformation language e-Motions [11], and real-time model transformations in MOMENT2 [12].

Ptolemy II [13] is a well established open-source modeling and simulation tool used in industry. A major reason for its popularity is Ptolemy II's powerful yet intuitive graphical modeling language that allows a user to build hierarchical models that combine different models of computations. In this paper, we focus on *discrete-event* (DE) models; such models are are explicit about the timing behavior of systems, which is an essential feature for the high-level specification of embedded system applications [14, 15]. Discrete-event modeling is a time honored and widely used approach for system simulation [16]. More recently, it has been proposed as basis for synthesis of embedded real-time software [17]. The Ptolemy II DE models have a semantics rooted in the fixed-point semantics of synchronous languages [18].

Like many graphical modeling languages, Ptolemy II DE models lack at present formal verification capabilities. Furthermore, Ptolemy II DE models seem to fall outside the class of languages which can be given an automaton-based semantics, because: (i) certain constructs, such as noninterruptible timers, require the use of data structures (such as lists) of unbounded size; (ii) the variables used in, e.g., the transition systems in FSM actors range over infinite domains such as the integers; (iii) executing a synchronous step requires fixed-point computations; and (iv) Ptolemy II has a powerful expression language.

This paper defines a Real-Time Maude semantics for a significant subset of *hierarchical* Ptolemy II DE models. We have used Ptolemy II's code generation infrastructure to automatically synthesize a Real-Time Maude verification model from a Ptolemy II model, and have integrated Real-Time Maude verification into Ptolemy II, so that Ptolemy II models can be formally analyzed from within Ptolemy II. We also define useful generic temporal logic propositions for such models, so that a Ptolemy II user can easily define his/her temporal logic requirements without understanding Real-Time Maude or the formal representation of a Ptolemy II model. This integration of Ptolemy II and Real-Time Maude enables a model-engineering process that combines the convenience of Ptolemy II modeling with formal verification in Real-Time Maude. We illustrate such formal verification on three case studies.

Our work on formalizing Ptolemy II is the first attempt to define a Real-Time Maude semantics for *synchronous* real-time languages. Apart from the important result of endowing hierarchical Ptolemy II DE models with a formal semantics and formal verification capabilities, the main contribution of this work is to show how Real-Time Maude can define the formal semantics of synchronous real-time languages with fixed-point semantics and hierarchical structure.

This paper extends the conference paper [19], that first outlined the Real-Time Maude semantics for flat DE models without general Ptolemy expressions, and the workshop paper [20], that proposed the extension to hierarchical DE models, by: (i) providing much more detail about our semantics, (ii) explaining how general Ptolemy expressions are handled, and (iii) describing two additional case studies.

The paper is organized as follows. Sections 2 and 3 introduce Real-Time Maude and Ptolemy II DE models, respectively. In order to convey the main ideas of our formalization of Ptolemy II DE models without obscuring the presentation with too much detail, we present the semantics in three steps: Section 4 defines the Real-Time Maude semantics of *flat* Ptolemy II DE models where Ptolemy II expressions are constants; Section 5 extends that semantics to hierarchical DE models; and Section 6 extends it to general

Ptolemy II expressions. Section 7 briefly explains how Real-Time Maude verification has been integrated into Ptolemy II and also presents some useful predefined atomic propositions that allow users to easily specify desired system requirements. Section 8 illustrates Real-Time-Maude-based verification in Ptolemy II with three case studies. Section 9 discusses related work, and Section 10 gives some concluding remarks. More details about the Real-Time Maude semantics of Ptolemy are given in the longer technical report [21].

2 Real-Time Maude

Real-Time Maude [5] is a language and tool extending Maude [22] to support the formal specification and analysis of *real-time* systems. The specification formalism is based on *real-time rewrite theories* [4]—an extension of *rewriting logic* [23, 24]—and emphasizes *ease* and *generality* of specification.

Real-Time Maude specifications are *executable* under reasonable assumptions, so that a first form of formal analysis consists of simulating the system's progress in time by *timed rewriting*. This can be very useful for simulating the system, but any such execution gives us only *one* behavior among the many possible concurrent behaviors of the systems. To gain further assurance about a system one can use *model checking* techniques that explore many different behaviors from a given initial state of the system. Timed *search* and *linear temporal logic model checking* can analyze *all* possible behaviors from a given initial state (possibly up to a given duration).

2.1 Preliminaries: Object-Oriented Specification in Maude

Since Real-Time Maude specifications extend Maude specifications, we first recall object-oriented specification in Maude.

A *membership equational logic* (MEL) [25] *signature* is a triple $\Sigma = (K, \sigma, S)$, with K a set of *kinds*, $\sigma = \{\sigma_{w,k}\}_{(w,k)\in K^* \times K}$ a many-kinded algebraic signature, and $S = \{S_k\}_{k\in K}$ a K-kinded family of disjoint sets of sorts. The kind of a sort s is denoted by $[s]$. A MEL algebra A contains a set A_k for each kind k, a function $A_f : A_{k_1} \times \cdots \times A_{k_n} \to A_k$ for each operator $f \in \sigma_{k_1 \cdots k_n, k}$, and a subset $A_s \subseteq A_k$ for each sort $s \in S_k$. $T_{\Sigma,k}$ and $T_\Sigma(X)_k$ denote, respectively, the set of ground Σ-terms with kind k and of Σ-terms with kind over the set X of kinded variables.

A MEL *theory* is a pair (Σ, E), where Σ is a MEL-signature, and E is a set of conditional equations of the form $(\forall X)\ t = t'$ **if** $\bigwedge_i p_i = q_i \wedge \bigwedge_j w_j : s_j$ and conditional memberships of the form $(\forall X)\ t : s$ **if** $\bigwedge_i p_i = q_i \wedge \bigwedge_j w_j : s_j$, for $t, t' \in T_\Sigma(X)_k, s \in S_k$, the latter stating that t is a term of sort s, provided the condition holds. Order-sorted notation $s_1 < s_2$ can be used to abbreviate the conditional membership $(\forall x : [s_1])\ x : s_2$ **if** $x : s_1$. Similarly, an operator declaration $f : s_1 \times \cdots \times s_n \to s$ corresponds to declaring f at the kind level and giving the membership axiom $(\forall x_1 : k_1, \ldots, x_n : k_n)\ f(x_1, \ldots, x_n) : s$ **if** $\bigwedge_{1\le i\le n} x_i : s_i$.

The intuitive meaning is that terms in sorts are well-defined, while elements without sorts, such as $fact(-5)$ and $fact(3-1)$ in some signature defining the factorial function $fact$, are either *error* (or "undefined") values such as $fact(-5)$, or are expressions, such as $fact(3-1)$, that are not yet "computed," but that will evaluate to well-sorted terms when fully evaluated.

A Maude module specifies a *rewrite theory* [24, 23] of the form $(\Sigma, E \cup A, R)$, where $(\Sigma, E \cup A)$ is a membership equational logic theory with A a set of equational axioms such as associativity, commutativity, and identity, so that equational deduction is performed *modulo* the axioms A. The theory $(\Sigma, E \cup A)$ specifies the system's state space as an algebraic data type. R is a collection of *labeled conditional rewrite rules* specifying the system's local transitions, each of which has the form[1]

$$[l] : t \longrightarrow t' \ \textbf{if} \ \bigwedge_{j=1}^{m} u_j = v_j,$$

[1] In general, the condition of such rules may not only contain equations $u_j = v_j$, but also rewrites $w_i \longrightarrow w_i'$ and *memberships* $t_k : s_k$; however, the specification in this paper does not use this extra generality.

where l is a *label*. Intuitively, such a rule specifies a *one-step transition* from a substitution instance of t to the corresponding substitution instance of t', *provided* the condition holds; that is, the substitution instances of the equalities $u_j = v_j$ follow from $E \cup A$. The rules are implicitly universally quantified by the variables appearing in the Σ-terms t, t', u_j, and v_j. The rules are applied *modulo* the equations $E \cup A$.[2]

We briefly summarize the syntax of Maude. Operators are introduced with the `op` keyword: `op` f `:` $s_1 \ldots s_n$ `->` s. They can have user-definable syntax, with underbars '`_`' marking the argument positions, and are declared with the sorts of their arguments and the sort of their result. Some operators can have equational *attributes*, such as `assoc`, `comm`, and `id`, stating, for example, that the operator is associative and commutative and has a certain identity element. Such attributes are then used by the Maude engine to match terms *modulo* the declared axioms. An operator can also be declared to be a *constructor* (`ctor`) that defines the carrier of a sort. There are three kinds of logical statements, namely, *equations*—introduced with the keywords `eq`, or, for conditional equations, `ceq`—*memberships*—declaring that a term has a certain sort and introduced with the keywords `mb` and `cmb`—and *rewrite rules*—introduced with the keywords `rl` and `crl`. The mathematical variables in such statements are either explicitly declared with the keywords `var` and `vars`, or can be introduced on the fly in a statement without being declared previously, in which case they must be have the form var`:`$sort$. We will make frequent use of the fact that an equation $f(t_1, \ldots, t_n) = t$ with the `owise` (for "otherwise") attribute can be applied to a subterm $f(\ldots)$ only if no other equation with left-hand side $f(u_1, \ldots, u_n)$ can be applied.[3] Finally, a comment is preceded by '`***`' or '`---`' and lasts till the end of the line.

In Maude, kinds are not explicitly declared; instead the kind of a sort s is denoted `[`s`]`. Maude also supports the declaration of partial functions using the arrow '`~>`':

`op` f `:` s_1 `...` s_n `~>` s `.`

The above declaration is equivalent to the kinded declaration

`op` f `:` `[`s_1`]` `...` `[`s_n`]` `->` `[`s`]` `.`

In object-oriented Maude modules one can declare *classes* and *subclasses*. A class declaration

`class` C `|` att_1 `:` s_1`,` `...` `,` att_n `:` s_n

declares an object class C with attributes att_1 to att_n of sorts s_1 to s_n. An *object* of class C in a given state is represented as a term

`<` O `:` C `|` att_1 `:` val_1`,` `...`, att_n `:` val_n `>`

of the built-in sort `Object`, where O is the object's name or identifier, and where val_1 to val_n are the current values of the attributes att_1 to att_n and have sorts s_1 to s_n. Objects can interact with each other in a variety of ways, including the sending of messages. A message is a term of the built-in sort `Msg`, where the declaration

`msg` m `:` p_1 `...` p_n `->` `Msg`

defines the syntax of the message (m) and the sorts (p_1 ... p_n) of its parameters. In a concurrent object-oriented system, the state, which is usually called a *configuration*, is a term of the built-in sort `Configuration`. It has the structure of a *multiset* made up of objects and messages. Multiset union for configurations is denoted by a juxtaposition operator (empty syntax) that is declared associative and commutative and having the `none` multiset as its identity element, so that order and parentheses do not matter, and so that rewriting is *multiset rewriting* supported directly in Maude. The dynamic behavior of object systems is axiomatized by specifying each of its concurrent transition patterns by a rewrite rule. For example, the configuration fragment on the left-hand side of the rule

[2] Operationally, a term is reduced to its E-normal form modulo A before any rewrite rule is applied in Maude. Under the coherence assumption [26] this is a complete strategy to achieve the effect of rewriting in $E \cup A$-equivalence classes.

[3] A specification with `owise` equations can be transformed to an equivalent system without such equations [22].

```
rl [l] :  m(O,w)  < O : C | a1 : x, a2 : y, a3 : z >  =>
                  < O : C | a1 : x + w, a2 : y, a3 : z >  m'(y,x)
```

contains a message `m`, with parameters `O` and `w`, and an object `O` of class `C`. The message `m(O,w)` does not occur in the right-hand side of this rule, and can be considered to have been *removed* from the state by the rule. Likewise, the message `m'(y,x)` only occurs in the configuration on the right-hand side of the rule, and is thus *generated* by the rule. The above rule, therefore, defines a parametrized family of transitions (one for each substitution instance) in which a message `m(O,w)` is read and consumed by an object `O` of class `C`, with the effect of altering the attribute `a1` of the object and of sending a new message `m'(y,x)`. By convention, attributes, such as `a3` in our example, whose values do not change and do not affect the next state of other attributes need not be mentioned in a rule or an equation. Attributes like `a2` whose values influence the next state of other attributes or the values in messages, but are themselves unchanged, may be omitted from right-hand sides of rules/equations.

A *subclass* inherits all the attributes, equations, and rules of its superclasses[4], and multiple inheritance is supported.

2.2 Object-Oriented Specification in Real-Time Maude

A Real-Time Maude *timed module* specifies a *real-time rewrite theory* [4], that is, a rewrite theory $\mathcal{R} = (\Sigma, E \cup A, R)$, such that:

1. $(\Sigma, E \cup A)$ contains an equational subtheory $(\Sigma_{TIME}, E_{TIME}) \subseteq (\Sigma, E \cup A)$, satisfying the $TIME$ axioms in [4], which specifies a sort `Time` as the time domain (which may be discrete or dense). Although a timed module is parametric on the time domain, Real-Time Maude provides some predefined modules specifying useful time domains. For example, the modules `NAT-TIME-DOMAIN-WITH-INF` and `POSRAT-TIME-DOMAIN-WITH-INF` define the time domain to be, respectively, the natural numbers and the nonnegative rational numbers, and contain the subsort declarations `Nat < Time` and `PosRat < Time`. These modules also add a supersort `TimeInf`, which extends the sort `Time` with an "infinity" value `INF`.

2. The sort of the "states" of the system has the designated sort `System`.

3. The rules in R are decomposed into:

   - "ordinary" rewrite rules that model *instantaneous* change, and
   - *tick (rewrite) rules* that model the elapse of time in a system. Such tick rules have the form $l : \{t\} \longrightarrow \{t'\}$ **if** *cond*, where t and t' are of sort `System`, `{_}` is a built-in constructor of a new sort `GlobalSystem`, and where we have associated to such a rule a term u of sort `Time` denoting the *duration* of the rewrite. In Real-Time Maude, tick rules, together with their durations, are specified with the syntax

     `crl [`l`] : {`t`} => {`t'`} in time` u `if` *cond* .

The initial state of a real-time system so specified must be reducible to a term $\{t_0\}$, for t_0 a ground term of sort `System`, using the equations in the specification. The form of the tick rules then ensures uniform time elapse in all parts of a system.

2.3 Formal Analysis in Real-Time Maude

We summarize below the Real-Time Maude analysis commands. All Real-Time Maude analysis commands and their semantics are explained in [5].

Real-Time Maude's *timed fair rewrite* command simulates *one* behavior of the system *up to a certain duration*. It is written with syntax

[4]The attributes, equations, and rules of a class cannot be redefined by its subclasses, but subclasses may introduce additional attributes, equations, and rules.

`(tfrew` *t* `in time <=` *timeLimit* `.)`

where *t* is the term to be rewritten ("the initial state"), and *timeLimit* is a ground term of sort `Time`.

Real-Time Maude provides a variety of search and model checking commands for further analyzing timed modules by exploring *all* possible behaviors—up to a given number of rewrite steps, duration, or satisfaction of other conditions—that can be nondeterministically reached from the initial state. For example, Real-Time Maude extends Maude's *search* command—which uses a breadth-first strategy to search for states that are reachable from the initial state and match the *search pattern* and satisfy the *search condition*—to search for states that can be reached within a given time interval from the initial state.

Finally, Real-Time Maude extends Maude's *linear temporal logic model checker* to check whether each behavior (possibly "up to a certain time," as explained in [5]) satisfies a temporal logic formula. *State propositions*, possibly parametrized, should be declared as operators of sort `Prop`, and their semantics should be given by (possibly conditional) equations of the form

`{`*statePattern*`} |=` *prop* `=` *b*

for *b* a term of sort `Bool`, which defines the state proposition *prop* to hold in all states `{`*t*`}` such that `{`*t*`} |=` *prop* evaluates to `true`. A temporal logic *formula* is constructed by state and clocked[5] propositions and temporal logic operators such as `True`, `False`, `~` (negation), `/\`, `\/`, `->` (implication), `[]` ("always"), `<>` ("eventually"), `U` ("until"), and `W` ("weak until"). The command

`(mc` *t* `|=u` *formula* `.)`

is the model checking command which checks whether the temporal logic formula *formula* holds in all behaviors starting from the initial state *t*.

3 Ptolemy II and its DE Model of Computation

The Ptolemy project[6] studies modeling, simulation, and design of concurrent, real-time, embedded systems. Ptolemy II is a modeling environment that supports multiple modeling paradigms, which we call *models of computations* (MoCs), that govern the interaction between concurrent components.

Modeling with heterogeneous MoCs [13] is a key research area of the Ptolemy project. The supported MoCs include FSM (finite state machine), dataflow, and DE (discrete events). Such MoCs can compose to create heterogeneous models with well-defined semantics.

3.1 Discrete-Event Models

A Ptolemy II model consists of a set of *actors*, each having a set of *ports* that can be connected. An *output port* outputs data. An *input port* receives data from the output port that it is connected to.

A composition of actors, including the interconnections between their ports, can be encapsulated as an actor in its own right, which may also have input and output ports. Such an actor obtained by composition is called a *composite actor*. An input port of a composite actor can be connected to input ports of the actors inside, which means that external inputs are transferred to those inner actors. Similarly, an output port of a composite actor can be connected to output ports of the actors inside. An actor that is not composite is called an *atomic actor*.

The focus of this paper is the formalization of Ptolemy II *discrete-event* (DE) models. In DE, the data sent and received at actors' ports are *events*. Each event has two components: a *tag* and a *value*. According to the *tagged signal model* [27], a tag *t* is a pair $(\tau, n) \in \mathbb{R}_{\geq 0} \times \mathbb{N}$, where τ is the *timestamp* denoting the model time at which the event occurs, and *n* is the *microstep index*. Microstep indices are useful for modeling

[5] A *clocked* proposition involves both the state and the duration of the path leading to the state (the system "clock"), as explained in [5].

