

A Formal Representation of Discrete Event Models in Ptolemy II

Hae Young Lee, Won-Tae Kim, In-Geol Chun, Woonchul Kang, Seung-Min Park

Electronics and Telecommunications Research Institute, Daejeon 305-700, Republic of Korea

{haelee, wtkim, igchun, wchkang, minpark}@etri.re.kr

Abstract—Within the embedded software design process, formal methods including formal specifications and formal verifications can be used to check designs, with the goal of ensuring the requirements of embedded software, such as reliability and concurrency. This paper presents a means to formally describe the discrete event models of Ptolemy II that is a modeling and simulation environment for embedded software. The proposed formal specification distinguishes between model structure and behavior and can construct complex models through hierarchical compositions of sub-model. The specification may be used as a basis for formal verifications of the models.

Keywords— discrete event model, formal specification, formal method, discrete event system specification, embedded system.

I. INTRODUCTION

Embedded software must be reliable, concurrent, and reactive in real time [1]. Moreover, it monitors and controls the physical processes, usually with feedback loops where physical processes affect computations and vice versa [2]. Current software engineering methodologies are inadequate for handling such features of embedded software [3]. A large pitfall of them is that most bugs are caught in the final phases of the process, at system integration and testing time, which may result in modifying the system requirements, specification or design [4]. Thus, model-based design paradigms, with its promise for greater design automation and formal guarantees of reliability, are particularly attractive in the embedded software development [5].

Ptolemy II [6] is the software infrastructure of the Ptolemy project studying heterogeneous modeling, simulation, and design of embedded software together with the systems within which it is embedded [2]. In order to handle the heterogeneity of and complexities of embedded systems, Ptolemy II provides the hierarchically heterogeneous approach in which models at the same level are locally homogeneous, while allowing different formal models of computation at different levels in a hierarchy [1]. The formal models of computation that Ptolemy II supports include the continuous time, discrete event, process network, dynamic dataflow, finite state machine, rendezvous, synchronous dataflow, synchronous reactive, wireless models, and some

experimental models. Although these models are mathematically well-grounded, they are realized as complex Java classes. Therefore, it might be difficult to directly apply formal verification or theorem proving techniques to the current form of the models.

In this paper, we propose a means to represent atomic and composite models in a more abstract and formal form, within the *discrete event (DE) models* [7] of Ptolemy II. The proposed specification consists of the *atomic model specification* that represents the *behavior* of an atomic model and the *composite model specification* that primarily describes the *structure* of a composite model consisting of multiple atomic models and couplings between them. The proposed specification can also represent the *hierarchical abstraction* [2] of Ptolemy II in which a composite model can be used as an atomic model at a higher-level. The specification may be used as a basis for verifying and proving properties of DE models, as shown in Fig. 1. The formal specifications for models may be converted from Ptolemy II DE models (i.e. Java source codes), or be used to generate source codes.

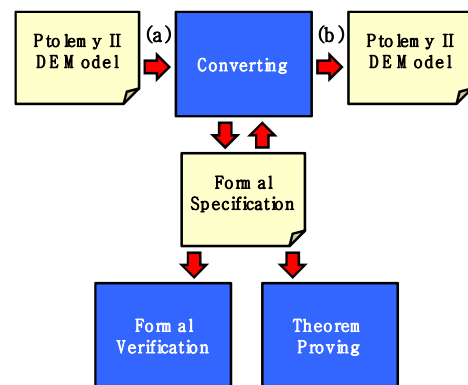


Figure 1. Formal specification of models for verification

The remainder of the paper is organized as follows: Section 2 described the proposed formal specification of DE models in detail. Section 3 compares the proposed specification and the discrete event system specification [8,9], which is a well-known formalism in discrete event simulation. Finally, the conclusions and future work are discussed in Section 4.

II. PTOLEMY DISCRETE EVENT MODEL SPECIFICATION (PDEMS)

This section describes the proposed formal specification of DE models, a *Ptolemy Discrete Event Model Specification* (PDEMS), in detail. It consists of the atomic model specification and the composite model specification.

