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

Kyungmin Bae[1], Peter Csaba Ölveczky[2], Thomas Huining Feng[3], and Stavros Tripakis[3]

[1] University of Illinois at Urbana-Champaign
[2] University of Oslo
[3] University of California, Berkeley

Abstract This paper shows how Ptolemy II discrete-event (DE) models can be formally analyzed using Real-Time Maude. We formalize in Real-Time Maude the semantics of a subset of hierarchical Ptolemy II DE models, and explain how the code generation infrastructure of Ptolemy II has been used to automatically synthesize a Real-Time Maude verification model from a Ptolemy II design model. This enables a model-engineering process that combines the convenience of Ptolemy II DE modeling and simulation with formal verification in Real-Time Maude.

1 Introduction

Model-based design principles put the construction of models at the center of embedded system design processes [16,9]. Useful models are executable, providing simulations of system functionality, performance, power consumption, or other properties. Ideally, models are translated (code generated) automatically to produce deployable embedded 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, and LabVIEW Embedded, from National Instruments. Models can also be used to guide formal verification, which can provide proofs of safety properties or identification of security vulnerabilities.

Ptolemy II is a well-established modeling and simulation tool, developed at UC Berkeley, that provides a powerful and intuitive graphical modeling language to allow a user to build hierarchical models that combine different models of computations [6]. In this paper, we focus on discrete-event (DE) models, which are explicit about timing behavior of systems. Discrete-event modeling is a time honored and widely used approach for system simulation [8]. More recently, it has been proposed as basis for synthesis of embedded real-time software [17]. The Ptolemy II realization of DE has a rigorous formal semantics rooted in the fixed-point semantics of synchronous languages [11].

This paper describes our work on enriching a significant subset of hierarchical Ptolemy II DE models with *formal verification* capabilities using Real-Time

Maude [13] as back-end. Real-Time Maude is a high-performance tool that extends the rewriting-logic-based Maude [3] system to support the formal specification and analysis of object-based real-time systems. Real-Time Maude provides a spectrum of formal analysis methods, including rewriting for simulation purposes, reachability analysis, and linear temporal logic (LTL) model checking.

In particular, we explain how we have enriched Ptolemy II DE models with formal verification capabilities by:

1. Formalizing the semantics of transparent hierarchical Ptolemy II DE models in Real-Time Maude.
2. Defining useful atomic state propositions, so that the Ptolemy user can specify temporal logic properties to be verified without understanding how Ptolemy models are represented in Real-Time Maude.
3. Using Ptolemy II's code generation infrastructure [18] to automatically synthesize a Real-Time Maude verification model from a Ptolemy design model, and by explaining how both code generation and verification have been integrated into Ptolemy, so that a Ptolemy model can be verified within Ptolemy.

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.

The main contributions of our work are:

– Enriching Ptolemy with formal verification capabilities to verify properties, such as the liveness property in Section 6, that cannot be checked using Ptolemy simulations. Furthermore, the synthesized verification model can be formally analyzed w.r.t. other properties (e.g., determinism, etc.).
– We show how Real-Time Maude can define the semantics of synchronous languages with fixed-point semantics. These techniques should be useful for defining the formal semantics of other synchronous languages. Our semantics also provides a basis for extensions to, e.g., *probabilistic* Ptolemy models, that can then be subjected to *statistical model checking* using tools like VeStA [15].

Our work is conducted in the context of the NAOMI project [4], where Lockheed Martin Advanced Technology Laboratories (LM ATL), UC Berkeley, UIUC, and Vanderbilt University work together to develop a multi-modeling design methodology. A key part of this project is the systematic use of model transformations and code generation to maintain consistency across models.

Section 2 briefly introduces Ptolemy II and Real-Time Maude. The Real-Time Maude semantics of Ptolemy II DE models is described in Section 3. Section 4 presents the atomic propositions that allow the user to specify his/her LTL properties without having the understand the Real-Time Maude translation of a Ptolemy model. Section 5 explains the Real-Time Maude code generation and integration into Ptolemy. Section 6 illustrates the use of our techniques to verify a Ptolemy model. Finally, Section 7 presents some related work and Section 8 gives some concluding remarks. More details about the Real-Time Maude semantics of Ptolemy DE, as well as three additional verification case studies, can be found in the longer technical report [1].