[6] http://ptolemy.org/

```
Q := empty; // Initialize the global event queue to be empty.
for each actor A do
  A.init(); // Initialize actor A, and possibly generate initial events, stored in Q.
end for;

while Q is non-empty do
  E := set of all simultaneous events at the head of Q;
  remove E from Q;
  initialize ports with values in E or "unknown";
  while port values changed do
    for each actor A do
      A.fire();   // May change port values
    end for;
  end while;      // Fixed-point reached for the current tag
  for each actor A do
    A.postfire(); // Updates actor state, and may generate new queue events
  end for;
end while;
```

Figure 1: Pseudo-code of Ptolemy II DE semantics.

multiple events with identical timestamps happening in sequence, where earlier events may cause later ones. Tags are totally ordered using a lexicographical order: $(\tau_1, n_1) \leq (\tau_2, n_2)$ if and only if $\tau_1 < \tau_2$, or $\tau_1 = \tau_2$ and $n_1 \leq n_2$. Two events are *simultaneous* if they have identical tags.

The operational semantics of DE in Ptolemy II can be explained with the pseudo-code in Figure 1. An *event queue* is used for the execution. Events in the event queue are ordered by their tags. Initially, the event queue is empty. At the beginning of the execution, all actors are initialized, and some actors may post initial events to the event queue. Operation then proceeds by iterations. In each iteration, the events with the smallest tag are extracted from the event queue and presented to the actors that receive them. Those actors are *fired*, which means they are invoked to process their input events, and they may also output events through their output ports. A difference between the DE MoC in Ptolemy II and standard DE simulators is that the former incorporates a synchronous-reactive semantics for processing simultaneous events [18]. When events are extracted from the event queue for the receiving actors to process, the semantics for that iteration is defined as the *least fixed-point* of the output values, in a way similar to a *synchronous* model [28]. Concretely, the outputs are first set to a predefined value called *unknown*. Then, the actors receiving events are fired in an arbitrary order, possibly repeatedly, until a fixed-point of all output values is reached. This allows Ptolemy II models to have *feedback loops*. If the model contains *causality cycles*, the fixed-point may have ports with value *unknown*. Finally, when the fixed-points for the port values have been found, the actors that have received input or have been fed events are executed, in the sense that their states are updated and that they may generate future events that are inserted into the event queue (*postfire*).

3.2 Atomic Actors

We briefly introduce a subset of the Ptolemy II atomic actors whose semantics has been formalized in Real-Time Maude. Their semantics is defined in terms of the actions *init*, *fire*, and *postfire*. (We ignore other actions, such as *prefire* and *finalize*, which are not important in this paper.)

- *Clock*. Ptolemy's *clock* actors have as parameters a *clock period*, and same-sized arrays *values* and *offsets*. In each period, a clock generates events with given values and offsets within the period. For example, if the period is 5, the values are $\{3, 8\}$, and the offsets are $\{2, 4\}$, then an event with value 3 is generated at times 2, 7, 12, 17, 22, ..., and an event with value 8 is generated at times 4, 9, 14, ... That is, the *init* action posts an event to the event queue with timestamp 0 for itself to process; the

fire action is triggered by that event and sends the value to the output port; and the *postfire* action posts the next event to the event queue, with timestamp equal to the beginning of the next period.

- *Current Time.* Ptolemy's *current time* actor produces an output token on each firing with a value that is the current model time. That is, the *init* and *postfire* actions do nothing, and the *fire* action consumes an input event, and outputs an event whose timestamp and value are both equal to the timestamp of the input event.

- *Pulse.* When an input is received, a *pulse* actor outputs pulses with values given by the *values* parameter; the parameter *indexes* specifies when those values should be produced. A zero is produced when the iteration count does not match an index. For example, if the *indexes* parameter is "{1, 3, 0, 2, 4}", and the *values* are stored in array A, then the output in the first 5 invocation of *fire* is $A[1]$, $A[3]$, $A[0]$, $A[2]$, and $A[4]$. After that, the output is always 0, unless yet another parameter, *repeat*, is set to true, in which case the output is repeated. The *init* action does nothing, *fire* outputs a value, and *postfire* updates the number of times *fire* has been invoked.

- *Time Delay.* A *timed delay* actor propagates an incoming event after a given delay, which is given by the *delay* parameter. If the *delay* parameter is 0.0, then there is a "microstep" delay in the generation of the output event.

- *Variable Delay.* A *variable delay* actor works in a similar way as a timed delay actor, except that the amount of time delay is specified by an incoming token through the *delay port*.

- *Timer.* The difference between a *timer* actor and a delay actor is that the value of the generated output of a timer is not the same as the input, but is given by the *output* parameter of this actor. The length of the delay is specified by the input received in the actor's lone input port.

- *Noninterruptible Timer.* A *noninterruptible timer* is similar to a normal timer, but with the difference that the noninterruptible timer actor delays the processing of a new input if it has not finished processing a previous input. That is, while an input event is being delayed and the corresponding output has not been sent, other input events are queued.

- *Timed Plotter.* A *timed plotter* records its received events and the times they were received.

- *(Atomic) Finite State Machine (FSM) Actor.* A *finite state machine* (FSM) actor is a transition system containing a finite set of states (or "locations"), a finite set of "variables," and a finite set of transitions. A transition has a guard expression, and can contain a set of output actions. Output actions may assign values to the variables belonging to the FSM actor and/or may send values to the output ports of the actor. When an FSM actor is fired, there is never more than one enabled transition. If there is exactly one enabled transition then it is chosen and the actions contained by the transition are executed. Under the DE director, only one transition step is performed in each iteration.

3.3 Composite Actors

An essential feature of Ptolemy II is hierarchy. It helps hide internal details of parts of a model. It is therefore crucial for managing model complexity, and for achieving modularity and scalability.

Ptolemy II *hierarchical* models contain components (or *actors*) that are themselves Ptolemy II models. Such a hierarchical model can again be encapsulated and be seen as a single *composite actor*. An inner actor of a DE composite actor is executed if that inner actor receives some events at its input ports or if it is fed an event from the event queue. Figure 2 illustrates a hierarchical composition of actors.

In Ptolemy II, each composite actor can have its own *director*, that is, model of computation, to support heterogeneous modeling. If the director of a composite actor is the same as the director of the parent actor, it is called a *transparent* actor. In this paper, we consider only transparent cases since we verify DE models.

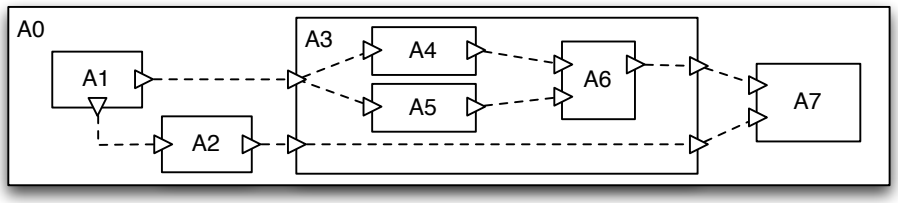


Figure 2: A hierarchical composition of actors. $A0$–$A7$ are actors, $A0$ and $A3$ are composite actors, triangles are ports, and dashed lines are connections.

3.4 Modal Models

Modal models are finite state machines where each state has a *refinement* actor, which is either a composite actor or an FSM actor. The input and output ports of the refinements are the same as those of the modal model. In the top level of a modal model, the output ports are regarded as *both* input and output ports so that the transitions of modal models may use the evaluation result of refinement actors in the *current* computation step. The left-hand side of Fig. 3 shows a modal model with two states.

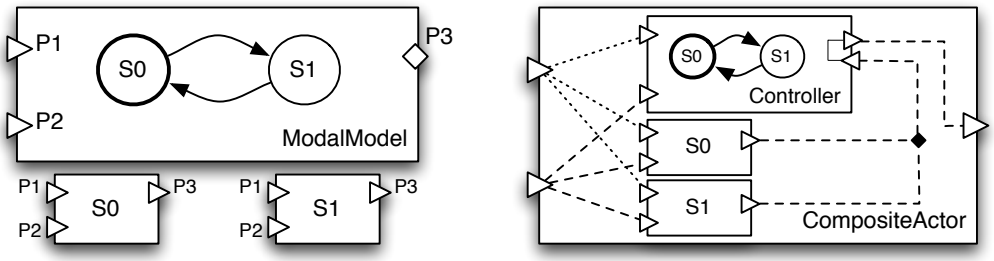


Figure 3: A modal model with 2 states and its equivalent representation as a composite actor. $S0$ and $S1$ are states, diamonds are input/output ports, and a solid line in the right-hand side means a coupled input/output ports.

When a modal model fires, the refinement of the current state is fired and the other refinements are *frozen*. The guards of all outgoing transitions from the current state of the modal model are then evaluated. If exactly one of those guards is true, then the transition is taken and the actions on the transition are executed. The refinement of the next state will be executed in the *next* iteration. In case of a conflict between the refinements and the parent actor, the latter overwhelms the former. For example, if the FSM controller of a modal model and the refinement of a current state are trying to write different values to the same output port, then the value of the FSM controller is taken.

A modal model can be seen as syntactic sugar for a composite actor with *frozen* inner actors, as shown in Fig. 3, where the right-hand side shows the equivalent composite-actor representation of the modal model in the left-hand side. That is, a modal model A is semantically equivalent to a composite actor $\tilde{A}$, with the same ports, that has the controller FSM actor and the refinement actors as inner actors, so that: (i) the ports are connected as indicated in Fig. 3; (ii) the controller FSM actor is fired *after* the refinement actors are fired; (iii) only the refinement inner actors corresponding to the current state of the controller are evaluated, whereas the other refinement actors are frozen, in the sense that their states do not evolve and the values of their outports are ignored; and (iv) if an output port of the controller FSM actor has no value

but its coupled input port has some value, then the output port will have the same value as the input port.

3.5 Subset of Ptolemy II with Real-Time Maude Semantics

We currently support Real-Time Maude analysis of *transparent discrete event* (DE) Ptolemy II models constructed by the following actors: composite actors, modal models, finite state machine (FSM), timed delay, variable delay, clock, current time, timer, noninterruptible timer, pulse, ramp, timed plotter, set variable, expression, single event actors, and algebraic actors such as add/subtract, const, and scale. We also support connections with multiple destinations, split signals, and both single ports and multi-input ports.

3.6 Code Generation Infrastructure

Ptolemy II is built in a highly modular manner, with flexible and extensible components that communicate through generic interfaces. This type of inter-component communication introduces overhead, however, which generally results in component models that are slower than custom-built code. To regain efficiency, Ptolemy II offers a code generation capability with which inter-component communication is reduced by generating "monolithic" code with highly specialized components.

The code generation framework uses an *adapter-based mechanism*. A *codegen adapter* is a component that generates code for an actor. Each actor may have multiple associated adapters, one for each target language (such as C and VHDL). An adapter essentially consists of a Java class file and a *code template* file that together specify the actor's behavior. The latter contains code blocks written in the target language. Supplied with a set of adapters and an initial model, the code generation framework examines the model structure and invokes the adapters to harvest code blocks from the code template files. The main advantages of this scheme are, first, that it decouples the writing of Java code and target code (otherwise the target code would be wrapped in strings and be interspersed with Java code), and second, that it allows using a target language specific editor while working on the target language code blocks.

3.7 Example: A Simple Traffic Light System

Figure 4 shows a Ptolemy DE model of a simple traffic light system consisting of one car light and one pedestrian light at a pedestrian crossing. Each light is represented by a set of *set variable* actors (`Pred` and `Pgrn` represent the pedestrian light, and `Cred`, `Cyel`, and `Cgrn` represent the car light). A light is *on* iff the corresponding variable has the value 1. The lights are controlled by two *finite state machine* (FSM) actors, `CarLight` and `PedestrianLight`, that send values to set the variables; in addition, `CarLight` sends signals (that are *delayed* by one time unit) to the `PedestrianLight` actor through its `Pgo` and `Pstop` output ports.

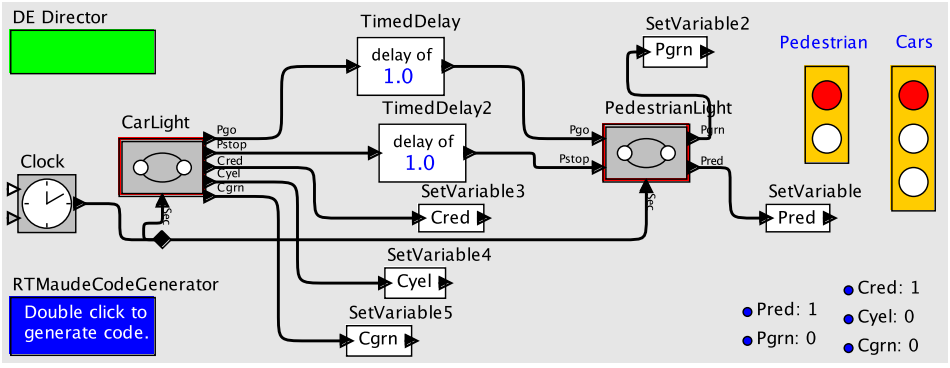
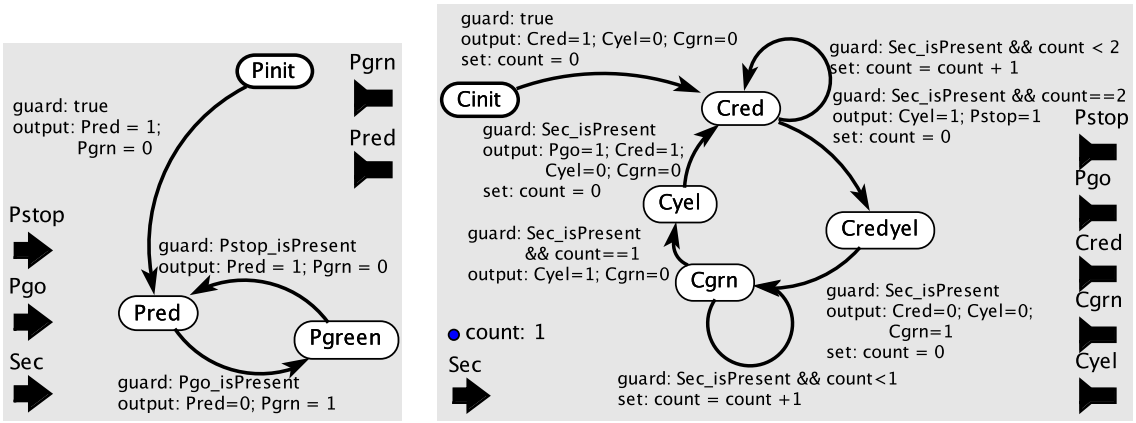


Figure 4: A simple traffic light model in Ptolemy II.

Figure 5a shows the FSM actor `PedestrianLight`. This actor has three input ports (`Pstop`, `Pgo`, and `Sec`), two output ports (`Pgrn` and `Pred`), three internal states, and three transitions. This actor reacts to

signals from the car light (by way of the delay actors) by turning the pedestrian lights on and off. For example, if the actor is in local state `Pred` and receives input through its `Pgo` port, then it goes to state `Pgreen`, outputs the value 0 through its `Pred` port, and outputs the value 1 through its `Pgrn` port.

Figure 5b shows the FSM actor `CarLight`. Assuming that the *clock* actor sends a signal every time unit, we notice, e.g., that one time unit after both the red and yellow car lights are on, these are turned off and the green car light is turned on by sending the appropriate values to the variables (`output: Cred = 0; Cyel = 0; Cgrn = 1`). The car light then stays green for two time units before turning yellow.



(a) `PedestrianLight` (b) `CarLight`

Figure 5: The FSM actors for pedestrian lights and car lights.

4 Real-Time Maude Semantics of Flat Ptolemy II DE Models

To convey our ideas underlying the Real-Time Maude formalization of the semantics of Ptolemy II DE models without introducing too many details, this section presents a slightly simplified version of our semantics, in that we present a semantics for

1. *flat* Ptolemy models; that is, models without hierarchical actors, and
2. assume that all Ptolemy II expressions are defined by constants and simple arithmetic and comparison operations.

Section 5 shows how this slightly simplified semantics is extended to *hierarchical* models, and Section 6 shows how we deal with general Ptolemy expressions that include variables. The entire executable Real-Time Maude semantics is available at `http://www.ifi.uio.no/RealTimeMaude/Ptolemy`.

4.1 Representing Flat Ptolemy II DE Models in Real-Time Maude

This section explains how a flat Ptolemy II DE model is represented as a Real-Time Maude term in (the slightly simplified version of) our semantics. We only show the representation for a subset of the atomic actors in Ptolemy II DE models, and refer to [21] for the definition of the other actors.

Our Real-Time Maude semantics is defined in an object-oriented style, where the global state has the form of a *multiset*

`{`*actors* *connections* `< global : EventQueue | queue :` *event* *queue* `>}`

where

- *actors* are objects corresponding to the actor instances in the Ptolemy model,
- *connections* are the connections between the ports of the different actors, and
- `< global : EventQueue | queue :` *event queue* `>` is an object whose `queue` attribute denotes the global event queue.

This section explains the representation of these entities in Real-Time Maude, and Section 4.2 defines the semantics of the behaviors of the Ptolemy II models.

4.1.1 Actors

Each Ptolemy II actor is modeled in Real-Time Maude as an object instance of a subclass of the following class `Actor`:

```
class Actor | ports : Configuration, parameters : Configuration .
```

The `ports` attribute denotes the set of *ports* of the actor. The `parameters` attribute represents the *parameters* of the actor, together with their user-defined values/expressions. In our model, both ports and parameters are modeled as objects. In particular, a *parameter* is represented as an object, with a name (the identifier of the parameter object) and one attribute `value`:

```
sorts ParamId .    subsort Qid < ParamId < Oid .  --- names for parameters

class Parameter | value : Value .
```

This simple parameter model will be extended in Section 6 when we consider a parameters whose values are expressions that may include variables.