A. DE Domain

DE domain provides a general environment for time-oriented simulations of systems [10]. In DE domain, models, namely *DE actors*, communicate by sending events across connections. Each event is formed by the token sent and the time at which it was sent. Upon receiving an event, the actor is activated and a reaction takes place. The reaction may result in any state change of the actor and possibly generations of new events. DE scheduler ensures that events are processed chronologically [7].

B. Methods of DEActor Class

In Ptolemy II, an atomic DE model (a DE actor) is implemented as a Java class that inherits `ptolemy.domains.de.kernel.DEActor` class (e.g. `ptolemy.domains.de.lib.Server` class). Within the class, the principal behavior of a DE model is mainly determined by three methods: `prefire`, `fire`, and `postfire`.

`prefire` determines whether the model is ready to execute `fire`, which basically determines the outputs of the model, or not, when external or internal events occur. An *external event* occurs when a *token* (a job from outside of the model) has just arrived at the model. An *internal event* occurs by the model itself. Within `postfire`, the model can make an internal event scheduled to occur at a particular time, using `fireAt` or `fireAfterDelay` methods. Such *event scheduling* is usually used to notify the model itself that the processing on a token has just been finished (and thus some output should be produced). If `prefire` returns false for an event, `fire` including `postfire` will not be executed for the event. Otherwise, `fire` and `postfire` will be executed sequentially. Within `prefire`, the model can read the current input token, its current state, and the global clock time. However, any changes of them may result in misbehavior (i.e. they are read-only in principle). `prefire` returns true by default.

`fire` determines the outputs of the model for internal events. However, the state of the model can also change by the method due to storing of input tokens. For example, a server with a queue has to store a newly arrived (input) token into the queue if it is already processing another token. Thus, the state of the server transitions due to the change in the queue. `fire` may be executed multiple times within a fixed time in Ptolemy II. In order to fix the state of and output of the model within a fixed time, the model has to first save new state temporarily in `fire` and then update their state in `postfire` [10]. Within `fire`, the model's temporary state and output are determined by the current state, the current

input token, and the global clock time. In most cases models first consume input tokens if exist. They then produce outputs if necessary (i.e., if internal events occur). An input token should be consumed for each `fire` execution if exists.

`postfire` determines the time of the next internal event (event scheduling) and the new state of the model. `prefire` (and possibly the other two methods) will be executed on the time of the next event. If the model has nothing to process, the time would be infinite (∞). In the case the model will be in the idle state until any input tokens arrive. The time and state are determined based on the current state of and the temporary state (determined in `fire`) of the model, and the global clock time. The time of the next event is generally stored within the model since this time is used within `fire`; `fire` will produce an output if the global clock time is equal to the time stored.

C. Atomic Model Specification

In PDEMS, an atomic DE model is formally represented in the following form:

$$PDEMR = \langle X, Y, S, S_T, prefire, fire, postfire \rangle,$$

where

X is the set of input tokens,

Y is the set of output tokens,

S is the set of sequential states,

S_T is the set of *temporary states*,

$prefire: Q \times (X \cup \{\emptyset\}) \rightarrow \{0, 1\}$ is the pre-fire function,

$fire: Q \times (X \cup \{\emptyset\}) \rightarrow (Y \cup \{\emptyset\}) \times S_T$ is the fire function,

$postfire: Q \times S_T \rightarrow S \times (\mathbb{R}^+ \cup \infty)$ is the post-fire function,

$Q = \{(s, t) \mid s \in S, t \in \mathbb{R}^+ \cup \infty\}$ is the set of total states,

t is the global clock time.

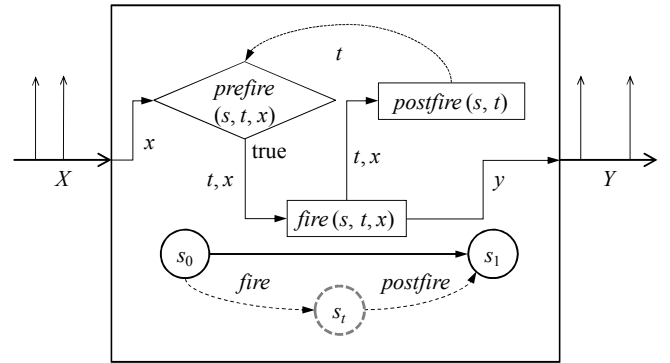


Figure 2. Elements in an atomic PDEMS

The interpretation of these elements is illustrated in Fig. 2. An atomic PDEMS describes the behavior of an atomic DE model. The *pre-fire function* is an abstraction of `prefire` which determines whether the model will execute the fire and the post-fire functions or not for an external or internal event. The function uses the current state s of the model, the global clock time t , and an input token x , and returns 0 or 1. The model will not execute the fire and post-fire functions for an event if the pre-fire function returns 0 (false) for the event. Otherwise, they will be executed sequentially.