2 Preliminaries on Real-Time Maude and Ptolemy

2.1 Rewriting Logic and Real-Time Maude

Modeling. A Real-Time Maude *timed module* specifies a *real-time rewrite theory* of the form (Σ, E, IR, TR), where:

- (Σ, E) is a *membership equational logic* [3] theory with Σ a signature[4] and E a set of *confluent and terminating conditional equations.* (Σ, E) specifies the system's state space as an algebraic data type, and must contain a specification of a sort `Time` modeling the (discrete or dense) time domain.
- IR is a set of (possibly conditional) *labeled instantaneous rewrite rules* specifying the system's *instantaneous* (i.e., zero-time) local transitions, written `rl [`l`] : `t` => `t', where l is a *label.* Such a rule specifies a *one-step transition* from an instance of t to the corresponding instance of t'. The rules are applied *modulo* the equations E.[5]
- TR is a set of *tick (rewrite) rules*, written with syntax

  `rl [`l`] : {`t`} => {`t'`} in time `τ` .`

  that model time elapse. `{_}` is a built-in constructor of sort `GlobalSystem`, and τ is a term of sort `Time` that denotes the *duration* of the rewrite.

The initial state must be a ground term of sort `GlobalSystem` and must be reducible to a term of the form `{`t`}` using the equations in the specifications.

The Real-Time Maude syntax is fairly intuitive. For example, function symbols, or *operators*, are declared with the syntax `op` f `:` $s_1 \ldots s_n$ `->` s. f is the name of the operator; $s_1 \ldots s_n$ are the sorts of the arguments of f; and s is its (value) *sort.* Equations are written with syntax `eq` t `=` t', and `ceq` t `=` t' `if` *cond* for conditional equations. The mathematical variables in such statements are declared with the keywords `var` and `vars`. We refer to [3] for more details on the syntax of Real-Time Maude.

We make extensive 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.[6]

In object-oriented Real-Time Maude modules, a *class* declaration

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

declares a 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 `<` $O : C \mid att_1 : val_1, ..., att_n : val_n$ `>` of sort `Object`, where O, of sort `Oid`, is the object's *identifier*, and where val_1 to val_n are the current values of the attributes att_1 to att_n. In a concurrent

[4] i.e., Σ is a set of declarations of *sorts*, *subsorts*, and *function symbols*

[5] E is a union $E' \cup A$, where A is a set of equational axioms such as associativity, commutativity, and identity, so that deduction is performed *modulo* A. Operationally, a term is reduced to its E'-normal form modulo A before any rewrite rule is applied.

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

object-oriented system, the state is a term of the 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, so that rewriting is *multiset rewriting* supported directly in Real-Time Maude.

The dynamic behavior of concurrent object systems is axiomatized by specifying each of its transition patterns by a rewrite rule. For example, the rule

```
rl [l] :  < O : C | a1 : 0, a2 : y, a3 : w, a4 : z >  =>
          < O : C | a1 : T, a2 : y, a3 : y + w, a4 : z >
```

defines a parameterized family of transitions (one for each substitution instance) which can be applied whenever the attribute `a1` of an object `O` of class `C` has the value `0`, with the effect of altering the attributes `a1` and `a3` of the object. "Irrelevant" attributes (such as `a4`, and the *right-hand side occurrence* of `a2`) need not be mentioned in a rule (or equation).

A *subclass* inherits all the attributes and rules of its superclasses.

Formal Analysis. A Real-Time Maude specification is *executable*, and the tool offers a variety of formal analysis methods. The *rewrite* command simulates *one* behavior of the system *up to a certain duration*. It is written with syntax (`trew` t `in time <=` τ `.`), where t is the initial state and τ is a term of sort `Time`. The *search* command uses a breadth-first strategy to analyze all possible behaviors of the system, by checking whether a state matching a *pattern* and satisfying a *condition* can be reached from the initial state.

Real-Time Maude also extends Maude's *linear temporal logic model checker* to check whether each behavior, possibly up to a certain time bound, satisfies a temporal logic formula. *State propositions* are terms 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} where {t} `|=` *prop* evaluates to `true`. A temporal logic *formula* is constructed by state propositions and temporal logic operators such as `True`, `False`, `~` (negation), `/\`, `\/`, `->` (implication), `[]` ("always"), `<>` ("eventually"), and `U` ("until"). The time-bounded model checking command has syntax

(`mc` t `|=t` *formula* `in time <=` τ `.)`

for initial state t and temporal logic formula *formula* .

2.2 Ptolemy II and its DE Model of Computation

The Ptolemy project[7] studies modeling, simulation, and design of concurrent, real-time, embedded systems. The key underlying principle in the project is the

[7] `http://ptolemy.eecs.berkeley.edu/`

use of well-defined *models of computation* (MoCs) that govern the interaction between concurrent components. A major problem area being addressed is the use of *heterogeneous* mixtures of MoCs [6]. A result of the project is a software system called Ptolemy II, implemented in Java. Ptolemy II allows a user to build hierarchical models that combine different MoCs, including state machines, data flow, and discrete-event models. Models can be visually designed and simulated. In addition, Ptolemy II's *code generation* capabilities allow models to be translated into models in other languages or into imperative code, e.g., in C and Java.