Some actors, such as clocks and timed plotters, have an internal clock measuring "model time." Such actors are represented as object instances of subclasses of the following class `TimeActor`, where `currentTime` denotes the current model time:

```
class TimeActor | currentTime : Time .       subclass TimeActor < Actor .
```

Clocks As explained above, the Ptolemy parameters of an actor (*period*, *offsets*, and *values* for clock actors) are represented in the `parameters` attribute. The only additional attribute needed for the Real-Time Maude representation of *clock* actors is the attribute `index` keeping track of the "index" of the *offsets* and *values* arrays for the next event to be generated:

```
class Clock | index : Nat .      subclass Clock < Actor .
```

For instance, the initial state of the clock described above is represented by the object[7]

```
< 'Clock : Clock | index : 0,
                   parameters : < 'period : Parameter | value : # 5 >
                                < 'offsets : Parameter | value : {# 2.0, # 4.0} >
                                < 'values : Parameter | value : {# 3, # 8} >,
                   ports : < 'output : OutPort | value : # 0, status : absent >
                           < 'trigger : InPort | value : # 0, status : absent >
                           < 'period : InPort | value : # 0, status : absent >  >
```

Current Time Since the superclass `TimeActor` already contains the current time in the `currentTime` attribute, the `CurrentTime` subclass does not add any new attributes:

```
class CurrentTime .          subclass CurrentTime < TimeActor .
```

[7] See Section 4.1.2 for the representation of ports.

Timed Plotter A *timed plotter* records its received data values and the times they were received. In our representation, these values are recorded as a list `(source:` s_1 `time:` t_1 `value:` v_1`) ++ ... ++ (source:` s_n `time:` t_n `value:` v_n`)` of triples `(source:` s_i `time:` t_i `value:` v_i`)`, denoting, respectively, the port from which the data was received, the time it was received, and the received data value. Since such an actor must keep track of the `currentTime`, the `TimedPlotter` class is a subclass of `TimeActor`:

```
class TimedPlotter | eventHistory : EventHistory .     subclass TimedPlotter < TimeActor .

sort EventTriple  EventHistory .
subsort EventTriple < EventHistory .
op source:_time:_value:_ : EPortId Time Value -> EventTriple [ctor] .
op emptyHistory : -> EventHistory [ctor] .
op _++_ : EventHistory EventHistory -> EventHistory [ctor assoc id: emptyHistory] .
```

Other Actors Since the actor *parameters* are represented in the `parameters` attribute of the superclass `Actor`, most actors do not add any new attributes to the attributes inherited from `Actor`. The *pulse* actor adds an attribute `index` that keeps track of the iteration count:

```
class Delay .        --- timed delay
class VariableDelay .
class Timer .
class Pulse | index : Nat .

subclass Delay VariableDelay Timer Pulse < Actor .
```

A *noninterruptible timer* needs some attributes to keep track of the state: `processing` is `true` when the timer has not finished processing previous inputs. The `waitQueue` is a list that stores (the values of) the inputs received while the timer is "busy." This list is therefore a list of time values declared in the usual Maude style. The Real-Time Maude declaration of this class is

```
class NonInterruptibleTimer | processing : Bool,  waitQueue : TimeList .
subclass NonInterruptibleTimer < Actor .

sort TimeList .    subsort Time < TimeList .
op emptyList : -> TimeList [ctor] .
op __ : TimeList TimeList -> TimeList [ctor assoc id: emptyList] .
```

Finite State Machine (FSM) Actors An `FSM-Actor` is characterized by its *current state*, its transitions, and its local variables (the latter are represented by parameters):

```
class FSM-Actor | currState : Location,  initState : Location,  transitions : TransitionSet .
subclass FSM-Actor < Actor .
```

A location is the sort of the local "states" of the transition system. In particular, quoted identifiers (`Qid`s) are state names:

```
sort Location .    subsorts Qid < Location .
```

We model the set of transitions as a semi-colon-separated set of transitions of the form
s_1 `-->` s_2 `{guard:` g `output:` p_{i_1} `|->` $e_{i'_1}$`;...;` p_{i_k} `|->` $e_{i'_k}$ `set:` v_{j_1} `|->` $e_{j'_1}$`;...;` v_{i_l} `|->` $e_{j'_l}$`}`
for states/locations s_1 and s_2, port names p_i, variables v_i, and expressions e_i. The guard, output, and/or set parts may be omitted. In Real-Time Maude, such sets of transitions are declared as follows:

```
sorts Transition TransitionSet .   subsort Transition < TransitionSet .
op _-->_‘{_‘} : Location  Location TransBody -> Transition [ctor] .
op emptyTransitionSet : -> TransitionSet [ctor] .
op _;_ : TransitionSet TransitionSet -> TransitionSet [ctor assoc comm id: emptyTransitionSet] .

sort TransBody .
op guard:_output:_set:_ : Exp AssignMap AssignMap -> TransBody [ctor] .
```

In the flat setting, we assume that all expressions consist of

- constants (which have sort `Value`) : (0, 1, `true`, ...)
- variables (which are represented by `parameter` objects)
- simple arithmetic, logical, and comparison operators: `+`, $\times$, `&&`, `!`, `<`, ...
- `isPresent(`*P*`)`, which is `true` if there is some (current) input in the given port `P`, and is `false` if there is no current input in port `P`.

4.1.2 Ports

A *port* is represented as an object, with a name (the identifier of the port object), a status (`unknown`, `present`, or `absent`, denoting the "current" knowledge about whether there is input/output in the current iteration), and a `value`. We also have subclasses for input and output ports:

```
sorts PortId .    subsort Qid < PortId < Oid .  --- names for (local) ports

class Port | status : PortStatus, value : Value .

class InPort .     subclass InPort < Port .
class OutPort .    subclass OutPort < Port .

sort PortStatus .
ops unknown present absent : -> PortStatus [ctor] .
```

We also support multiple input ports, which are connected to multiple output ports:

```
class MultiInPort | source : EPortIdSet .    subclass MultiInPort < InPort .
```

4.1.3 Connections

A connection is a term p_o `==>` $p_{i_1}; \ldots ; p_{i_n}$ of sort `Connection`, where the p_js are either local port names or have the form a`!`p for a the *relative* name of an actor. Such a connection connects the output port p_o to all the input ports $p_{i_1}, \ldots, p_{i_n}$. Since connections appear in configurations, and are not messages, they are also terms of sort `ObjectConfiguration`:

```
sort Connection .
op _==>_ : EPortId EPortIdSet -> Connection [ctor] .
subsort Connection < ObjectConfiguration .

sort EPortId .
op _!_ : ActorID PortId -> EPortId [ctor] .

sort EPortIdSet .      subsort EPortId < EPortIdSet .
op noPort : -> EPortIdSet [ctor] .
op _;_ : EPortIdSet EPortIdSet -> EPortIdSet [ctor assoc comm id: noPort] .
```

A multiple input port and its connection are transformed to a set of input ports with duplicated connections, whose port names are annotated by the name of their source ports as follows:

```
var PORTS : Configuration . vars P P' : PortId . var PS : PortStatus . var EPIS EPIS' : EPortIdSet .
vars O O' : Oid . var V : Value .

eq < O : Actor | ports : < P : MultiInPort | status : PS, value : V,
                                             source : (O' ! P') ; EPIS' > PORTS >
   (O' ! P' ==> (O ! P) ; EPIS)
=
   < O : Actor | ports : < P # (O' ! P') : InPort | status : PS, value : V >
                         < P : MultiInPort | source : EPIS' > PORTS >
   (O' ! P' ==> (O ! P # (O' ! P')) ; (O ! P) ; EPIS) .

eq < O : Actor | ports : < P : MultiInPort | source : noPort > PORTS >
   (O' ! P' ==> (O ! P) ; EPIS)
 = < O : Actor | ports : PORTS >   (O' ! P' ==> EPIS) .
```

4.1.4 The Global Event Queue

The global event queue is represented by an object

< global : EventQueue | eventQueue : *event queue* >

where *event queue* is an ::-separated list, ordered according to time until firing, of terms of the form

set of events ; *time to fire* ; *microstep*

where the *set of events* is a set of events, with each event characterized by the "global port name" where the generated event should be output and the corresponding value, *time to fire* denotes the time *until* the events are supposed to fire, and *microstep* is the additional "microstep" until the event fires:

```
sort Event .
op event : EPortId Value -> Event [ctor] .

sort Events .       subsort Event < Events .
op noEvent : -> Events [ctor] .
op __ : Events Events -> Events [ctor assoc comm id: noEvent] .

sort TimedEvent .
op _;_;_ : Events Time Nat -> TimedEvent [ctor] .

sort EventQueue .     subsort TimedEvent < EventQueue .
op nil : -> EventQueue [ctor] .
op _::_ : EventQueue EventQueue -> EventQueue [ctor assoc id: nil] .
```

4.1.5 Example: Representing the Flat Traffic Light Model

Consider the flat non-fault-tolerant traffic light system given in Section 3.7. The Real-Time Maude representation of the TimedDelay2 delay actor in the initial state is then

```
< 'TimedDelay2 : Delay | parameters :  < 'delay : Parameter | value : # 1.0 >,
                         ports : < 'input : InPort | value : # 0, status : absent >
                                 < 'output : OutPort | value : # 0, status : absent >  >
```

Likewise, the FSM actor CarLightNormal is represented as the term[8]

[8] To save space, some terms are replaced by '...'

```
< 'CarLightNormal : FSM-Actor |
        initState : 'Cinit,  currState : 'Cinit,
        parameters :  < 'count : Parameter | value : # 1 >,
        ports : < 'Sec : InPort | value : # 0, status : absent >
                < 'Pgo : OutPort | value : # 0, status : absent >
                ...,
        transitions :
          ('Cinit --> 'Cred
              {guard: (# true)
               output: ('Cred |-> # 1) ; ('Cyel |-> # 0) ; ('Cgrn |-> # 0)
               set: 'count |-> # 0}) ;
          ('Cred --> 'Cred
              {guard: (isPresent('Sec) && ('count lessThan # 2))
               output: emptyMap  set: 'count |-> ('count + # 1)}) ; ... >
```

The connection from the output port `output` of the `Clock` actor to the input port `Sec` of `CarLightNormal` and the input port `Sec` of `PedestrianLightNormal` is represented by the term

```
('Clock ! 'output)  ==>  ('PedestrianLightNormal ! 'Sec) ; ('CarLightNormal ! 'Sec)
```

The entire state thus consists of two FSM actor objects, ten connections, two delay objects, five set variable objects, and the global event queue object.

4.2 Specifying the Behavior of Flat DE Models

The behavior of Ptolemy DE models can be summarized as repeatedly performing the following actions:

- Advance time until the time when the first event(s) in the event queue should fire.
- Then an iteration of the system is performed. That is,
  1. The events that are supposed to fire are added to the corresponding output ports; the `status` of all other ports is set to `unknown`.
  2. (Fire) Then the *fixed point* of all ports is computed by gradually increasing the knowledge about the presence/absence of inputs to and output from ports until a fixed-point is reached.
  3. (Postfire) Finally, the actors with inputs or scheduled events are executed; states are changed and new events are generated and inserted into the global event queue.

The following tick rule advances time until the time when the first events in the event queue are scheduled; that is, until the time-to-fire of the first events in the event queue is `0` (we first declare all the variables used in this section):

```
var CF : Configuration . vars NECF NECF' : NEConfiguration .
vars SYSTEM OBJECTS REST PORTS PORTS2 PARAMS : ObjectConfiguration . vars OBJ OBJECT : Object .
vars O O' : Oid. vars P P : PortId . var PS : PortStatus . vars EPIS EPIS : EPortIdSet .
var VI : VarId . vars V V1 V2 TV : Value. vars E TG : Exp .
var EVTS : Events . var QUEUE : EventQueue . var EH : EventHistory .
var T T' : Time . var NZT : NzTime . var N : Nat . var NZ : NzNat . vars STATE STATE' : Location .
var TRANSSET : TransitionSet . var BODY : TransBody . vars OL AL : AssignMap .

rl [tick] :
   {SYSTEM  < global : EventQueue | queue : (EVTS ; NZT ; N) :: QUEUE >}
  =>
   {delta(SYSTEM, NZT)
    < global : EventQueue | queue : (EVTS ; 0 ; N) :: delta(QUEUE, NZT) >}
  in time NZT .
```

In this rule, the first element in the event queue has non-zero delay `NZT`. Time is advanced by this amount `NZT`, and as a consequence, the (first component of the) event timer goes to zero. In addition, the function `delta` is applied to all the other objects (denoted by `SYSTEM`) in the system. The function `delta` defines the effect of time elapse on the objects. This function is also applied to the other elements in the event queue, where it decreases the remaining time of each event set by the elapsed time `NZT` (x `monus` y equals $\max(0, x - y)$):

```
op delta : EventQueue Time -> EventQueue .
eq delta((EVTS ; T ; N) :: QUEUE, T') = (EVTS ; T monus T' ; N) :: delta(QUEUE, T') .
eq delta(nil, T) = nil .
```

The function `delta` on configurations distributes over the elements in the configuration, and must be defined on the single objects which are instance of `TimeActor`, as shown later.

```
op delta : Configuration Time -> Configuration .
eq delta(none, T) = none .
eq delta(NECF NECF', T) = delta(NECF, T) delta(NECF', T) .
```

The next rule is a "microstep tick rule" that advances "time" with some microsteps if needed to enable the first events in the event queue:

```
rl [shortTick] :
   {SYSTEM  < global : EventQueue | queue : (EVTS ; 0 ; NZ) :: QUEUE >}
  =>
   {SYSTEM  < global : EventQueue | queue : (EVTS ; 0 ; 0) :: QUEUE >} .
```

Finally, when the remaining time and microsteps of the first events in the event queue are both zero, an iteration of the system can be performed:

```
rl [executeStep] :
   {SYSTEM < global : EventQueue | queue : (EVTS ; 0 ; 0) :: QUEUE >}
  =>
   {< global : EventQueue | queue : QUEUE >
    postfire(portFixPoints(addEventsToPorts(EVTS, clearPorts(SYSTEM))))} .
```

The function `clearPorts` starts the execution of an iteration by clearing all ports; that is, it sets the `status` of each port in the system to `unknown`. The operator `addEventsToPorts` inserts the events scheduled to fire into the corresponding output ports. The `portFixPoints` function then finds the fixed points for all the port (this function corresponds to the *fire* action in Ptolemy), and `postfire` "executes" the steps on the computed port fixed-points by changing the states of the objects and generating new events and inserting them into the global event queue.

It is important to notice that these functions are declared to be *partial* functions. Since the equations defining these functions only apply to terms of sort `Configuration` and its subsorts (`NEConfiguration`, `ObjectConfiguration`, and so on), this ensures that `clearPorts` has been computed *before* `addEventsToPorts` is computed, which again must happen before `portFixPoints` is computed, and so on. Mathematically, this means that our equations are *confluent*.

```
ops clearPorts portFixPoints postfire : Configuration ~> Configuration .
```

To completely define the behavior of the actors, we must define the functions `clearPorts`, `portFixPoints`, `postfire`, and `delta` on the different objects in the system.

4.2.1 Clearing Ports

The `clearPorts` function distributes to each actor object in the state, and then clears all the ports of each actor, that is, sets the `status` to `unknown` (notice, as mentioned above, that the equations only apply to terms of sort `Configuration`):

```
eq clearPorts(OBJ CF) = clearPorts(OBJ) clearPorts(CF) .

eq clearPorts(< O : Actor | ports : PORTS >) = < O : Actor | ports : clearPorts(PORTS) > .
eq clearPorts(< P : Port | status : PS > PORTS) = < P : Port | status : unknown > clearPorts(PORTS) .
eq clearPorts(CF) = CF [owise] .
```

4.2.2 Computing the Fixed-Point for Ports

The idea behind the definition of the function `portFixPoints`, that computes the fixed-point for the values of all the ports, is simple. The state has the form `portFixPoints`(*objects and connections*), where initially, the only port information are the events scheduled for this iteration. For each possible case when the status of an `unknown` port can be determined to be either `present` or `absent`, there is an equation

```
eq portFixPoints(< O : ... | ports : < P : Port | status : unknown > PORTS,  ... >
                 connections and other objects) =
   portFixPoints(< O : ... | ports : < P : Port | status : present , value : ... > PORTS, ... >
                 connections and other objects) .
```

(and similarly for deciding that input/output will be `absent`). The fixed-point is reached when no such equation can be applied. Then, the `portFixPoints` operator is removed by using the `owise` construct of Real-Time Maude:

```
eq portFixPoints(OBJECTS) = OBJECTS [owise] .
```

We first define the general cases of `portFixPoints` that apply to any `Actor` instance. The following equation propagates port status from a "known" output port to a connecting `unknown` input port. The present/absent `status` (and possibly the `value`) of the output port `P` of actor `O` is propagated to the input port `P'` of the actor `O'` through the connection `(O ! P) ==> ((O' ! P') ; EPIS)`:

```
ceq portFixPoints(< O : Actor | ports : < P : OutPort | status : PS , value : V > PORTS >
                  ((O ! P) ==> ((O' ! P') ; EPIS))
                  < O' : Actor | ports : < P' : InPort | status : unknown > PORTS2 >
                  REST)
  = portFixPoints(< O : Actor | >
                  ((O ! P) ==> ((O' ! P') ; EPIS))
                  < O' : Actor | ports : < P' : InPort | status : PS, value : V > PORTS2 >
                  REST)
   if PS =/= unknown .
```

If all input ports of an actor are absent, then the actor should not generate any output, unless it has a scheduled event from the event queue. In this case, the `status` of each output port of the actor is set to `absent`:

```
ceq portFixPoints(< O : Actor | ports : < P : OutPort | status : unknown > PORTS >  REST)
  = portFixPoints(< O : Actor | ports : < P : OutPort | status : absent >
                                        setUnknownOutPortsAbsent(PORTS) >  REST)
  if allInputPortsAbsent(PORTS) .

op allInputPortsAbsent : Configuration -> Bool .
eq allInputPortsAbsent(< P : InPort | status : PS > PORTS)
```

```
  = (PS == absent) and allInputPortsAbsent(PORTS) .
eq allInputPortsAbsent(PORTS) = true [owise] .

op setUnknownOutPortsAbsent : Configuration ~> Configuration .
eq setUnknownOutPortsAbsent(< P : OutPort | status : unknown >  PORTS)
 = < P : OutPort | status : absent >  setUnknownOutPortsAbsent(PORTS) .
eq setUnknownOutPortsAbsent(PORTS) = PORTS [owise] .
```

It is also possible that some actor has an *isolated* input port that has no incoming connection. Obviously, the input port has no value; i.e., its `status` should be `absent`:

```
ceq portFixPoints(< O : Actor | ports : < P : InPort | status : unknown > PORTS >  REST)
  = portFixPoints(< O : Actor | ports : < P : InPort | status : absent > PORTS >  REST)
  if not connectedInPort(O ! P, REST) .

op connectedInPort : EPortId Configuration -> Bool .
eq connectedInPort(O ! P,  (O' ! P' ==> (O ! P) ; EPIS)  < O' : Actor | > CF) = true .
eq connectedInPort(O ! P, CF) = false [owise] .
```

The `portFixPoints` function must then be defined for each kind of actor to decide whether the actor produces any output in a given port. For example, the *timed delay* actor does not produce any output in this iteration as a result of any input. Therefore, if its `status` is `unknown` (that is, the delay actor did not schedule an event for this iteration), its output port should be set to `absent`:

```
eq portFixPoints(< O : Delay | ports : < P : OutPort | status : unknown > PORTS >  REST)
 = portFixPoints(< O : Delay | ports : < P : OutPort | status : absent > PORTS >  REST) .
```

Actors, such as variable delay, clock actors, timers, etc., that generate "delayed" events as a result of receiving input, have the same definition of `portFixPoint`.