Discrete-Event Models in Ptolemy II The focus of this paper is the formalization of a *subset* of Ptolemy II *discrete-event* (DE) models in Real-Time Maude. A Ptolemy II model is a hierarchical composition of *actors* with *connections* between the actors' *input ports* and *output ports*. The actors represent data manipulation units, whose execution is governed by a special attribute belonging to the model called *director*. Such a model can itself be treated as an actor, that we call a *composite actor*. (Non-composite actors are called *atomic actors*.) Ptolemy II also supports *modal models*, which are models with finite state machine controllers. See Section 2.3 for an example of a Ptolemy II model.

DE actors consume and produce *events* at their input and output ports, according to the *tagged signal model* [10]. A tagged event is a pair (v, t) where v is a *value* in a complete partial order (CPO) and t is a *tag*, modeling the time at which the event occurs. Ptolemy II DE models use *super-dense* time, in which a tag t is a pair $(\tau, n) \in \mathbb{R}_{\geq 0} \times \mathbb{N}$, where τ is the *timestamp* that indicates the model time when this event occurs, and n is the *microstep index*. Super-dense time is useful for modeling multiple events that happen at the same time (i.e., have the same timestamp), but in sequence, where perhaps some events cause other events. Super-dense tags are totally ordered using a *lexicographic* order: $(\tau_1, n_1) \leq (\tau_2, n_2)$ iff $\tau_1 < \tau_2$, or $\tau_1 = \tau_2$ and $n_1 \leq n_2$.

The semantics of Ptolemy II DE models [10] combines a synchronous-reactive fixed-point iteration with advancement of time governed by an event queue [11]. Events in that queue are ordered by their tags. Operation proceeds by iterations, each time removing one or more events with the smallest tag from the queue. That tag is considered the current *model time*. The removed events are fed to their designated actors. After that, actors with events available are executed, which may generate new events into the queue. A difference between Ptolemy II and standard DE simulators is that, at any model time (τ, n), the semantics is defined as the *least fixed-point* of a set of equations, similarly to a synchronous model [5]. This allows Ptolemy II models to have arbitrary *feedback loops*. Semantics of such models can always be given although they may result in *unknown* (*bottom*) values, in case the model contains *causality cycles*. Conceptually, the semantics can be captured as shown in Figure 1.

Code Generation Infrastructure. Ptolemy II offers a code generation framework using an *adapter-based mechanism*. A *codegen adapter* is a component that generates code for a Ptolemy II actor. An adapter essentially consists of a Java

```
Q := empty; // Initialize the event queue to be empty.
for each actor A do
  A.inititialize(); // Initialize A; may generate new events in Q

while Q is not empty do
  E := set of all events in Q with smallest tag;
  remove elements of E from Q;
  initialize ports with values in E or "unknown" (bottom of CPO);
  while port values changed do
    for each actor A receiving new values do
      if A.prefire() then  // Determine whether A needs to be fired
        A.fire(); // May increase port values according to CPO
  end while;      // Fixed-point reached for the current tag
  for each actor A that has been fired do
    A.postfire(); // Updates actor state; may generate new events in Q
end while;
```

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

class file and a *code template* file that together specify the actor's behavior. The code template file 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 advantage of this scheme is that it decouples the writing of Java code and target code (otherwise the target code would be wrapped in strings and interspersed with Java code).

2.3 Example: A Simple Traffic Light System

Figure 2 shows a Ptolemy DE model of a simple non-fault-tolerant traffic light system consisting of one car light and one pedestrian light at a pedestrian crossing. Each traffic 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 considered to be *on* iff the corresponding variable has the value 1. The lights are controlled by the two *finite state machine* (FSM) actors `CarLightNormal` and `PedestrianLightNormal` that send values to set the variables; in addition, `CarLightNormal` sends signals to the `PedestrianLight-Normal` actor through its `Pgo` and `Pstop` output ports. These signals are received by the `PedestrianLightNormal` actor after a *delay* of one time unit.

Figure 3 shows the FSM actor `PedestrianLightNormal`. 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` input port, then it goes to state `Pgreen`, outputs the value 0 through its `Pred` output port, and outputs the value 1 through its `Pgrn` port.