Other actors generate immediate output when receiving input. For example, when the *current time* actor gets an input, it outputs the current model time, given by its `currentTime` attribute. Furthermore, when its lone input port is `absent`, its lone output port is also set to `absent`:

```
ceq portFixPoints(< O : CurrentTime | currentTime : T,
                                      ports : < P : InPort | status : PS >
                                              < P' : OutPort | status : unknown > >
                  REST)
  = portFixPoints(< O : CurrentTime | ports : < P : InPort | >
                                              < P' : OutPort | status : PS, value : # T >
                  REST)
  if PS =/= unknown .
```

Likewise, when a *pulse* actor gets input through its *trigger* port, it should generate immediate output through its *output* port. Then an output value is produced as described in Section 3.2, which is done by the function `getValue`:

```
eq portFixPoints(< O : Pulse | index : N,
                               parameters : < 'indexes : Parameter | value : V1 >
                                            < 'values : Parameter | value : V2 > PARAMS,
                               ports : < 'trigger : InPort | status : present >
                                       < 'output : OutPort | status : unknown > PORTS >
                  REST)
   = portFixPoints(< O : Pulse | ports : < 'trigger : InPort | >
                                         < 'output : OutPort | status : present,
                                                               value : getValue(V1, V2, N) >
                                         PORTS >
                   REST) .
```

For *FSM* actors, the `portFixPoints` function must check whether at most one transition is enabled at any time by evaluating the guard expressions. In the following equation, one transition from the current state `STATE` is enabled, and there is no other enabled transition. In addition, there is *some* input to the actor (through input port `P'`), and some output ports have status `unknown`. The function `updateOutPorts` then updates the status and the values of the output ports according to the current state and input:

```
ceq portFixPoints(< O : FSM-Actor | ports : < P' : InPort | status : present >
                                        < P : OutPort | status : unknown > PORTS,
                                    currState : STATE,  parameters : PARAMS,
                                    transitions : (STATE --> STATE' {guard: TG output: OL set: AL}) ;
                                                  TRANSSET >
                  REST)
=
    portFixPoints(< O : FSM-Actor | ports : < P' : InPort | >
                                            updateOutPorts(PARAMS, OL, < P : OutPort | > PORTS) >
    REST)
if transApplicable(< P' : InPort | > < P : OutPort | > PORTS, PARAMS, TG)
 /\ noGuardTrue(< P' : InPort | > < P : OutPort | > PORTS, PARAMS, TRANSSET) .
```

The function `transApplicable` holds if the guard evaluates to `true`, for the current values of the local state variables (as given by the `parameters` objects) and current knowledge of port states and values. Likewise, `noGuardTrue` holds iff no transition in the transitions set given as last argument starts in the given location and satisfies the guard. The definition of these functions is straight-forward and is not shown here.

The `updateOutPorts` function is defined as follows. Each output port is assigned a value of the corresponding output action of the given transition, and all remaining output ports are set to be absent in the end of the update process:

```
op updateOutPorts : Configuration AssignMap Configuration -> Configuration .
eq updateOutPorts(PARAMS, (VI |-> V ; OL), < VI : OutPort | status : unknown > PORTS)
 = < VI : OutPort | status : present, value : V >  updateOutPorts(PARAMS, OL, PORTS) .
eq updateOutPorts(PARAMS, OL, PORTS) = setUnknownOutPortsAbsent(PORTS) [owise] .
```

Other equations for `portFixPoints` on FSM actors specify the cases when no transition is enabled. In these cases, every output ports should be set to *absent*:

```
ceq portFixPoints(< O : FSM-Actor | ports : < P : InPort | status : present > PORTS,
                                    parameters : PARAMS,  transitions : TRANSSET >
              REST)
=
    portFixPoints(< O : FSM-Actor | ports : < P : InPort | > setUnknownOutPortsAbsent(PORTS) >
                  REST)
 if allGuardsFalse(< P : InPort | > PORTS, PARAMS, TRANSSET) .
```

The function `setUnknownOutPortsAbsent` sets the `status` of each output port with `status unknown` to `absent`, and the function `allGuardsFalse` checks whether the guard in each transition evaluates to `false` in the current environment.

4.2.3 Postfire

The `postfire` function updates internal states and generates future events that are inserted into the event queue. The `postfire` function distributes over the actor objects in the configuration:

```
eq postfire(OBJECT NECF) = postfire(OBJECT) postfire(NECF) .
eq postfire(CF) = CF [owise] .
```

The second equation defines the "default" case when `postfire` does not change the state of an actor and does not generate a new event. Therefore, we only to define the positive cases where either the internal state of an actor should be changed as a result of the firing, and/or when when the actor generates a future event that should be inserted into the event queue. For example, the *current time* actor does not have a state that is changed, except by the passage of time, and does not schedule later events, so that we do not need to specify an equation defining `postfire` for current time objects.

Sometimes, `postfire` generates a new event with value v that should fire at time t and microstep n from the current time. In these cases, `postfire` puts the new event into the event queue, and the corresponding equation has the form

```
eq postfire(< O : C | ports : < P : OutPort | >  PORTS, ... >)
   < global : EventQueue | queue : QUEUE >
  =
   < O : C | ... >
   < global : EventQueue | queue : addEvent(event(O ! P, v), t, n, QUEUE) > .
```

where the function `addEvent` inserts the new event in the correct place in the event queue.

Delay If a time delay actor has input in its `'input` port, then it generates an event with delay equal to the current value of the `'delay` parameter. If this delay is `0.0`, the microstep is `1`, otherwise the microstep is `0`:

```
eq postfire(< O : Delay | ports : < 'input : InPort | status : present, value : V >
                                  < 'output : OutPort | >,
                            parameters : < 'delay : Parameter | value : TV > PARAMS >)
   < global : EventQueue | queue : QUEUE >
  =
   < O : Delay | >)
   < global : EventQueue | queue : addEvent(event(O ! 'output, V), toTime(TV),
                                           if toTime(TV) == 0 then 1 else 0 fi, QUEUE) > .
```

The *variable delay* actor has an extra *delay* port to specify time delay. If the delay port is absent, the behavior is the same as the delay actor. However, if the delay port has some value, the value of the port is used instead of the `'delay` parameter:

```
eq postfire(< O : VariableDelay | ports : < 'input : InPort | status : present, value : V >
                                          < 'delay : InPort | status : present, value : TV >
                                          < 'output : OutPort | > PORTS >)
 = < O : VariableDelay | >
   < global : EventQueue | queue : addEvent(event(O ! 'output, V), toTime(TV),
                                           if toTime(TV) == 0 then 1 else 0 fi, QUEUE) > .
```

Clock When a clock actor produces *output*, the `postfire` function should schedule the next event, and update the `index` variable (in the second equation a new "cycle" is started):

```
ceq postfire(< O : Clock | ports : < P : OutPort | status : present > PORTS,
                           parameters : < 'offsets : Parameter | value : V1 >
                                        < 'values : Parameter | value : V2 > PARAMS,
                           index : N >)
    < global : EventQueue | queue : QUEUE >
  =
    < O : Clock | index : N + 1 >
    < global : EventQueue | queue : addEvent(event(O ! P, V2(#(s N))), TIME-TO-FIRE,
                                             if TIME-TO-FIRE == 0 then 1 else 0 fi, QUEUE) >
   if TIME-TO-FIRE := toTime((V1(#(s N))) - (V1(# N)))
      /\ ((# N + # 1) lessThan (V1 .. 'length(()) )) == # true .
```

If A is an array and n a number, then the expression A(# n) denotes value of the nth element of A. A similar equation defines `postfire` when a new "cycle" is started; that is, when `N + 1` equals the length of the *offsets* array.

Timer If a timer actor received input at its `input` port, it generates an event with value equal to the current value of the `output` parameter. The event is scheduled to fire in the time given by the value of the `input` port:

```
eq postfire(< O : Timer | parameters : < 'output : Parameter | value : V > PARAMS,
                          ports : < 'input : InPort | status : present , value : TV >  PORTS >)
   < global : EventQueue | queue : QUEUE >
 =
   < O : Timer | >
   < global : EventQueue | queue : addEvent(event(O ! 'output, V), toTime(TV),
                                            if toTime(TV) == 0 then 1 else 0 fi, QUEUE) > .
```

Timed Plotter At the end of an iteration, the timed plotter records any input through its *multi-input* port by adding triple `source:` *`channel`* `time:` *`current time`* `value:` *`value of input`* for each such input to its `eventHistory` attribute. This job is done by the auxiliary function `genEventHistory` which traverses the instances of `'input` ports and generates a "history triple" for those ports where input were present:

```
eq postfire(< O : TimedPlotter | currentTime : T,  eventHistory : EH, ports :  PORTS >)
 = < O : TimedPlotter | eventHistory : EH ++ genEventHistory(T, PORTS) >) .

op genEventHistory : Time Configuration ~> EventHistory .
eq genEventHistory(T, < 'input # (O ! P) : InPort | status : present, value : V >  PORTS)
 = (source: O ! P time: T value: V) ++ genEventHistory(T, PORTS) .
eq genEventHistory(T, PORTS) = emptyHistory [owise] .
```

FSM Actors An FSM actor does not generate future events, but `postfire` updates the internal state (location and variables/parameters) of the actor if it has gotten input and exactly one of its transitions was enabled:

```
ceq postfire(< O : FSM-Actor | ports : < P : InPort | status : present > PORTS,
                               parameters : PARAMS,  currState : STATE,
                               transitions : STATE --> STATE' {guard: TG output: OL set: AL} ;
                                             TRANSSET >)
  =
     < O : FSM-Actor | parameters : updateParam(PARAMS, AL, PARAMS),  currState : STATE' >)
 if transApplicable(< P : InPort | > PORTS, PARAMS, TG)
  /\ noGuardTrue(< P : InPort | > PORTS, PARAMS, TRANSSET) .

op updateParam : Configuration AssignMap Configuration -> Configuration .
eq updateParam(CF, (VI  |-> E ; AL), < VI : Parameter | > PARAMS)
 = < VI : Parameter | value : [[ E ]] CF > updateParam(CF, AL, PARAMS) .
eq updateParam(CF, AL, PARAMS) = PARAMS [owise] .
```

Here, `[[ E ]] CF` gives the value of the expression `E` when evaluated in the environment `CF`. Notice that the "old" environment is used to compute the value of each expression.

4.2.4 Defining Timed Behavior

Finally, we must define the function `delta`, that specifies the effect of time elapse, on single actors. Time only affects the internal state of `TimeActor` objects (`CurrentTime` and `TimedPlotter`), that have an internal "clock" attribute `currentTime`, by increasing the value of `currentTime` according to the elapsed time:

```
eq delta(< O : TimeActor | currentTime : T >, T') = < O : TimeActor | currentTime : T + T' > .
```

Time elapse does not affect other actors and connections:

```
eq delta(CF, T) = CF [owise] .
```

4.3 Defining Initial States

The initial state is defined as the term:

```
{init(< global : EventQueue | queue : nil >  actors)  connections}
```

where `init` adds the initial events of the system to the global event queue. In our flat subset, only *single event* (not shown) and *clock* actors generate such initial events:

```
eq init(< O : Clock | parameters : < 'value : Parameter | value : V1 >
                                   < 'offsets : Parameter | value : V2 >  PARAMS >
        < global : EventQueue | queue : QUEUE > REST)
 =
   < O : Clock | >
   init(< global : EventQueue | queue : addEvent(event(O ! 'output, V1(# 0)), toTime(V2(# 0)), 0, QUEUE) >
        REST) .

eq init(CF) = CF [owise] .
```

5 Real-Time Maude Semantics for Hierarchical DE Models

We define the Real-Time Maude semantics for transparent hierarchical DE models by extending our semantics for flat models to composite actors and modal models, and by making some changes to the flat semantics as described below. Our representation preserves the hierarchical structure of a Ptolemy II model; therefore such models and their Real-Time Maude counterparts are essentially isomorphic, so that we can easily reconstruct the original Ptolemy II models to provide graphical counter-examples.

Some of the difficulties involved in extending the semantics to the hierarchical case include:

- The event management is different. DE models have a *global* event queue, but events could be generated at any level in the hierarchy and/or must be fed to actors deep down in the hierarchy.
- Computing fixed-points for hierarchical models is much harder than in the flat case. Naive approaches easily fall into infinite loops or unnecessarily complex semantics. In addition, the fixed-point computation should be finished only after all levels of fixed-point computation are completed.
- The semantics of modal models in the Ptolemy II documentation is somewhat unclear. There are many subtle or implicit assumptions concerning the execution of modal models, such as the evaluation order of inner actors, event generation in frozen actors, and handling input/output ports of modal models. We proposed the transformation from modal models to composite actors for clarifying the semantics of modal models.

5.1 Representing Hierarchical Actors

Composite actors are modeled as object instances of the class `CompositeActor`, which extends its superclass `Actor` with one attribute, `innerActors`, which denotes the inner actor objects and connections of the composite actor:

```
class CompositeActor | innerActors : Configuration .    subclass CompositeActor < Actor .
```

We also add the following new class `AtomicActor` to distinguish the atomic actors from composite actors, and declare each *atomic* actor class to be a subclass of `AtomicActor`.

```
class AtomicActor .    subclass AtomicActor < Actor .
```

Each actor can be uniquely identified by its *global actor identifier*, which is a list $o_1 \,.\, o_2 \,.\, \ldots \,.\, o_n$ of object names, where o_1 is the name a top-level actor, and o_{i+1} is the name of an inner actor of the composite actor with global actor identifier $o_1 \,.\, \ldots \,.\, o_i$.

We represent modal models as composite actors according to the frozen-composite-actor semantics for modal models described in Section 3. The class `ModalModel` has an additional attribute `controller` pointing to the controller FSM in `innerActors`, and the additional `refinementSet` attribute mapping each state in the modal model to its refinement:

```
class ModalModel | controller : Oid,  refinement : RefinementSet .
subclass ModalModel < CompositeActor .
```

In addition, the definition of the basic `Actor` class adds an attribute `status` whose value is either `enabled` or `disabled`, depending on whether the actor is disabled as a result of being contained in a refinement of a "frozen" state in a modal model. Any equation generating a value at outports or changing parameters, such as those defining `portFixPoints` and `postfire`, only apply to objects whose `status` is `enabled`. Other equations, such as those defining `clearPorts`, also apply to `disabled` actors.

5.2 Extracting and Adding Events to the Event Queue

In the flat setting, each actor is at the same hierarchical level as the global `EventQueue` object. Each actor therefore has direct access to the event queue, so that at the start of an iteration, the scheduled events could be directly inserted into the corresponding actor ports (by the function `addEventsToPorts`), and actors could add generated events directly into the global event queue (in `postfire`).