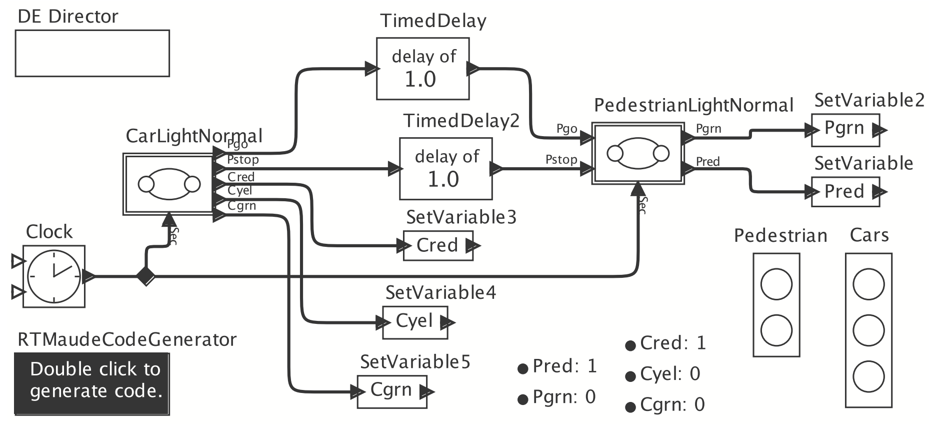

Figure 2. Simple Traffic Light model in Ptolemy II

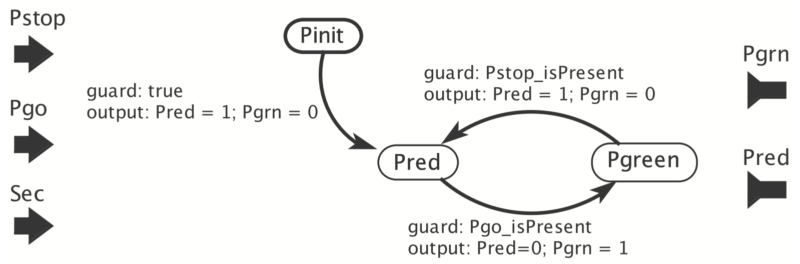

Figure 3. The `PedestrianLightNormal` FSM actor.

Figure 4 shows the FSM actor `CarLightNormal`. 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.

3 Real-Time Maude Semantics of Ptolemy DE Models

This section gives a brief overview of the Real-Time Maude formalization of the Ptolemy DE semantics that is the basis for our work. Due to the lack of space, and in order to convey our ideas without introducing too much detail, we present 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 expressions are *constants*.

The report [1] explains the Real-Time Maude semantics that we actually use, and that covers the subset of Ptolemy listed in Section 3.1, including hierarchical

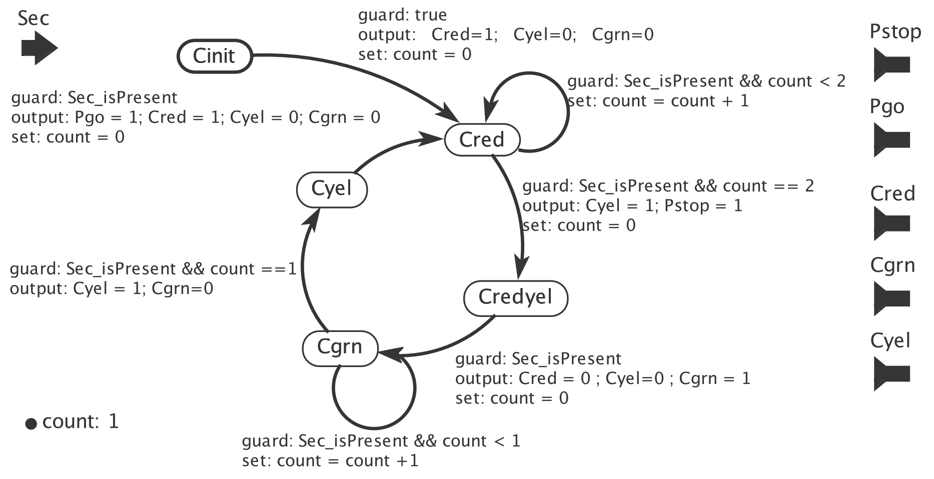
Sec
Cinit
guard: true
output: Cred=1; Cyel=0; Cgrn=0
set: count = 0
guard: Sec_isPresent && count < 2
set: count = count + 1
Pstop
Pgo
guard: Sec_isPresent
output: Pgo = 1; Cred = 1; Cyel = 0; Cgrn = 0
set: count = 0
Cred
Cyel
guard: Sec_isPresent && count == 2
output: Cyel = 1; Pstop = 1
set: count = 0
Cred
guard: Sec_isPresent && count ==1
output: Cyel = 1; Cgrn=0
Cgrn
Credyel
Cgrn
guard: Sec_isPresent
output: Cred = 0 ; Cyel=0 ; Cgrn = 1
set: count = 0
Cyel
count: 1
guard: Sec_isPresent && count < 1
set: count = count +1

Figure 4. The `CarLightNormal` FSM actor.

actors and expressions with variables. The entire executable Real-Time Maude semantics is available at `http://www.ifi.uio.no/RealTimeMaude/Ptolemy`.