In the hierarchical case, an actor that receives or generates an event from/to the global event queue can be located deep down in the actor hierarchy. Events communicated between the actors and the event queue may therefore cross hierarchical boundaries. We have modeled this "traveling" of events by "method calls" or "message passing." For example, inserting an event into the output port p of some actor with global actor identifier g corresponds to generating the message `active-evt(event(`g`!`p`,`v`))`. Likewise, an event generated by an actor is "sent" to the event queue as a message of the form `schedule-evt(`*event*`,` *time*`,` *microstep*`)`:

```
msg schedule-evt : Event Time Nat -> Msg .
msg active-evt : Event -> Msg .
```

For example, when an actor generates an event, it creates an `schedule-evt` "message" (we again first declare the variables used in this section):

```
vars O O CO : Oid . vars CF CF' : Configuration . var MSGS : MsgConfiguration .
vars SYSTEM OBJECTS REST REST2 PORTS PORTS2 PARAMS : ObjectConfiguration .
var AI : ActorID . var NAI : NEActorID . var ST : ActorStatus .
vars P P : PortId . var PS : PortStatus . vars EPIS EPIS : EPortIdSet .
var REFS : RefinementSet . vars STATE STATE' : Location . vars V TV : Value. var N : Nat .
var EVENT : Event . var EVTS : Events . var QUEUE : EventQueue . var T : Time .

eq postfire(< O : Delay | status : enabled,
                          parameters : < 'delay : Parameter | value : TV >   PARAMS,
                          ports : < 'input : InPort | status : present, value : V >
                                  < 'output : OutPort | > PORTS >)
 = schedule-evt(event(O ! 'output, V), toTime(TV), if toTime(TV) == 0 then 1 else 0 fi)
   < O : Delay | > .
```

Such an event is propagated towards the top of the actor hierarchy by the following equation, which moves the `schedule-evt` message inside `innerActors` of a composite actor one level up:

```
eq < O : CompositeActor | innerActors : CF schedule-evt(event(AI ! P, V), T, N) >
 = < O : CompositeActor | innerActors : CF > schedule-evt(event((O . AI) ! P, V), T, N) .
```

When the `schedule-evt` request has reached the top of the hierarchy, it is added to the global event queue:

```
eq < global : EventQueue | queue : QUEUE > schedule-evt(EVENT, T, N)
 = < global : EventQueue | queue : addEvent(EVENT, T, N, QUEUE) > .
```

The propagation of `active-evt`s from the event queue to some inner actor is explained below.

The rewrite rule `executeStep` that models an iteration of the system is modified compared to the flat case, so that for each event `event(`*globalActorId* `!` *portId*`,` *v*`)` scheduled for this iteration (i.e., included in `EVTS` below), a "message" `active-evt(event(`*globalActorId* `!` *portId*`,` *v*`))` is added to the state; the function `releaseEvt` generates this message set from a set of events:

```
rl [executeStep] :
   {SYSTEM  < global : EventQueue | queue : (EVTS ; 0 ; 0) :: QUEUE >}
  =>
   {< global : EventQueue | queue : QUEUE >
    postfire(portFixPoints(releaseEvt(EVTS) clearPorts(SYSTEM)))} .
```

Since messages are not terms of sort `ObjctConfiguration`, subtle use of variables of the subsort `Object-Configuration` in equations defining `portFixPoints` ensure that all events are delivered to the actors before `portFixPoints` is computed.

5.3 Defining `clearPorts`, `portFixPoints`, and `postfire` for Hierarchical Models

For *atomic* actors, `clearPorts` should just set the `status` of each port of the actor to `unknown`, as before. For *composite* actors, it should also propagate to the inner actors. To ensure that the appropriate equation applies to an actor, we must modify the definition of `clearPorts` for atomic actors to apply only to objects of class `AtomicActor`:

```
eq clearPorts(< O : AtomicActor | ports : PORTS >)  =  < O : AtomicActor | ports : clearPorts(PORTS) > .
eq clearPorts(< O : CompositeActor | innerActors : CF, ports : PORTS >)
 = < O : CompositeActor | innerActors : clearPorts(CF), ports : clearPorts(PORTS) > .
```

The `postfire` function is almost unchanged for the "flat" actors; the only modification is to ensure that `postFire` is not applied to *disabled* actors, since disabled actors should not change their states or generate new events. For a composite actor, `postfire` just propagates to its inner actors. The condition ensures that this equation is not applied to modal models:

```
ceq postfire(< O : CompositeActor | status : ST, innerActors : CF >)
  = < O : CompositeActor | innerActors : if ST == enabled then postfire(CF) else CF fi >)
 if class(< O : CompositeActor | >) == CompositeActor .
```

The extension of the `portFixPoints` function to the hierarchical case is more subtle. The `portFixPoints` function should distribute to the submodels of composite actors to compute the fixed points of these subsystems. However, an equation of the form

```
eq portFixPoints(< O : CompositeActor | innerActors : OBJECTS, ... > REST)
 = portFixPoints(< O : CompositeActor | innerActors : portFixPoints(OBJECTS), ... > REST) .
```

would be applicable again when the inner `portFixPoints` function disappears, leading to nontermination (and non-applicability of the `owise` equation defining the end of the fixed-point computation). The problem with the above equation is that `portFixPoints` is applied to inner actors even though they may already have reached their fixed points. To avoid this situation, we execute `portFixPoints` for inner actors *only if* some inner actors have not yet reached a fixed-point. This can be easily accomplished since actors are activated in DE models only if input ports of the actors receive some value either from the event queue or from the other actors by connections.

We therefore start the fixed-point computation of inner actors in the `portFixPoints` function of composite actors in the following cases:

1. Some events from the event queue are passed to some inner actors.
2. An input port of a composite actor is connected to some inner actors and the status of the port is decided (i.e., either received some value or became absent).

In Case 1, when released events are propagated to some inner actor of a composite actor, the `portFixPoints` computation of those inner actors begins. The following equations describe the propagation of `active-evt`s from the event queue to inner actors. If there are *some* events toward an inner actor of a composite actor, then *all* such events are passed to the inner actors and `portFixPoints` of the inner actors is started. This equation is the only equation defined on the sort `Configuration`, so that it is executed before the other `portFixpoints` equations are applied:

```
ceq portFixPoints(active-evt(event((O . AI) ! P, V))
                  < O : CompositeActor | innerActors : OBJECTS > CF)
  = portFixPoints(< O : CompositeActor | innerActors : portFixPoints(MSGS OBJECTS) > CF')
 if fr(MSGS, CF') := filterMsg(O, CF, active-evt(event(AI ! P, V)) ) .
```

The function `filterMsg` separates the events toward inside from the others, and returns a constructor `fr(`*Events*`,`*Conf*`)` which is a pair of the desired events and the other configuration:

```
op filterMsg : Oid Configuration MsgConfiguration ~> FilterResult .
eq filterMsg(O, active-evt(event((O . NAI) ! P, V)) CF, MSGS)
 = filterMsg(O, CF, active-evt(event(NAI ! P, V)) MSGS) .
eq filterMsg(O, CF, MSGS) = fr(MSGS, CF) [owise] .
```

In Case 2, we must define the `portFixPoints` function for the port-propagation of composite actors. An input to a composite actor will lead to an input to one of its subactors, and an output at a subactor will lead to an output from the containing composite actor. (We use the special name '`parent`' to denote the containing actor of an actor in port names.) When a composite actor passes a value (or the knowledge that input will be `absent`) to inner actors, if the inner fixed-point computation has not started yet or is already finished, then `portFixPoints` must again be called to (re-) compute the fixed-point of the inner diagram:

```
ceq portFixPoints(
       < O : CompositeActor | status : enabled,
                              ports : < P : InPort | status : PS, value : V >  PORTS,
                              innerActors :
                                (parent ! P) ==> (O' ! P' ; EPIS)
                                < O' : Actor | ports : < P' : InPort | status : unknown > PORTS2 > REST2 >
       REST)
  =
    portFixPoints(
       < O : CompositeActor | innerActors : portFixPoints(     *** (re-) start the inner fixed-point
                              (parent ! P) ==> (O' ! P' ; EPIS)
                              < O' : Actor | ports : < P' : InPort | status : PS, value : V > PORTS2 >
                              REST2) >
       REST)  if PS =/= unknown .
```

Of course, a composite actor can pass an updated port status/value to its inner actors also when those inner actors are already computing `portFixPoints`; that case is modeled by an equation that is very similar to the above equation and is not shown.

Likewise, an inner actor can propagate the status of output ports to the containing actor. In this case, we only consider when the inner fixed-point is already finished, because in Ptolemy II an inner actor has a higher priority than a parent actor in the evaluation order:

```
ceq portFixPoints(
        < O : CompositeActor |
                ports : < P : OutPort | status : unknown >  PORTS,
                innerActors :
                  (O' ! P') ==> (parent ! P ; EPIS)
                  < O' : Actor | status : enabled,
                                 ports : < P' : OutPort | status : PS, value : V >  PORTS2 > REST2 >
        REST)
  =
    portFixPoints(
        < O : CompositeActor |
                ports : < P : OutPort | status : PS, value : V >  PORTS,
                innerActors : (O' ! P') ==> (parent ! P ; EPIS)
                              < O' : Actor | ports : < P' : OutPort | > PORTS2 > REST2 >
        REST)  if PS =/= unknown .
```

Similarly, if some output port of a composite actor is directly connected to its input port, the status (and the value if the status is present) of the input port is transferred to the output port after the inner fixed-point is finished:

```
ceq portFixPoints(
        < O : CompositeActor | status : enabled,
                               ports : < P : InPort | status : PS, value : V >
                                         < P' : OutPort | status : unknown > PORTS,
                               innerActors : (parent ! P) ==> (parent ! P' ; EPIS) REST2 >
        REST)
=
   portFixPoints(
        < O : CompositeActor | ports : < P : InPort | >
                                       < P' : OutPort | status : PS, value : V > PORTS >
       REST)  if PS =/= unknown .
```

All input and output ports of inner actors in *disabled* composite actors become `absent`, since there is no computation for disabled actors. The `setAllPortsAbsent` function make the status of every port `absent`, including inner actors of composite actors.

```
eq portFixPoints(
       < O : CompositeActor |
          status : disabled,
          innerActors : < O' : Actor | ports : < P : Port | status : unknown > PORTS > REST2 >
       REST)
=
   portFixPoints(
       < O : CompositeActor | innerActors : setAllPortsAbsent(< O' : Actor | > REST2) > REST) .
```

We also have equations setting the output ports of composite actors to `absent` if there are no connections into these ports.

An `owise` equation is again used to end the fixed-point computation when no equation adding new information about the ports can be applied. However, to end the fixed-point computation of a (sub)system,

the fixed-point computations of the subsystems of composite actors must have finished. Therefore, this `owise` equation should only be applied when there is no `portFixPoints` operator in the `innerActors` of the `CompositeActors` in the system. Since `portFixPoints` is declared as a *partial* function, no object with an occurrence `portFixPoint` operator somewhere in its inner actors (or in some subactor of an inner actor) will be a term of sort `Object`. That is, actors of sort `Object` do not contain `portFixPoints`:

```
ceq portFixPoints(OBJECTS) = OBJECTS [owise] .
```

5.3.1 Modal Models

Most of the semantics for modal models is borrowed from the semantics of composite actors, except for frozen actors, coupled ports, and the evaluation order between the controller and refinements. For modal models, `postfire` also sets the `status` attribute of the inner actors according to the current state of the controller to freeze all refinement actors except the refinement of the current state:

```
ceq postfire(
    < O : ModalModel | status : enabled, controller : CO, refinement : REFS, innerActors : CF >)
 =
    < O : ModalModel | innerActors : (< CO : FSM-Actor | > setStateRefinement(STATE, REFS, OBJECTS)) >
 if < CO : FSM-Actor | currStatus : STATE >  OBJECTS  :=  postfire(CF) .
```

The function `setStateRefinement` disables all refinements except the refinement of the current state.

```
op setStateRefinement : Location RefinementSet Configuration -> Configuration .
eq setStateRefinement(STATE, refine-state(STATE', O) REFS, < O : Actor | status : ST > REST)
 = < O : Actor | status : if STATE == STATE' then enabled else disabled fi >
   setStateRefinement(STATE, REFS, REST) .
eq setStateRefinement(STATE, empty, REST) = REST .
```

Notice that, because of the way the other equations are defined, it is not necessary to set the `status` flag to `disabled` in subactors of frozen actors.

If the controller depends on the result of `portFixpoints` of some refinement actors, then the result must be transferred through some coupled input port of the controller actor. Hence the evaluation order between the controller and refinements is automatically treated in our representation. The only part not yet covered is to handle coupled input/output ports in the controller FSM actor of a modal model. In our representation, the coupled input/output ports have the same name, and the value of the input port will be copied only if the coupled output port is *absent*:

```
eq portFixPoints(
       < O : ModalModel | status : enabled,  controller : CO,
                          innerActors :
                            < CO : FSM-Actor | status : enabled,
                                               ports : < P : InPort | status : present, value : V >
                                                       < P : OutPort | status : absent > PORTS >
                             REST2 >
       REST)
 =
   portFixPoints(
       < O : ModalModel | innerActors : portFixPoints(
                            < CO : FSM-Actor |
                                ports : < P : InPort | >
                                        < P : OutPort | status : present, value : V > PORTS >
                            REST2 >)
       REST) .
```

The above equation can be only applied after the inner fixed-point computation triggered by the controller FSM actor has been finished. Therefore, an output port copies a value from its coupled input port only if no value is generated at the output port when the controller is computed.

However, because of the above equation, the absent status of coupled output ports should not be transferred to the parent until we can decide whether the associated coupled input port is absent or not. For this reason we do not explicitly represent the connections between coupled output ports of the controller and the output ports of the parent modal model. Instead, we define the following equations to propagate the value of the coupled output ports:

```
eq portFixPoints(
     < O : ModalModel | status : enabled, controller : CO,
                        ports : < P : OutPort | status : unknown > PORTS,
                        innerActors : < CO : FSM-Actor | ports :
                                            < P : OutPort | status : present, value : V > PORTS2 >
                                        OBJECTS >
      REST)
  =
    portFixPoints(
       < O : ModalModel | ports : < P : OutPort | status : present, value : V > PORTS >
       REST) .
```

The absent status of a coupled output port is propagated only if the associated input port is also absent:

```
eq portFixPoints(
      < O : ModalModel | status : enabled, controller : CO,
                         ports : < P : OutPort | status : unknown > PORTS,
                         innerActors :
                            < CO : FSM-Actor |
                               ports : < P : InPort | status : absent >
                                       < P : OutPort | status : absent > PORTS2 >
                            OBJECTS >
      REST)
  =
    portFixPoints(
        < O : ModalModel | ports : < P : OutPort | status : absent > PORTS >  REST) .
```

6 Extending the Real-Time Maude Semantics to DE Models with Expressions

Ptolemy II provides a language to specify algebraic expressions; such expressions are used to specify the values of parameters, guards, and actions in state machines, and for the computations performed by *expression* actors. The Ptolemy expression language is similar to expression languages in widely used programming languages. An expression can include variables that refer to parameters and input ports. An expression containing references to input ports can be computed when the status of the input ports are known to be either `present` or `absent`.

Parameters whose values are expressions are functions of the values of other parameters that were computed at the previous iteration. Only the value of a parameter that was computed in the previous iteration is used by `portFixPoints` and `postfire`. After `postfire`, the new values of parameters are computed and used in the next iteration.

Unlike parameters, the evaluated value of the expressions in other components (ports, guards, actions, etc.) are immediately used in the current computation step. Such expressions can be evaluated at any time during `portFixPoints` and `postfire`. For example, the *guard* expressions of FSM actors are evaluated during `portFixPoints`, and the *set* actions of FSM actors are computed during `postfire`.

In hierarchical Ptolemy II models, the values of expressions in some actor cannot be easily computed by a simple function such as `computeValue(`*`expr`*`,`*`PARAMS`*`,`*`PORTS`*`)`, because the parameters referred to by some variables may not be included in the actor, but in a composite actor that contains the actor. For that reason, we may need to look at the entire hierarchy of the actor structure to compute expressions. Moreover, the status of some input ports in the expression may be *unknown*, so that the computation may have to be postponed until all input ports in the expression are *present*.

To resolve the above difficulties, we add a *processor* for each actor that computes expressions. Whenever the value of an expression needs to be computed, the *computation configuration* of the expression, which holds all the information for evaluating the given expression, is created in the processor. The value of the expression is then computed inside the processor using the computation configuration, and every information for the evaluation is sent from the outside to the processor. Basically, to evaluate an expression, we need to decide all *free variables* in the expression. Hence, a computation configuration consists of an expression and the assignment map (or variable environment) that contains the values of the free variables in the expression. For each free variable in the variable environment, the corresponding value is transferred into the environment when it is available. The value of a parameter computed at the previous step is transferred, and the value of an input port is transferred when the status of the port becomes *present*.

We can then *independently* define the semantics of the Ptolemy expression language using a computation configuration. Section 6.1 briefly introduces the syntax and the simple denotational semantics of Ptolemy expression language in rewriting logic. Section 6.2 extends our Real-Time Maude semantics of Ptolemy II to models whose parameters, guards, and actions are generic Ptolemy II expressions.

6.1 The Ptolemy Expression Language and its Semantics

Ptolemy II expressions consist of constants, variables, and operators. A constant is a number, a Boolean value, or a string. Variables are references to parameters or ports of actors, and may refer to parameters of composite actors that contains the actors. Operators can be arithmetic (`+`, `-`, `*`, `/`, `^`, `%`), bitwise (`&`, `|`, `#`, `~`), logical (`&&`, `||`, `!`, `&`, `|`), shift (`<<`, `>>`, `>>>`), or conditional (*condition* `?` exp_1 `:` exp_2).

The Ptolemy II expression language provides functional expressions. A functional expressions is either a method call $object.method(arg_1, \ldots, arg_n)$ on objects or a general function call $function\_name(arg_1, \ldots, arg_n)$. It is possible to define a function in a way similar to lambda calculus: `function(`arg_1`:`$Type_1$`, ..., `arg_n`:`$Type_n$`)` *function_body_expression*. In addition, the expression language includes a set of built-in methods and functions, such as `sin()`, `cos()`, etc.