3.1 Supported Subset of Ptolemy

We currently support Real-Time Maude analysis of *transparent discrete event* (DE) Ptolemy models; that is, DE models where subdiagrams are also executed under the DE director. We support composite actors, modal models, and the following atomic actors: finite state machine (FSM), timed delay, variable delay, clock, current time, timer, noninterruptible timer, pulse, ramp, timed plotter, set variable, and single event actors. We also support connections with multiple destinations, split signals, and both single ports and multi-input ports.

3.2 Representing Flat Ptolemy DE Models in Real-Time Maude

This section explains how a Ptolemy model is represented as a Real-Time Maude term in (the slightly simplified version of) our semantics. We only show the representation for a small subset of the actors listed above, and refer to [1] 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.

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

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

The `ports` attribute denotes the set of *ports* of the actor. In our model, a port is modeled as an object, as shown below. The `parameters` attribute represents the *parameters* of the corresponding Ptolemy actor, together with their values, as a semicolon-separated set of terms of the form `'`*parameter-name* `|->` *value*.

Current Time. Ptolemy's *current time* actor produces an output token on each firing with a value that is the current model time. In Real-Time Maude, such actors are represented as object instances of the following class that extends the class `Actor` with an attribute `current-time` denoting the current "model time":

```
class CurrentTime | current-time : Time .    subclass CurrentTime < Actor .
```

Timed Delay. A *timed delay* actor propagates an incoming event after a given time delay. If the *delay* parameter is 0.0, then there is a "microstep" delay on the generation of the output event. Since the *delay* parameter is represented in the `parameters` attribute of `Actor`, this subclass does not add any attributes:

```
class Delay .        subclass Delay < Actor .
```

Set Variable. This actor contains a variable that is set by a signal:

```
class SetVariable | variableName : VarId .   subclass SetVariable < Actor .
```

Finite State Machine (FSM) Actors. An FSM actor is a transition system containing finite sets of states (or "locations"), local variables, and transitions. A transition has a guard expression, and can contain a set of output actions and variable assignments. 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.

An `FSM-Actor` is then characterized by its *current state*, its transitions, and the current values of its local variables:

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

A location is the sort of the local "states" of the transition system. In particular, quoted identifiers (`Qids`) are state names. We model the transitions as a semicolon-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_{j_l} `|->`$e_{j'_l}$`}`
for state/locations s_1 and s_2, port names p_i, variables v_i, and expressions e_i.

Ports. A *port* is represented as an object, with a name (the identifier of the port object), a status (`unknown`, `present`, or `absent`), and a `value`. We also have subclasses for input and output 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] .
```

Connections. A connection is represented as a term p_o `==>` $p_{i_1};\ldots;p_{i_n}$ of sort `Connection`, where the p_js have the form a`!`p for a a name of an actor and p a name of a port. 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, the sort `Connection` is defined to be a subsort of the sort `Object`.

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, 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.

Example: Representing the Flat Traffic Light Model. Consider the flat non-fault-tolerant traffic light system given in Section 2.3. The Real-Time Maude representation of the `TimedDelay2` delay actor is then

```
< 'TimedDelay2 : Delay |
       parameters : 'delay |-> # 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]

```
< 'CarLightNormal : FSM-Actor |
     initState : 'Cinit,  currState : 'Cinit,  variables : 'count |-> # 1,
     ports : < 'Sec : InPort | value : # 0, status : absent >
             < 'Pgo : OutPort | value : # 0, status : absent >
             ...,
```

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

```
    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 SetVariable objects, and the global event queue object.

3.3 Specifying the Behavior of Flat DE Models

As explained in Section 2.2, the behavior of Ptolemy DE models can be summarized as repeatedly performing the following actions:

- Advance time until the time to fire the first events in the queue is $(0, 0)$.
- Then an iteration of the system is performed. That is:
  1. (Prefire) 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, states are updated for actors with inputs or scheduled events, and new events are generated and inserted into the event queue.

The following tick rule advances time until the time when the first events in the event queue are scheduled (we first declare all the variables used):

```
vars SYSTEM OBJECTS PORTS PORTS' REST : ObjectConfiguration . var N : Nat .
vars O O' : Oid . var EVTS : Events . var NZT : NzTime . var NZ : NzNat .
vars P P' : PortId . var QUEUE : EventQueue . vars T T' : Time .
var PS : PortStatus . var EPIS : EportIdSet . vars V TV : Value .
var VAL : ValueMap . vars STATE STATE' : Location .
var BODY : TransBody . var TG : TransGuard . var TRANSSET : TransitionSet .

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 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` (where 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 single objects, as shown later.

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 >
    postfireAll(portFixPoints(addEventsToPorts(EVTS, prefire(SYSTEM))))} .
```

The function `prefire` initializes each actor before firing. In particular, it sets the `status` of each port 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 ports (fire), 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. These functions have sort `Configuration`, whereas the equations defining them involve variables of the subsort `ObjectConfiguration`, ensuring that each function has finished computing before the "next" function is computed:

```
ops prefire portFixPoints postfire : Configuration -> Configuration .
```

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

Initialize Actors. In our simplified setting, the `prefire` function just clears all the ports, that is, sets their `status` to `unknown`:

```
eq prefire(< O : Actor | ports : PORTS > REST)
     = < O : Actor | ports : clearPorts(PORTS) >   prefire(REST) .
eq prefire(SYSTEM) = SYSTEM [owise] .

op clearPorts : Configuration -> Configuration .
eq clearPorts(< P : Port | status : PS > PORTS)
     = < P : Port | status : unknown > clearPorts(PORTS) .
eq clearPorts(none) = none .
```

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(`*actors 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 is `absent`). The fixed-point is reached when no such equation can be applied. The `portFixPoints` operator is then removed by using the `owise` construct of Real-Time Maude:

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

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

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) .
```

Other actors generate immediate output when receiving input. For example, when a *current time* actor fires, it outputs the current model time:

```
ceq portFixPoints(
     < O : CurrentTime | current-time : 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 .
```

The definition of `portFixPoints` for FSM actors relies on the assumption that at most one transition is enabled at any time. In the following conditional equation, one transition from the current state `STATE` is enabled. 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, variables : VAL,
                        transitions : (STATE --> STATE' {BODY}) ; TRANSSET >
                   REST)
  = portFixPoints(< O : FSM-Actor |
                        ports : updateOutPorts(VAL, BODY,
                                 < P : OutPort | > < P' : InPort | > PORTS) >
                   REST)
 if transApplicable(< P : OutPort | > < P' : InPort | > PORTS, VAL, BODY) .
```

Another equation sets all output ports to `absent` if there is enough information to determine that no transition can become enabled in the current round.

Postfire. The `postfire` function updates internal states and generates new events that are inserted into the event queue. The `postfire` function distributes over the actor objects in the configuration. An *owise* equation defines `postfire` to be the identity function (that neither generates a future event nor changes its local state) on those actors that do not have other equations defining `postfire`.

If a time *delay* actor has input in its `'input` port, then it generates an event with a 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`. This event is added to the global event queue using the `addEvent` function that adds the new event into the correct place in the event queue:

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

An FSM actor does not generate future events, but `postfire` updates the location and variables of the actor if it has input and has an enabled transition:

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

Defining Timed Behavior. Finally, we must define the function `delta`, that specifies the effect of time elapse, on single actors. Time elapse affects the internal state of `CurrentTime` actors by increasing the value of the `current-time` attribute according to the elapsed time. Time elapse does not affect the other actors we discuss here:

```
eq delta(< O : CurrentTime | current-time : T >, T') =
         < O : CurrentTime | current-time : T + T' > .