The Ptolemy II expression language also supports composite data types such as arrays, records, and matrices. Arrays are lists of expressions in curly brackets, e.g., `{1, 2.0, "x"}`. Records are lists of fields where each field consists of a name and a value. For example, `{a=1, b="foo"}` is a record with two fields, named `a` and `b`, with values `1` and `"foo"`, respectively. A matrix data structure in Ptolemy II expression language describes a usual $n \times m$ matrix. Matrices are specified with square brackets, using commas to separate row elements and semicolons to separate rows, e.g., `[1, 2 ; 3, 4]` represents the matrix $\left(\begin{smallmatrix} 1 & 2 \\ 3 & 4 \end{smallmatrix}\right)$.

6.1.1 The Real-Time Maude Representation of Ptolemy II Expressions

Our Real-Time Maude semantics supports the entire expression language described above. However, in the following presentation, we explain only how we deal with constants, variables, the built-in operators mentioned above, conditionals, and arrays.

In our representation, Ptolemy II expressions are terms of sort `Exp`. A *value* is an expression that cannot be further evaluated; such values are represented as terms of the sort `Value`, which is a subsort of `Exp`. Ptolemy variables are terms of sort `VarId` in our semantics. Constants have sort `Value`, and are represented by the corresponding values in Real-Time Maude, prefixed with the `#` symbol. Numerical constants are either rational numbers (which contain the integers) or fixed-point constants. Operators (unary, binary, and conditional) are defined by Real-Time Maude operator declarations as follows:

```
ops -_ ~_ !_ : Exp -> Exp .                    --- negative, complements, negation
```

```
ops _+_ _-_ _*_ _/_ _%_ _^_ : Exp Exp -> Exp .        --- numerical binary operators
...
op _?_:_ : Exp Exp Exp -> Exp [ctor prec 60] .        --- the conditional operator
```

The algebraic semantics of each operator is defined as usual way. For example, the conditional expression is defined as follows (we first declare all the variables used in this section):

```
var O : Oid . var AI : ActorID . var ECF : [Configuration] . var EVTS : Events . var QUEUE : EventQueue .
vars SYSTEM STABLEPORTS STABLEPARAMS STABLEACTORS : StableConfiguration .
vars OBJECTS REST PORTS PARAMS : ObjectConfiguration . var P : PortId . var N : Nat .
var RI : ParamId . var VI : VarId . var VIS : VarIdSet . var CI : ComputationID . var ENV : EnvMap .
vars V V' : Value . vars E E1 E2 E3 : Exp . var EL : ExpList . var PE : ProperExp .

eq # true ? E1 : E2 = E1 .
eq # false ? E1 : E2 = E2 .
```

Arrays are represented by an `ExpList` enclosed by the operator `{_}`. Arrays have some methods and functions to manipulate them. The method *length*() returns the size of an array, and a functional expression {*array*}(i) returns the i-th element of the array (starting from 0):

```
eq  { E, EL }(# 0)  = E .   --- array access
ceq { E, EL }(# N) = { EL }(#(N - 1)) if N > 0 .

eq { () } .. 'length(())    = # 0 .   --- array length
eq { E, EL } .. 'length(()) = # 1 + ({ EL } .. 'length(())) .
```

In addition, we define a sort `ProperExp` for the expressions that are not values. All expressions that can be further evaluated are defined as `ProperExp`:

```
subsorts VarId < ProperExp < Exp                              --- variables
ops -_ ~_ !_ : ProperExp -> ProperExp .                       --- negative, complement, negation
ops _+_ _-_ _*_ _/_ _%_ _^_ : ProperExp Exp -> ProperExp .    --- numerical binary operators
ops _+_ _-_ _*_ _/_ _%_ _^_ : Exp ProperExp -> ProperExp .    --- numerical binary operators
...
op {_} : ProperExpList -> ProperExp [ctor] .                  --- array containing proper exp
```

6.1.2 Rewriting Semantics of Ptolemy II Expression Language

The semantics of the Ptolemy II expression language is defined on a computation configuration of an expression. A computation configuration is either a pair of an expression and a variable environment that holds all free variables in the expression, or the value of the evaluation result:

```
sorts ComputationConfig ConfigItem .
op result : Value -> ComputationConfig [ctor] .
op __ : ConfigItem ConfigItem -> ComputationConfig [ctor comm] .
op exp : Exp -> ConfigItem [ctor] .
op env : EnvMap -> ConfigItem [ctor] .
```

With the denotational style of the language semantics, the expression is evaluated when the values of all free variables in the environment are decided:

```
op [[_]]_ : Exp EnvMap ~> Value .
ceq exp(E) env(ENV) = [[ E ]] ENV if allFreeVariableDecided(ENV) .

eq [[ VI ]] (VI <-| V ; ENV) = V .
eq [[ E1 + E2 ]] ENV = [[ E1 ]] ENV + [[ E2 ]] ENV .
...
eq [[ E1 ? E2 : E3 ]] ENV = [[ E1 ]] ENV ? [[ E2 ]] ENV : [[ E3 ]] ENV .
```

6.2 Real-Time Maude Semantics of Ptolemy II DE Models with Generic Expression

We extend the `Actor` class with an additional attribute `computation` to model the processor of an actor, in which expressions are evaluated:

```
class Actor | ports : Configuration, parameters : Configuration,
              status : ActorStatus,  computation : Computation .
```

The sort `Computation` is either `noComputation` or a *computation configuration* tagged with an identifier.

```
sorts Computation ComputationID ComputationConfig .
op noComputation : -> Computation [ctor] .
op _/_ : ComputationID ComputationConfig -> Computation [ctor] .
```

Parameters are now objects with three attributes; `exp`, `value` and `next-value`. The attribute `exp` has the expression of a parameter, and the `value` attribute has the current value of `exp` that was computed in the previous iteration. The `next-value` attribute contains the value that will be used at the next computation step. It is initially `noValue`, and becomes `computing` when the `exp` attribute is computing at the current computation step.

```
class Parameter | exp : Exp, value : Value, next-value : Value? .

sort Value? .  subsort Value < Value? .
ops noValue computing : -> Value? [ctor] .
```

6.2.1 Computing Expressions with the Computation Configuration

Whenever some expression needs to be evaluated, the computation configuration for the expression is created in the `computation` attribute. When creating a computation configuration, the variable environment of an expression is constructed from the free variables of the expression. The function `freeVars(`*`PE`*`)` returns the set of all free variables in the expression, and the `constructEnv` creates the assignment map from those free variables, where each variable is initially set to be *unknown* (denoted by *`NAME`* `<-?`):

```
op constructEnv : Exp -> EnvMap .
eq constructEnv(E) = constructEnv(freeVars(E)) .

op constructEnv : VarIdSet -> EnvMap .
eq constructEnv(VI ; VIS) = (VI <-?) ; constructEnv(VIS) .
eq constructEnv(none) = empty .
```

For example, if some output port has status `present` but has non-value expression (i.e., `ProperExp`), the configuration for the expression is created to evaluated it, and the resulting value is plugged back into the output port:

```
eq < O : Actor | ports : < P : OutPort | status : present, value : PE > PORTS,
                 computation : noComputation >
 = < O : Actor | computation : #port(P) / exp(PE) constructEnv(PE) > .

eq < O : Actor | ports : < P : OutPort | status : present > PORTS,
                 computation : #port(P) / result(V) >
 = < O : Actor | ports : < P : OutPort | value : V > PORTS, computation : noComputation > .
```

Similarly, parameters are computed when the `next-value` is computing, and the result value will be written to the `next-value`.

```
eq < O : Actor | parameters : < RI : Parameter | exp : E, next-value : computing > PARAMS,
                 computation : noComputation >
 = < O : Actor | computation : #param(RI) / exp(E) constructEnv(E) > .

eq < O : Actor | parameters : < RI : Parameter | next-value : computing > PARAMS,
                 computation : #param(RI) / result(V) >
 = < O : Actor | parameters : < RI : Parameter | next-value : V > PARAMS,
                 computation : noComputation > .
```

For each *unknown* free variable in the variable environment, the corresponding value is transferred when it is available. The value of an input port is transferred when the port becomes *present*:

```
eq < O : Actor | ports : < P : InPort | status : present, value : V > PORTS,
                 computation : CI / env(P <-?   ; ENV) exp(E) >
 = < O : Actor | computation : CI / env(P <-| V ; ENV) exp(E) > .
```

Similarly, the value of the parameter is transferred:

```
eq < O : Actor | parameters : < RI : Parameter | value : V > PARAMS,
                 computation : CI / env(RI <-?   ; ENV) exp(E) >
 = < O : Actor | computation : CI / env(RI <-| V ; ENV) exp(E) > .
```

If a variable in an expression refers to a parameter higher in the actor containment hierarchy, this hierarchical scope is handled using messages by the similar way with the event handling in composite actors. If a variable is not available in the current environment, query this variable to the parent actor by a message `query-var`:

```
ceq < O : Actor | computation : CI / env(RI <-?        ; ENV) exp(E) >
  = < O : Actor | computation : CI / env(requesting RI ; ENV) exp(E) > query-var(O, VI)
 if not RI in ENV .
```

If the variable is not available in the current composite actor, then the message is passed to its parent. Otherwise, the corresponding value is returned by another message `return-var` as follows:

```
eq < O : CompositeActor | parameters : < RI : Parameter | value : V > PARAMS,
                          innerActors : query-var(AI, RI) ECF >
 = < O : CompositeActor | innerActors : return-var(AI, RI, V) ECF > .
```

And the returned value is plugged into the variable environment:

```
eq < O : Actor | computation : CI / env(requesting RI ; ENV) exp(E) > return-var(O, RI, V)
 = < O : Actor | computation : CI / env(RI <-| V      ; ENV) exp(E) > .
```

During `portFixPoints` and `postFire`, such messages can freely move between different hierarchies:

```
eq portFixPoints(query-var(AI, VI) ECF) = query-var(AI, VI) portFixPoints(ECF) .
eq postfire(query-var(AI, VI) ECF) = query-var(AI, VI) postfire(ECF) .
eq return-var(AI, VI, V) portFixPoints(ECF) = portFixPoints(return-var(AI, VI, V) ECF) .
eq return-var(AI, VI, V) postfire(ECF) = postfire(return-var(AI, VI, V) ECF) .
```

Note that the variable `ECF` in the above equations is defined at the *kind* level so that those equations can be applied when `portFixPoints` and `postFire` is executed further down in the hierarchy.

All computations should be finished before computing the next semantics function, and before advancing to the next computation step. To ensure this, we introduced new sorts `StableObject` and `StableConfiguration`. An actor is `StableObject` only if there is no (possible) ongoing computation, defined by the following membership equations:

```
mb (< P : Port | value : V >) : StableObject .
mb (< RI : Parameter | next-value : noValue >) : StableObject .
mb (< RI : Parameter | next-value : V >) : StableObject .

mb (< O : AtomicActor | ports : STABLEPORTS, parameters : STABLEPARAMS >) : StableObject .
mb (< O : CompositeActor | innerActors : STABLEACTORS,
                           ports : STABLE-PORTS, parameters : STABLEPARAMS >) : StableObject .
```

Object configurations are `StableConfiguration` if all objects in them are stable objects. The rewrite rule `executeStep` is only applied when all actors are stable objects.

6.2.2 Actors with Generic Expression

Using the mechanism defined in the previous section, the semantics of actors with expressions can be easily defined. For example, expression actors can compute a general expression that may use the values of input ports. An expression actor has an output port `output` and may have several input ports. It has also the additional attribute `expression` for an expression.

The `portFixPoints` of expression actors are straightforward and very similar to the case for ports and parameters. If the output port is unknown, then the configuration for the expression is created and the output port will have the evaluated value of the expression.

```
eq portFixPoints(< O : Expression | expression : E,
                                    ports : < 'output : OutPort | status : unknown > PORTS,
                                    computation : noComputation > REST)
 = portFixPoints(< O : Expression | computation : #port('output) / exp(E) constructEnv(E) > REST) .

eq portFixPoints(< O : Expression | ports : < 'output : OutPort | status : unknown > PORTS,
                                    computation : #port('output) / result(V) > REST)
 = portFixPoints(< O : Expression | ports : < 'output : OutPort | status : present, value : V > PORTS,
                                    computation : noComputation > REST) .
```

FSM actors may have general expressions in their guards, output actions, and set actions. The semantics of FSM actors is similar to the above cases. During `portFixPoints`, all appropriate guard expressions are set to be computed in the `computation` attribute. If only one guard expression is evaluated to `true`, the expressions in the output actions of the transition is transferred to the related output ports, and the expressions in the ports are computed by the expression semantics of ports. Similarly, the guard expressions are computed again during `postfire`, and the set actions of the enabled transition is transferred to the `exp` attributes of the corresponding parameters, and the `next-value` attributes are set to `computing`. Then the expression semantics of parameters computes those expressions and all `next-value` attributes will eventually have the evaluated values.

6.2.3 Parameter Computation in Computation Step

A parameter with generic expressions is a function of the values of the other parameters which are computed in the previous iteration. If a parameter is changed during *postfire* (e.g., set actions of FSM actors), the `exp` and the `next-value` attributes are updated. Otherwise, the next values of all parameters need to be computed after *postfire*. Also, the value in the `next-value` attribute is transferred to the `value` attribute before starting the next computation step. Therefore, the `executeStep` rule is modified as follows:

```
rl [executeStep] :
   {< global : EventQueue | queue : (EVTS ; 0 ; 0) :: QUEUE > SYSTEM}
  =>
   {< global : EventQueue | queue : QUEUE >
    update(computeNextParams(postfire(portFixPoints(releaseEvt(EVTS) clearPorts(SYSTEM)))))} .
```

The `computeNextParams` function initiates the computation of the next values of parameters if they are not computed yet, and overwrites the `exp` attribute if they are changed during `postfire`.

```
op computeNextParams : Configuration ~> Configuration .
eq computeNextParams(< O : AtomicActor | parameters : PARAMS >)
 = < O : AtomicActor | parameters : computeNextParams(PARAMS) > .
eq computeNextParams(< O : CompositeActor | parameters : PARAMS, innerActors : OBJECTS >)
 = < O : CompositeActor | parameters : computeNextParams(PARAMS),
                          innerActors : computeNextParams(OBJECTS) > .

eq computeNextParams(< RI : Parameter | exp : E, next-value : noValue > PARAMS)
 = < RI : Parameter | next-value : computing > computeNextParams(PARAMS) .
 eq computeNextParams(< RI : Parameter | next-value : V > PARAMS)
 = < RI : Parameter | exp : V > computeNextParams(PARAMS) .
eq computeNextParams(none) = none .
```

The `update` function updates the value of the parameters, and clears the `next-value` attribute.

```
op update : Configuration ~> Configuration .
eq update(< O : AtomicActor | parameters : PARAMS >)
 = < O : AtomicActor | parameters : updateParams(PARAMS) > .
eq update(< O : CompositeActor | parameters : PARAMS, innerActors : OBJECTS >)
 = < O : CompositeActor | parameters : updateParams(PARAMS), innerActors : update(OBJECTS) > .

op updateParams : Configuration ~> Configuration .
eq updateParams(< RI : Parameter | value : V, next-value : V' > PARAMS)
 = < RI : Parameter | value : V', next-value : noValue > updateParams(PARAMS) .
eq updateParams(none) = none .
```

7 Formal Verification of Ptolemy II DE Models in Ptolemy II

This section first briefly explains how the Ptolemy II code generation infrastructure has been used to both automatically synthesize a Real-Time Maude verification model from a Ptolemy II DE design model, and to integrate the verification of the synthesized model into Ptolemy II. We then explain how the synthesized model can be easily model checked from within Ptolemy II.

Ptolemy II gives the user the possibility of adding a "code generation button" to a (top-level) Ptolemy II model. When the blue `RTMaudeCodeGenerator` button in a Ptolemy II DE model is double-clicked, Ptolemy II opens a dialog window which allows the user to start code generation and to give simulation and model checking commands to execute and formally analyze the generated code. After clicking the `Generate` button in the dialog window, the generated Real-Time Maude code and the result of executing the analysis commands are displayed. Figure 6 shows the dialog window for the flat traffic light system in Section 3.7. The two temporal logic properties discussed below have been entered into the window. The `Generate` button has already been clicked and the results of model checking those properties are displayed in the "Code Generator Commands" box.

As mentioned in Section 2, the synthesized Real-Time Maude verification model can be analyzed in different ways. This paper focuses on linear temporal logic (LTL) model checking.

In Real-Time Maude, an LTL formula is constructed from a set of (possibly parametric) *atomic state propositions* and the usual Boolean and LTL operators. Having to define such state propositions makes the verification process nontrivial for the Ptolemy user, since it requires some knowledge of the Real-Time Maude representation of the Ptolemy model, as well as the ability to define functions in Real-Time Maude. To free the user from this burden, we have predefined several generic atomic propositions for Ptolemy II models. For example, the proposition

$$actorId \ \mid \ var_1 = value_1 \, , \ldots , \, var_n = value_n$$

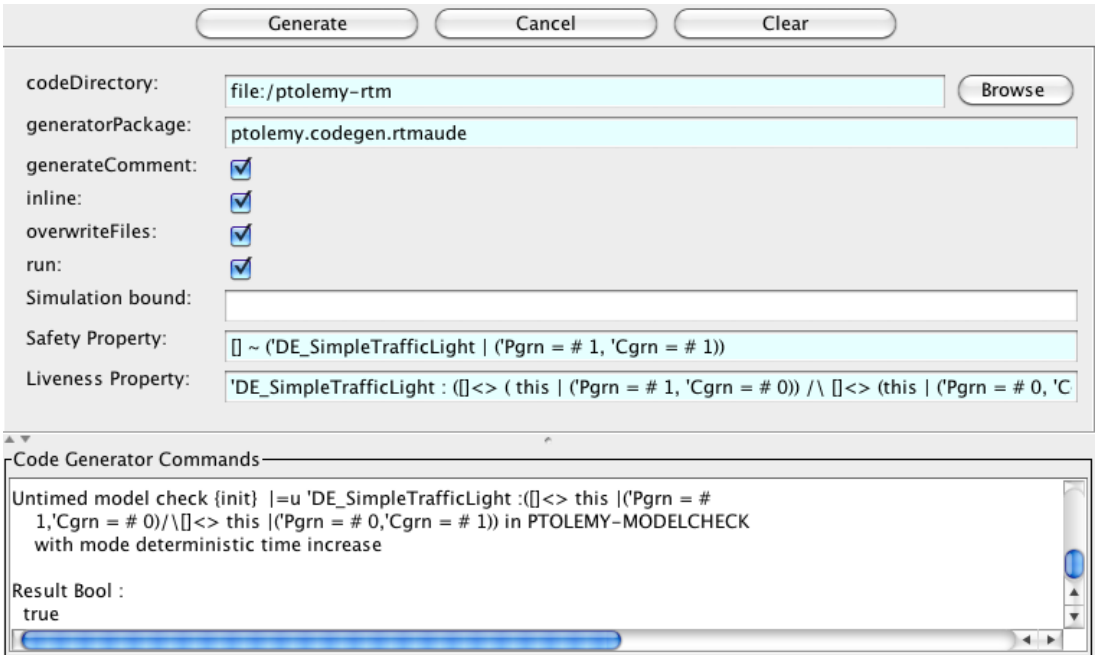


Figure 6: Dialog window for the Real Time Maude code generation

holds in a state if the value of the parameter var_i of an actor equals $value_i$ for each $1 \leq i \leq n$, where *actorId* is the *global actor identifier* of a given actor. Similarly, the propositions

actorId `|` `port` *p* `is` *value* *actorId* `|` `port` *p* `is` *status* *actorId* `?` *boolean_expression*

hold if, respectively, the port *p* of actor *actorId* has the value *value*, the port *p* has status *status*, or the given boolean expression *boolean_expression* is evaluated to `true`.

For FSM actors and modal models, the proposition

actorId `@` *location*

is satisfied if and only if the actor with global name *actorId* is in location (or "local state") *location*.

The semantics of the above atomic propositions is defined as explained in Section 2.3. In particular, the proposition `_@_` for locations is defined by:

```
eq {< O : FSM-Actor | currState : L > CF} |= O @ L = true .
eq {< O : ModalModel | controller : CO, innerActors : ACTS > CF} |= O @ L = {ACTS} |= CO @ L .
eq {< O : CompositeActor | innerActors : OBJECTS > CF} |= (O . AI) @ L = {OBJECTS} |= AI @ L .
eq {< O : Actor | > CF} |= O @ L = false [owise] .
```

The definitions of atomic propositions for parameters and ports are similar.

An LTL formula may contain multiple occurrences of atomic propositions. To avoid having to write long global actor names too many times, we can simplify a formula with *actor scope*, so that

actorId `:` *formula*

denotes that *formula* should hold in the actor with the global identifier *actorId*. For example, the formula $o_1 \,.\, o_2$ `:` `[]` $(o_3$ `@` l_1 `/\` $o_4 \,.\, o_5$ `@` $l_2)$ equals the formula `[]` $(o_1 \,.\, o_2 \,.\, o_3$ `@` l_1 `/\` $o_1 \,.\, o_2 \,.\, o_4 \,.\, o_5$ `@` $l_2)$.

Consider the flat traffic light system given in Section 3.7, where each traffic light is represented by set of variables. The safety property we want to verify is that it is never the case that both the car light and the pedestrian light show green at the same time. If the name of the model is `'DE_SimpleTrafficLight`, then `('DE_SimpleTrafficLight | ('Pgrn = # 1, 'Cgrn = # 1))` holds in all states where the `Pgrn` and `Cgrn` variables both have the value 1. The safety property we are interested in, that such a state can *never* be reached, can be defined as the LTL formula

```
[] ~ ('DE_SimpleTrafficLight | ('Pgrn = # 1, 'Cgrn = # 1))
```

Alternatively, the LTL formula

```
[] ~ 'DE_SimpleTrafficLight : ('CarLightNormal @ 'Cgrn /\ 'PedestrianLightNormal @ 'Pgreen)
```

states that it is never the case that the `CarLightNormal` FSM actor is in local state `Cgrn` when the `PedestrianLightNormal` actor is in local state `Pgreen`.

We can also check the liveness property that both pedestrian and cars can cross infinitely often. That is, it is infinitely often the case that the pedestrian light is green when the car light is *not* green, and it is also infinitely often the case that the car light is green when the pedestrian light is not green:

```
'DE_SimpleTrafficLight : ([]<>(this | 'Pgrn = #1, 'Cgrn = #0) /\ []<>(this | 'Pgrn = #0, 'Cgrn = #1))
```

8 Case Studies

This section presents three Ptolemy II discrete-event models and shows how they have been verified in Real-Time Maude from within Ptolemy II. Section 8.1 presents the benchmark railroad crossing example, Section 8.2 presents a hierarchical model of a fault-tolerant traffic light system, and Section 8.3 presents an assembly line due to Misra [29].

8.1 Railroad Crossing

In the benchmark railroad crossing example, a gate at the intersection of the train track and a road should be lowered when a train is in the intersection. Figure 7 shows a Ptolemy II DE model `RailroadSystem` of such a system. This model consists of two finite state machine (FSM) actors: a `Train` actor that models trains, and a `Gate` actor that controls the gate. In addition, the model has Boolean variables `Tin` (which is 1 when a train is in the intersection), `Tleave` (which is 1 when a train is leaving), `Tapproaching`, and `Gopen` (which is 1 when the gate is open). State changes are triggered by a `Clock` actor. These variables are set by signals from the output ports of the train and the gate controller.

The `Train` actor has five states (or locations), and a local variable `distance` denoting the distance between the train and the beginning of the intersection. The `Train` has one input port `Sec`, and three output ports `Tin`, `Tleave`, and `Tapproaching`. Initially, the state is in location `Tinit`. In the first step, a new train is arriving, but is yet `far` away at a distance -10. The FSM actor stays in location `far` as long as the `distance < -3`. The value of `distance` increases by 2 each time there is input in the `Sec` port (that is, each time unit in our case) as long as the train is state `far`. When `distance` has reached -3, the train takes a transition to location `approaching`, where it stays until the `distance` reaches 0. At the same time, it outputs a signal with value 1 through its `Tapproaching` output port. A train that is approaching the intersection slows down; therefore, the distance only increases by one for each time unit in location `approaching` (as well as in locations `within` and `leaving`). When the `distance` to the intersection is 0, the actor goes to state `within`, and emits a signal through its `Tin` port. When the `distance` is greater than or equal to 3, the train is `leaving` the intersection, and an output is emitted through the `Tleave` port. Finally, when the `distance` becomes greater than or equal to 10, the train disappears and a signal with value 0 is output through all three output ports. The actor goes to location `far` and the next train is seen in the horizon, and the `distance` is set to -10.

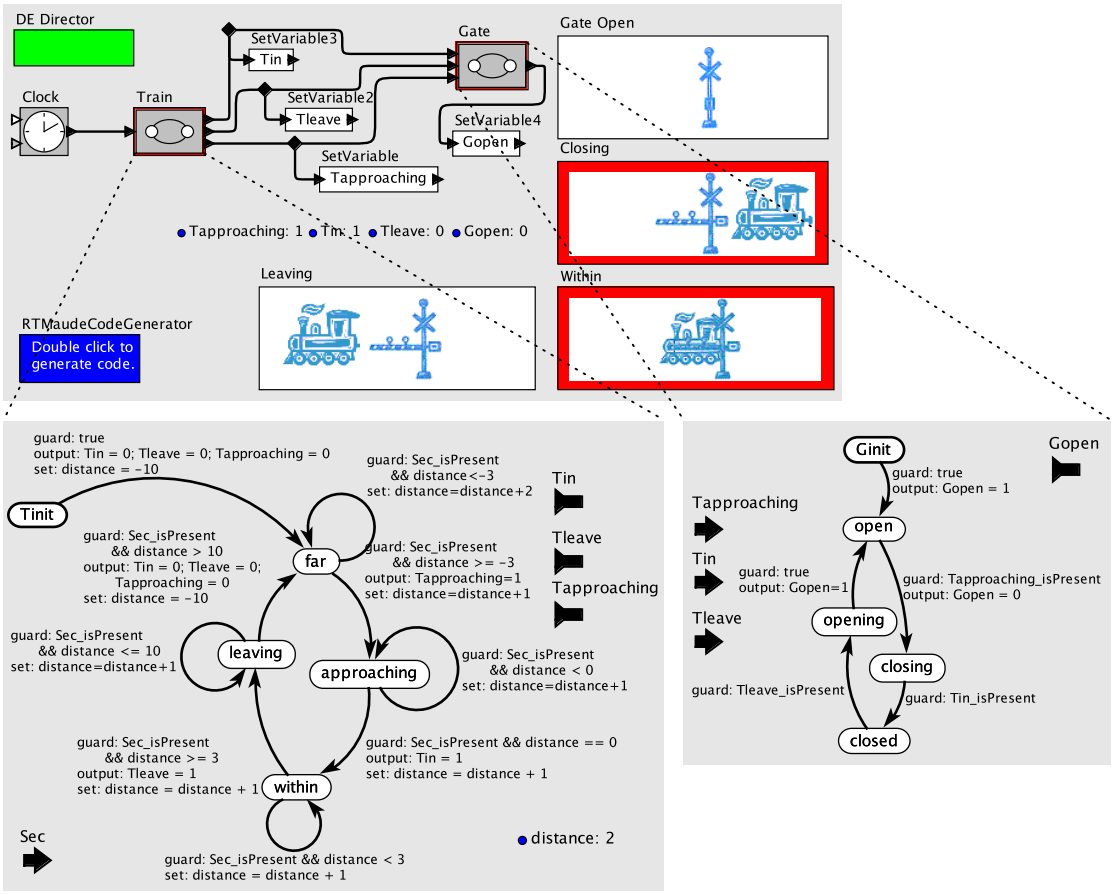


Figure 7: Ptolemy II DE model of the railroad crossing.

The `Gate` actor responds to input from the `Train` actor through its `Tapproaching`, `Tin`, and `Tleave` input ports by the necessary signal through its `Gopen` output port.

The main property that `RailroadSystem` must satisfy is the safety property that whenever a train is in the intersection, the gate must be closed. In our model, a train is in the intersection when the `'Train` actor is in location `'within`, and the gate is closed when the `'Gate` actor is in location `'closed`. Using the propositions defined in Section 7, the proposition `('RailroadSystem . 'Train @ 'within)` and `('RailroadSystem . 'Gate @ 'closed)` hold in these cases, respectively. We want to verify that it is *always* the case that the former *implies* the latter. In temporal logic, this can be given by the formula:

```
[] (('RailroadSystem . 'Train @ 'within) -> ('RailroadSystem . 'Gate @ 'closed))
```

Verification of this property through the Real-Time Maude code generation and analysis interface in Ptolemy II yielded the expected result `true`, proving that the desired property is satisfied in this Ptolemy model.

In addition, we have verified the following *time-bounded* property that says that it is always the case that the `Train` actor will reach the state `within` within 7 time units from the start of system execution:

```
<> ('RailroadSystem . 'Train @ 'within) in time <= 7
```

The execution of each verification command in this case study took less than one second on a 2.4 GHz Intel Core 2 Duo processor.

8.2 Hierarchical Traffic Light

This section describes the verification of the *hierarchical* Ptolemy II DE model in [30] that extends the flat pedestrian crossing system described in Section 3.7 to a fault-tolerant traffic light system consisting of one car light and one pedestrian light.

Figure 8 shows the model. The FSM actor `Decision` "generates" failures and repairs by alternating between staying in location `Normal` for 15 time units and staying in location for `Abnormal` for 5 time units. Whenever the actor takes a transition with target `Normal`, it sends a signal through its `Ok` port, and whenever it reaches, or stays in, location `Abnormal`, the actor sends a signal through its `Error` port. `TrafficLight` is a *modal model*; whenever it is in `error` mode and receives a signal through its `Ok` port, the actor goes to `normal` mode, and vice versa when it receives an `Error` event in `normal` mode. The FSM actor that refines the `error` mode of `TrafficLight` has three states. In this mode, all lights are turned off (by sending a value 0 through the corresponding port), except for the yellow light of the car light, which is blinking. The refinement of the `normal` mode in `TrafficLight` is the composite actor that consists of the two FSM actors `CarLight` and `PedestrianLight`, that define the behavior of the two lights during normal operations, and that were explained in Section 3.7. As before, `Pred`, `Pgrn`, `Cred`, `Cyel`, and `Cgrn` are variables that denote the current color(s) (if any) of the lights. Finally, the `Clock` actor produces an event every time unit.

The main properties we have verified are the safety and liveness properties described in Section 7:

```
[] ~ ('HierarchicalTrafficLight | ('Pgrn = # 1, 'Cgrn = # 1) )

'HierarchicalTrafficLight :
    ([] <> (this | 'Pgrn = # 1, 'Cgrn = # 0) /\ [] <> (this | 'Pgrn = # 0, 'Cgrn = # 1))
```

Finally, if some error has occurred (i.e., the decision actor generates an error), then no traffic light is green until the error is repaired (i.e., the decision actor generates `Ok`):

```
'HierarchicalTrafficLight : (
  [] ('Decision | (port 'Error is present) ->
    <> (this | ('Pgrn = # 0, 'Cgrn = # 0) U 'Decision | (port 'Ok is present))))
```

The execution of each verification command took around seven seconds in this case study.

8.3 Assembly Line

Finally, we have simulated in Real-Time Maude the "assembly line" example of Misra [29] given in Fig. 9. Here, an "advanced" clock `Jobs` generates a set of *jobs* at certain times. The timed plotter `JobArrivedTime` records the actual times (obtained through the `CurrentTime` actor) when the jobs arrived.

Each job has to be executed in three different ways (at `Station1`, `Station2`, and `Station3`). First, a job gets assigned the time it takes to execute the first task of the job. This is done by the `Ramp` actor `ServiceTimes1`. The actual "wait" is first done at the *noninterruptible timer* `Station1`. The point of using a noninterruptible timer is that the count down does not start if some other job is serviced. This can be compared to a gas station. It takes so and so long to fill up the gas tank of your car, but if someone else is already pumping gas, you must also wait for that car to stop pumping and to drive away. After finishing the first part of the job, the job is then assigned a duration of the second part in the ramp `ServiceTimes2`, and waits accordingly at the noninterrruptible timer `Station2`. Finally, when that wait is over, the process repeats for the third part of the task. The timed plotter `StationsFinishedTimes` records the times when jobs finish executing the first, the second, and the third "part" of the jobs.

To simulate the system up to time t in Real-Time Maude, we just write the time bound t in the `Simulation bound` item of the dialog window (see Fig. 6). The output shows the final state, where the `'StationsFinishedTimes` object shows the times when events happened at the different ports:

```
Result ClockedSystem :
< 'AssemblyLine : CompositeActor |
```

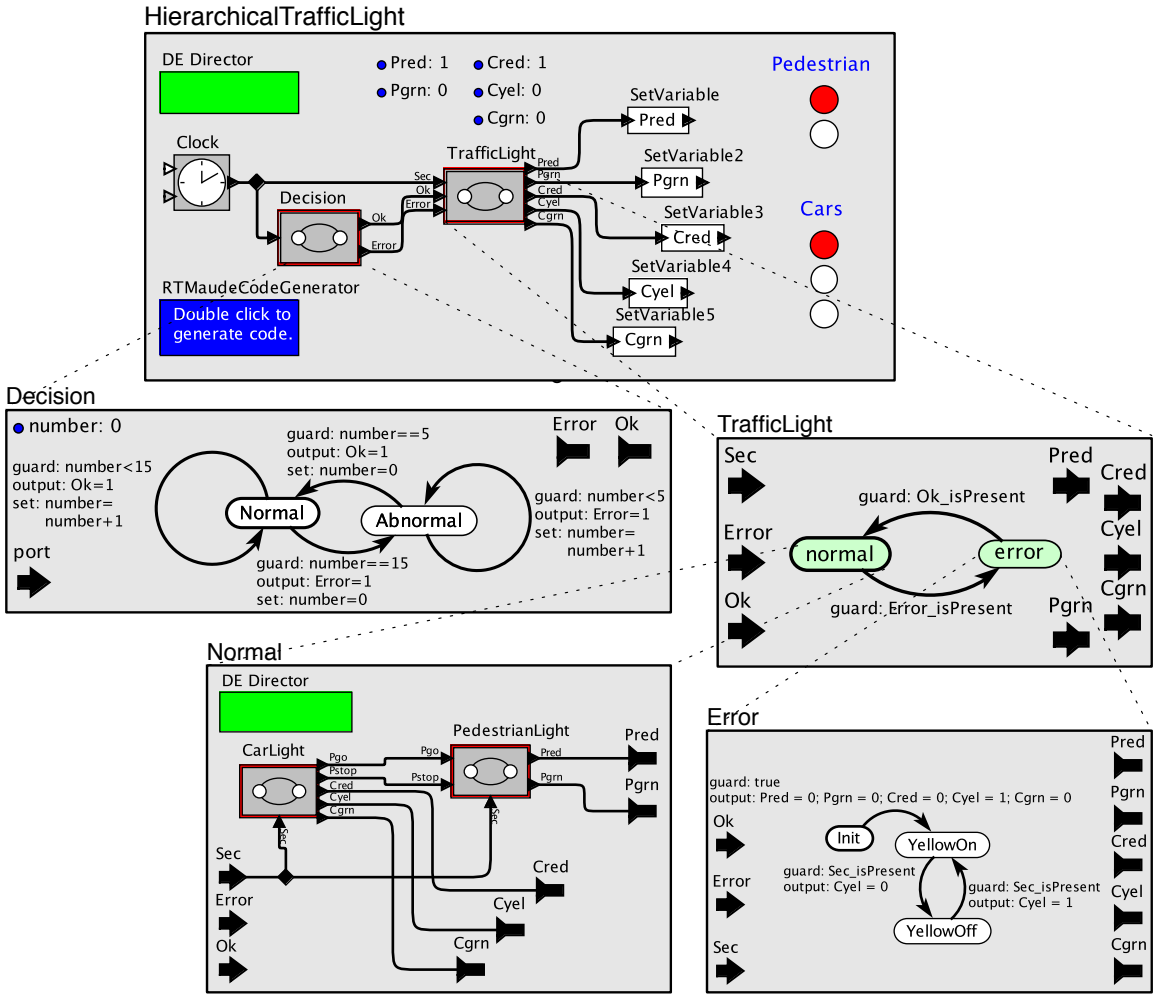


Figure 8: A hierarchical fault-tolerant traffic light system.

```
innerActors : (
  < 'StationsFinishedTimes : TimedPlotter |
    currentTime : 49,
    event-history :
      (source: 'Station1 ! 'output time:  9 value: # 1) ++
      (source: 'Station1 ! 'output time: 19 value: # 1) ++
      (source: 'Station2 ! 'output time: 21 value: # 2) ++
      (source: 'Station3 ! 'output time: 23 value: # 3) ++
      (source: 'Station1 ! 'output time: 31 value: # 1) ++
      (source: 'Station2 ! 'output time: 36 value: # 2) ++
      (source: 'Station1 ! 'output time: 37 value: # 1) ++
      (source: 'Station2 ! 'output time: 38 value: # 2) ++
      (source: 'Station3 ! 'output time: 39 value: # 3) ++
      (source: 'Station3 ! 'output time: 40 value: # 3) ++
      (source: 'Station2 ! 'output time: 45 value: # 2) ++
```