eq delta(O:Object, T) = O:Object [owise] .
```

Defining Initial Events. 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.

4 Formal Analysis of Ptolemy II DE Models

As mentioned in Section 2.1, the Real-Time Maude verification model synthesized from a Ptolemy design model can be analyzed in different ways. In this paper, we focus 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 LTL operators, such as `/\` (conjunction), `~` (negation), `[]` ("always"), `<>` ("eventually"), etc. Having to define atomic 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 Maude. To free the user from this burden, we have predefined some generic atomic propositions. For example, the property

$$\mathit{actorId}\ \texttt{|}\ \mathit{var_1}\ \texttt{=}\ \mathit{value_1}\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{var_n}\ \texttt{=}\ \mathit{value_n}$$

holds in a state if the value of each "variable" var_i in the `parameters`s of an actor equals value_i for each $i \in \{1, \ldots, n\}$. Here, *actorId* is the *global actor name* of a given actor (see below). For FSM actors (and modal models), the property

$$\mathit{actorId}\ \texttt{@}\ \mathit{location}$$

holds if and only if the FSM actor with global name *actorId* is in location (or "local state") *location*. Since we may analyze also hierarchical models, the global actor name *actorId* in the above formulas must be a list m `.` o_1 `.` o_2 `.` ... `.` o_n, for $n \geq 0$, where m is the name of the (top-level) model, o_1 is the object name of a top-level actor in this model, and o_{i+1} is the name of an inner actor of the actor o_i. Section 6 shows how these propositions can be used to verify Ptolemy models.

5 Code Generation and Integration with Ptolemy

This section shows how Ptolemy's adapter code generation infrastructure has been used to automatically generate Real-Time Maude code from a Ptolemy DE model, and to integrate Real-Time Maude analysis of DE models into Ptolemy.

5.1 Implementing the Real-Time Maude Code Generator

Ptolemy gives the user the possibility of adding a "code generation button" to a (top-level) Ptolemy model. When this button is double-clicked, Ptolemy opens a dialog window which allows the user to start code generation and give commands to execute the generated code.

Ptolemy provides an *adapter* infrastructure to support the generation of code into any target language. In particular, Ptolemy provides a Java class `Code-GeneratorHelper` that contains utility methods such as `getComponent()`, which returns a (Java) object containing all information about an actor, including its name, parameters, ports, inner actors, etc. This class furthermore contains "skeleton" functions like `String generateFireCode()`, which should generate the code executed when the actor is "fired," `Set getSharedCode()`, which should generate code shared by multiple instances of the same actor class, and so on.

An adapter class may have an associated *template file* containing code blocks of the form `/***`*header*`(`*parameters*`)***/` *code pattern* `/**/`, where the *code pattern* is code written in the target language, but that can be parametrized with variables, and also have macro functions. Macros are prefixed with '`$`'.

The Real-Time Maude code generation is implemented by redefining the functions `getSharedCode()` and `generateFireCode()` in the adapter class for each type of actor. For each adapter class A, its associated template file includes a code block with header `semantics_`A that is just the Real-Time Maude module defining the formal semantics of the actor A! The template file also includes a code block with header `attr_`A that defines the attributes of the actor and their initial values. In Ptolemy, each actor class is a subclass of the class `Entity`. Therefore, we defined an adapter class for `Entity` that is a superclass of every actor adapter class. The template file for `Entity` hence contains

```
/***semantics_Entity***/
(tomod ACTOR is
...
  class Actor | ports : Configuration,  parameters : ValueMap .
...
endtom)
/**/

/***fireBlock($attr_terms)***/
< '$info(name) : $info(class) | $attr_terms >
/**/

/***attr_Entity***/
ports : ($info(ports)),
parameters : ($info(parameters))
/**/
```

The parameter `attr_terms` will be replaced by set of `attr_`*`Actor`* code blocks for each *`Actor`* a super class of the given actor. `$info` is a macro that uses Ptolemy's `getComponent()` to extract information, such as the name, the class, etc., about the actor instance. Likewise, the template file for `CurrentTime` contains

```
/***semantics_CurrentTime***/
(tomod CURRENT-TIME is inc ACTOR .
  ...
  class CurrentTime | current-time : Time .   subclass CurrentTime < Actor .
  ...
  eq portFixPoints(...) = ... .
endtom)
/**/

/***attr_CurrentTime***/
```

```
current-time : 0
/**/
```

The function `getSharedCode()` is used to generate the Real-Time Maude modules defining the semantics of those actors that appear in the Ptolemy model, and is defined as a Java function that returns the set of all code blocks (from the related template files) whose header starts with "`semantics`." Hence, for a `CurrentTime` actor, `getSharedCode()` returns the above two Real-Time Maude modules (and adds modules for LTL model checking in the same way).

The function `generateFireCode()` is used to generate the Real-Time Maude term representing the (initial state of the) given Ptolemy model. It generates the code from the code templates with header `fireBlock` and `$attr` in the appropriate adapter classes; that is, a Real-Time Maude object corresponding to the initial state of the actor.

5.2 Verifying Ptolemy DE Models from within Ptolemy

We have integrated *both* code generation and *verification* into Ptolemy, so that a Ptolemy DE model can be verified by pushing the *RTMaudeCodeGenerator* button in the Ptolemy model. The dialog window that then pops up allows the user to write his/her simulation and model checking commands. After clicking the `Generate` button of the dialog window, the generated Real-Time Maude code and the result of executing the analysis commands are shown in the dialog box.

6 Examples and Case Studies

This section shows how the LTL model checking infrastructure in Section 4 and the Real-Time Maude code generation can be used to verify Ptolemy DE models.

Verifying the Traffic Light Model. In the Ptolemy model in Section 2.3, 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 /\
        'DE_SimpleTrafficLight . '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 the pedestrian light is green when the car light is *not* green, and it also infinitely often the case the car light is green when the pedestrian light is not green:

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

As explained in Section 5.2, these formulas can be entered into the dialog box that pops up when the `RTMaudeCodeGenerator` button is clicked in the Ptolemy model; the dialog box will then display the results of the verification.

Other Examples and Case Studies. We have also verified the following three larger Ptolemy DE models in the same way (see [1] for details):

1. A *hierarchical* fault-tolerant extension of the traffic light system,
2. the railroad crossing benchmark, and
3. an assembly line system originally due to Misra.

7 Related Work

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 [2]. However, they require *data abstraction* to map models into finitary automata, and they do not use the code generation framework. On the other hand, Maude has been used to give semantics to a wide range of programming and modeling languages (see, e.g., [7,12]). Real-Time Maude is also used to analyze AADL [14] models of avionics embedded systems, but we are not aware of any translation of a synchronous real-time language into Maude or Real-Time Maude.

8 Concluding Remarks

We have formalized the semantics of a significant subset of transparent hierarchical Ptolemy II DE models in Real-Time Maude, and have shown how such Ptolemy design models can be verified by integrating Real-Time Maude code generation and model checking into Ptolemy, enabling a model-engineering process for embedded systems that leverages the convenience of Ptolemy II DE modeling and simulation with the formal verification of Real-Time Maude.

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.

Acknowledgments. This work is part of the Lockheed Martin Advanced Technology Laboratories' NAOMI project. We thank the members of the NAOMI project for enabling and encouraging this research; Christopher Brooks, Chihhong Patrick Cheng, Edward A. Lee, and Man-Kit Leung for discussions on Ptolemy II; Edward A. Lee for suggestions to drafts of this paper; José Meseguer for encouraging us to study the formal semantics of Ptolemy in Real-Time Maude; and Lockheed Martin, through the NAOMI project, and The Research Council of Norway for financial support.

References

1. Bae, K., Ölveczky, P., Feng, T.H., Tripakis, S.: Verifying Ptolemy II discrete-event models using Real-Time Maude (2009), manuscript, `http://www.ifi.uio.no/RealTimeMaude/Ptolemy`
2. Cheng, C.P., Fristoe, T., Lee, E.A.: Applied verification: The Ptolemy approach. Technical Report UCB/EECS-2008-41, EECS Department, University of California, Berkeley (April 2008)
3. Clavel, M., Durán, F., Eker, S., Lincoln, P., Mart-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework, LNCS, vol. 4350. Springer (2007)
4. Denton, T., Jones, E., Srinivasan, S., Owens, K., Buskens, R.W.: NAOMI – an experimental platform for multi-modeling. In: MoDELS'08. LNCS, vol. 5301. Springer (2008)
5. Edwards, S., Lee, E.: The semantics and execution of a synchronous block-diagram language. Science of Computer Programming 48, 21–42(22) (July 2003)
6. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity—the Ptolemy approach. Proceedings of the IEEE 91(2), 127–144 (2003)
7. Farzan, A., Chen, F., Meseguer, J., Rosu, G.: Formal analysis of Java programs in JavaFAN. In: CAV. LNCS, vol. 3114, pp. 501–505. Springer (2004)
8. Fishman, G.S.: Discrete-Event Simulation: Modeling, Programming, and Analysis. Springer-Verlag (2001)
9. Giese, H., Karsai, G., Lee, E., Rumpe, B., Schätz, B. (eds.): Model-based Engineering of Embedded Real-time Systems. Dagstuhl Seminar Proc. 07451 (2007)
10. Lee, E.A.: Modeling concurrent real-time processes using discrete events. Annals of Software Engineering 7(1-4), 25–45 (1999)
11. Lee, E.A., Zheng, H.: Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In: EMSOFT. ACM (2007)
12. Meseguer, J., Rosu, G.: The rewriting logic semantics project. Theoretical Computer Science 373(3), 213–237 (2007)
13. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation 20(1-2), 161–196 (2007)
14. SAE: AADL (2007), `http://www.aadl.info/`
15. Sen, K., Viswanathan, M., Agha, G.A.: VeStA: A statistical model-checker and analyzer for probabilistic systems. In: QEST'05. IEEE (2005)
16. Sztipanovits, J., Karsai, G.: Model-integrated computing. IEEE Computer pp. 110–112 (1997)
17. Zhao, Y., Lee, E.A., Liu, J.: A programming model for time-synchronized distributed real-time systems. In: RTAS'07. IEEE (2007)
18. Zhou, G., Leung, M.K., Lee, E.A.: A code generation framework for actor-oriented models with partial evaluation. In: ICESS. LNCS, vol. 4523. Springer (2007)