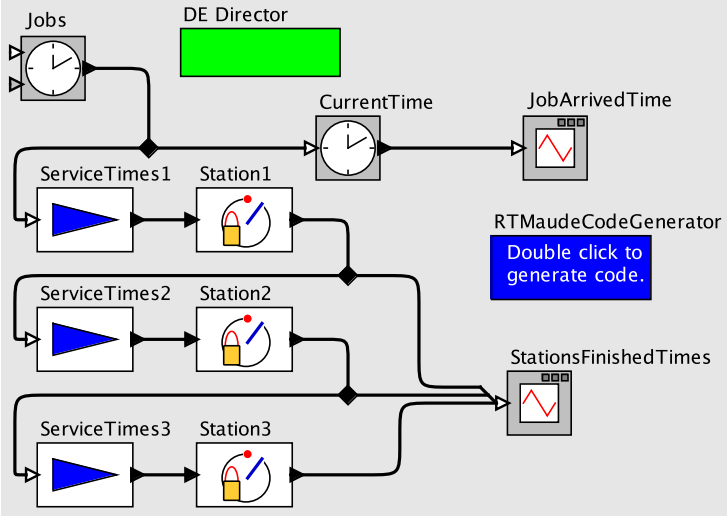


Figure 9: The assembly line example.

```
      (source: 'Station3 ! 'output time: 49 value: # 3),
    parameters : none, computation : noComputation, status : enabled >
  ...),
 parameters : none, ports : none, status : enabled, computation : noComputation >
< global : EventQueue | queue : nil > in time 49
```

For example, we see that `Station2` finish each job at time 21, 36, 38 and 45, respectively. These results are the same as the results shown in the Ptolemy II timed plotters after the Ptolemy II executions.

9 Related work

As mentioned in the introduction, this paper is a significantly extended version of an earlier conference paper [19] and an earlier workshop paper [20]; the former defines a Real-Time Maude semantics for *flat* Ptolemy II DE models and the latter proposes an extension to hierarchical models. Apart from providing much more detail about the semantics, this paper both extends the previous semantics to handle complex Ptolemy II expressions and also describes two additional case studies.

The semantics of Ptolemy II is often given in terms of *abstract semantics*, which consists of a set of functions such as "initialize", "fire", "postfire", etc., that actors are free to implement in different ways [13, 31]. Denotational semantics of DE models based on metric spaces are given in [32, 33, 34]. A different type of denotational semantics, based on complete partial orders and domain theory, are given in [35, 36]. The semantics proposed in [36] is however different from the semantics implemented in Ptolemy II. Obviously, these semantics differ a lot from ours, e.g., in that they are not executable and therefore cannot be used for formal model checking analyses.

A preliminary exploration of translations of *synchronous reactive* (i.e., untimed) Ptolemy II models into Kripke structures, that can be analyzed by the NuSMV model checker, and of DE models into communicating timed automata is given in [37]. However, they require *data abstraction* to map models into finitary automata, and they do not use the code generation framework.

In the context of model transformations of embedded systems, [38] describes a method to automatically translate discrete-time Simulink models to programs written in the synchronous language Lustre [39]. Discrete-time Simulink and Lustre are close to the SR (synchronous-reactive) model of computation of Ptolemy II, but quite different from DE, e.g., SR models lack a notion of quantitative time. [40] describes a method to automatically translate Stateflow models to Lustre. Stateflow is Simulink's hierarchical state

machine notation, visually akin to Statecharts [41], but with different semantics. Automatic translation of more general Simulink/Stateflow models to hybrid automata [42], using a different technique of graph transformations is described in [43]. Key in this technique is the use of metamodels to specify the source and target models, as well as the transformation rules [3]. This type of model transformation is different from the code generation technique used in this paper, which is an extension of the methods described in [44]. The works [38, 40, 43] can also be seen as giving formal semantics to Simulink/Stateflow, via Lustre or hybrid automata. A direct approach to giving formal semantics to Stateflow is described in [45].

On the other hand, Maude has been used to give semantics to a wide range of programming and modeling languages (see, e.g., [46, 47]). And, as mentioned in the introduction, Real-Time Maude has been used to define the semantics of an array of real-time modeling languages [7, 8, 9, 10, 11, 12], but we are not aware of any translation of a synchronous real-time language into Maude or Real-Time Maude.

10 Concluding Remarks

This paper has explained how we have formalized in Real-Time Maude the semantics of a large subset of Ptolemy II DE models. This is a challenging task, since Ptolemy II DE models combine a fixed-point synchronous semantics with hierarchical structure, explicit time, and a rich expression language. The expressiveness of Real-Time Maude is necessary to define this semantics, including the use of unbounded data structures, nested objects, and advanced membership equational logic features such as partial functions and the '`owise`' construct. An additional contribution of our work is the clarification of the semantics of modal models, for which we have given a composite-actor semantics in Ptolemy II.

We have leveraged Ptolemy II's adapter code generation infrastructure to automatically generate Real-Time Maude code from a Ptolemy II DE model. Furthermore, we have integrated Real-Time Maude verification into Ptolemy II, and have defined useful atomic propositions, so that a Ptolemy II DE model can be easily verified in Ptolemy II. This enables a model-engineering process that combines the convenience of Ptolemy II modeling and simulation with formal verification in Real-Time Maude. We have illustrated such formal verification by LTL model checking on two case studies, and have verified properties that cannot be verified by Ptolemy II simulations.

The techniques used to define the Real-Time Maude semantics for Ptolemy II DE models should be useful for defining the semantics of other hierarchical synchronous languages. For example, motivated by the complexity-reducing PALS (physically asynchronous, logically synchronous) architecture pattern [48, 49], which allows us to verify a synchronous real-time system design while ensuring that the properties also hold for the system's distributed asynchronous implementation, there is currently work in extending the avionics modeling standard AADL [50] to synchronous behavioral AADL models. Since AADL models are hierarchical, techniques in this paper could carry over to the definition of a Real-Time Maude semantics of such a synchronous version of AADL, endowing such AADL models with verification capabilities.

This work should continue in different directions. We should cover larger subsets of Ptolemy II and verify larger and more sophisticated applications. We should also add other relevant analysis methods, such as, e.g., statistical model checking to analyze probabilistic Ptolemy II models. Finally, counterexamples from Real-Time Maude verification should be visualized in Ptolemy II; this should be fairly easy to achieve since our semantics preserves the hierarchical structure of Ptolemy II models.

Acknowledgments

This work was done as part of the Lockheed Martin Advanced Technology Laboratories' NAOMI project [51] on multi-modeling design methodologies. We thank the members of the NAOMI project for encouraging this research; Christopher Brooks, Chihhong Patrick Cheng, and Man-Kit Leung for discussions on Ptolemy II; and José Meseguer for encouraging us to study the formal semantics of Ptolemy II in Real-Time Maude. We gratefully acknowledge financial support by Lockheed Martin Corporation, NSF Grant CNS 08-34709, and The Research Council of Norway.

This work was also supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #CCR-0225610 (ITR), #0720882 (CSR-EHS: PRET) and #0931843 (ActionWebs)), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312 and AF-TRUST #FA9550-06-1-0244), the Air Force Research Lab (AFRL), the Multiscale Systems Center (MuSyC) and the following companies: Bosch, National Instruments, Thales, and Toyota.

References

[1] J. Sztipanovits, G. Karsai, Model-integrated computing, IEEE Computer 30 (1997) 110–112.

[2] J. Sztipanovits, G. Karsai, Embedded software: Challenges and opportunities, in: EMSOFT'01, Springer, LNCS 2211, 2001.

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

[4] P. C. Ölveczky, J. Meseguer, Specification of real-time and hybrid systems in rewriting logic, Theoretical Computer Science 285 (2002) 359–405.

[5] P. C. Ölveczky, J. Meseguer, Semantics and pragmatics of Real-Time Maude, Higher-Order and Symbolic Computation 20 (1-2) (2007) 161–196.

[6] P. C. Ölveczky, J. Meseguer, Abstraction and completeness for Real-Time Maude, Electronic Notes in Theoretical Computer Science 176 (4) (2007) 5–27.

[7] H. Ding, C. Zheng, G. Agha, L. Sha, Automated verification of the dependability of object-oriented real-time systems, in: Proc. WORDS'03, IEEE Computer Society Press, 2003.

[8] M. AlTurki, J. Meseguer, Real-time rewriting semantics of Orc, in: M. Leuschel, A. Podelski (Eds.), Proc. PPDP'07, ACM, 2007, pp. 131–142.

[9] M. AlTurki, D. Dhurjati, D. Yu, A. Chander, H. Inamura, Formal specification and analysis of timing properties in software systems, in: FASE'09, Vol. 5503 of Lecture Notes in Computer Science, Springer, 2009, pp. 262–277.

[10] P. C. Ölveczky, A. Boronat, J. Meseguer, Formal semantics and analysis of behavioral AADL models in Real-Time Maude, in: Proc. FMOODS/FORTE'10, Lecture Notes in Computer Science, Springer, 2010, to appear.

[11] J. E. Rivera, F. Durán, A. Vallecillo, On the behavioral semantics of real-time domain specific visual languages, in: Proc. WRLA'10, Lecture Notes in Computer Science, Springer, 2010, to appear.

[12] A. Boronat, P. C. Ölveczky, Formal real-time model transformations in MOMENT2, in: FASE'10, Vol. 6013 of Lecture Notes in Computer Science, Springer, 2010, pp. 29–43.

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

[14] T. Henzinger, J. Sifakis, The discipline of embedded systems design, IEEE Computer 40 (10) (2007) 32–40.

[15] T. A. Henzinger, Two challenges in embedded systems design: predictability and robustness, Philosophical Transactions of the Royal Society A 366 (1881) (2008) 3727–3736.

[16] G. S. Fishman, Discrete-Event Simulation: Modeling, Programming, and Analysis, Springer-Verlag, 2001.

[17] Y. Zhao, E. A. Lee, J. Liu, A programming model for time-synchronized distributed real-time systems, in: RTAS'07, IEEE, 2007.

[18] E. A. Lee, H. Zheng, Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems, in: EMSOFT, ACM, 2007.

[19] K. Bae, P. C. Ölveczky, T. H. Feng, S. Tripakis, Verifying Ptolemy II discrete-event models using Real-Time Maude, in: ICFEM'09, Vol. 5885 of Lecture Notes in Computer Science, Springer, 2009, pp. 717–736.

[20] K. Bae, P. C. Ölveczky, Extending the Real-Time Maude semantics of Ptolemy to hierarchical DE models, in: Proc. 1st International Workshop on Rewriting Techniques for Real-Time Systems (RTRTS'10), 2010, to appear in Electronic Proceedings in Theoretical Computer Science.

[21] K. Bae, P. Ölveczky, T. H. Feng, S. Tripakis, Verifying Ptolemy II discrete-event models using Real-Time Maude, manuscript, http://www.ifi.uio.no/RealTimeMaude/Ptolemy (2009).

[22] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, All About Maude - A High-Performance Logical Framework, Vol. 4350 of Lecture Notes in Computer Science, Springer, 2007.

[23] R. Bruni, J. Meseguer, Semantic foundations for generalized rewrite theories, Theoretical Computer Science 360 (1-3) (2006) 386–414.

[24] J. Meseguer, Conditional rewriting logic as a unified model of concurrency, Theoretical Computer Science 96 (1992) 73–155.

[25] J. Meseguer, Membership algebra as a logical framework for equational specification, in: F. Parisi-Presicce (Ed.), Proc. WADT'97, Vol. 1376 of Lecture Notes in Computer Science, Springer, 1998, pp. 18–61.

[26] P. Viry, Equational rules for rewriting logic, Theoretical Computer Science 285 (2002) 487–517.

[27] E. Lee, A. Sangiovanni-Vincentelli, A unified framework for comparing models of computation, IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems 17 (12) (1998) 1217–1229.

[28] S. Edwards, E. Lee, The semantics and execution of a synchronous block-diagram language, Science of Computer Programming 48 (July 2003) 21–42(22).

[29] J. Misra, Distributed discrete-event simulation, ACM Comput. Surv. 18 (1) (1986) 39–65.

[30] C. Brooks, C. Cheng, T. H. Feng, E. A. Lee, R. von Hanxleden, Model engineering using multimodeling, in: 1st International Workshop on Model Co-Evolution and Consistency Management (MCCM '08), 2008.

[31] E. Lee, H. Zheng, Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems, in: EMSOFT '07: Proc. 7th ACM & IEEE Intl. Conf. on Embedded software, ACM, 2007, pp. 114–123.

[32] E. A. Lee, Modeling concurrent real-time processes using discrete events, Ann. Softw. Eng. 7 (1-4) (1999) 25–45.

[33] X. Liu, E. Matsikoudis, E. A. Lee, Modeling timed concurrent systems, in: C. Baier, H. Hermanns (Eds.), CONCUR'06, 2006, pp. 1–15.

[34] A. Cataldo, E. Lee, X. Liu, E. Matsikoudis, H. Zheng, 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), 2006.

[35] X. Liu, E. A. Lee, Cpo semantics of timed interactive actor networks, Theoretical Computer Science 409 (1) (2008) 110–125.

[36] A. Benveniste, P. Caspi, R. Lublinerman, S. Tripakis, Actors without Directors: a Kahnian View of Heterogeneous Systems, in: HSCC'09, Vol. 5469 of Lecture Notes in Computer Science, Springer, 2009, pp. 46–60.

[37] C. P. Cheng, T. Fristoe, E. A. Lee, Applied verification: The Ptolemy approach, Technical Report UCB/EECS-2008-41, EECS Department, University of California, Berkeley (April 2008).

[38] S. Tripakis, C. Sofronis, P. Caspi, A. Curic, Translating Discrete-Time Simulink to Lustre, ACM Transactions on Embedded Computing Systems 4 (4) (2005) 779–818.

[39] P. Caspi, D. Pilaud, N. Halbwachs, J. Plaice, Lustre: a declarative language for programming synchronous systems, in: 14th ACM Symp. POPL, ACM, 1987.

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

[41] D. Harel, Statecharts: A visual formalism for complex systems, Sci. Comput. Programming 8 (1987) 231–274.

[42] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, S. Yovine, The algorithmic analysis of hybrid systems, Theoretical Computer Science 138 (1995) 3–34.

[43] A. Agrawal, G. Simon, G. Karsai, Semantic translation of simulink/stateflow models to hybrid automata using graph transformations, Electron. Notes Theor. Comput. Sci. 109 (2004) 43–56.

[44] G. Zhou, M.-K. Leung, E. A. Lee, A code generation framework for actor-oriented models with partial evaluation, in: ICESS, Vol. 4523 of Lecture Notes in Computer Science, Springer, 2007.

[45] G. Hamon, A denotational semantics for Stateflow, in: EMSOFT '05, ACM, 2005, pp. 164–172.

[46] A. Farzan, F. Chen, J. Meseguer, G. Rosu, Formal analysis of Java programs in JavaFAN, in: R. Alur, D. Peled (Eds.), CAV, Vol. 3114 of Lecture Notes in Computer Science, Springer, 2004, pp. 501–505.

[47] J. Meseguer, G. Rosu, The rewriting logic semantics project, Theoretical Computer Science 373 (3) (2007) 213–237.

[48] A. Al-Nayeem, M. Sun, X. Qiu, L. Sha, S. P. Miller, D. D. Cofer, A formal architecture pattern for real-time distributed systems, in: Proc. 30th IEEE Real-Time Systems Symposium, IEEE, 2009.

[49] J. Meseguer, P. C. Ölveczky, Formalization and correctness of the PALS pattern for asynchronous real-time systems, Tech. rep., Department of Computer Science, University of Illinois at Urbana-Champaign, http://hdl.handle.net/2142/14214 (2009).

[50] SAE AADL Team, AADL homepage, http://www.aadl.info/ (2009).

[51] T. Denton, E. Jones, S. Srinivasan, K. Owens, R. W. Buskens, NAOMI – an experimental platform for multi-modeling, in: MoDELS'08, Vol. 5301 of LNCS, Springer, 2008.