The *fire function*, which is an abstraction of *fire*, determines the output of the model and the temporary state s_t for an event. They are determined by the current state s , the global clock time t , and an input token x . The temporary state is used in the post-fire function, to determine the final state of the model for the current event. An input token is consumed for each execution of the fire function.

The *post-fire function* is an abstraction of *postfire*. It determines the time of the next internal event and the new state of the model based on the current state s , the global clock's time t , and the temporary state s_t . The pre-fire function (and possibly the other two functions) will be executed on the time of the next event. Since input tokens are consumed in the fire function, the model cannot read any input tokens directly in the post-function. However, the tokens can be read through accessing s_t if they were stored in the fire function.

The proposed specification is much more rigorous than the Java implementation of DE models. In the pre-fire function, the model state, the global clock time, and the current input token are read-only variables, while they can be altered in *prefire* method. In the implementation, the model state and the temporary state are not discriminated explicitly. Thus, the state of a model might change within a fixed time. On the other hand, they are clearly distinguished (as S and S_T) in the specification. Any altering of the model state or any event scheduling within the fire function does not allowed in the specification. In the post-fire function, there is no way to access the current input token. On the other hand, it is possible to access the token, through a method *input.get*.

D. Example: Server

A server model, *ptolemy.domains.de.lib.Server*, can be formally represented in PDEMS as follows:

$PDEMR_{serviceTime} = \langle X, Y, S, S_T, prefire, fire, postfire \rangle$,
where

$X = V$ (an arbitrary set),

$Y = V$,

$S = S_T = (\mathbb{R}^+ \cup \infty) \times V^+$,

$prefire(nextTimeFree, (v_1, v_2, \dots, v_n), t, v) = 1$, (a)

$fire(nextTimeFree, (v_1, v_2, \dots, v_n), t, v)$

$= (\emptyset, nextTimeFree, (v_1, v_2, \dots, v_n, v))$
if $v \neq \emptyset \ \& \ nextTimeFree \neq t$, (b)

$= (v_1, \infty, (v_2, \dots, v_n, v))$
if $v \neq \emptyset \ \& \ nextTimeFree = t$, (c)

$= (v_1, \infty, (v_2, \dots, v_n))$
if $v = \emptyset \ \& \ nextTimeFree = t$, (d)

postfire

$(nextTimeFree, (v_1, \dots, v_n), t, nextTimeFree_t, (v_1, \dots, v_m))$
 $= (t + serviceTime, (v_1, v_2, \dots, v_m), t + serviceTime)$

if $nextTimeFree_t = \infty \ \& \ 0 < m$, (e)
 $= (nextTimeFree_t, (v_1, v_2, \dots, v_m), nextTimeFree)$
otherwise. (f)

This model receives and produces arbitrary tokens. Its state consists of the time of the next event, *nextTimeFree*, and a queue to store input tokens, V^+ . It starts with *nextTimeFree* = ∞ and an empty queue (i.e. $s_{initial} = (\infty, ())$). The model's pre-fire function always returns 1 (see (a)). That is, it executes the fire and post-fire functions for each event.

When the model receives a token (i.e. an external event occurs), it first puts the token into the queue (see (b) and (c)). The token will start to be processed in the post-fire function even if the model is in the idle state. When the model has just finished the process on a token (i.e. an internal event occurs), it outputs the processed token and then set the time of the next event as infinite (see (c) and (d)). That is, it transitions to the idle state temporarily. If any token exists in the queue, the model will start to process on the token in the post-fire function. Note that such changes in the queue or of the time are not directly reflected on the state of the model. They saved in the temporary state. The state changes of the model occur only in the post-fire function.

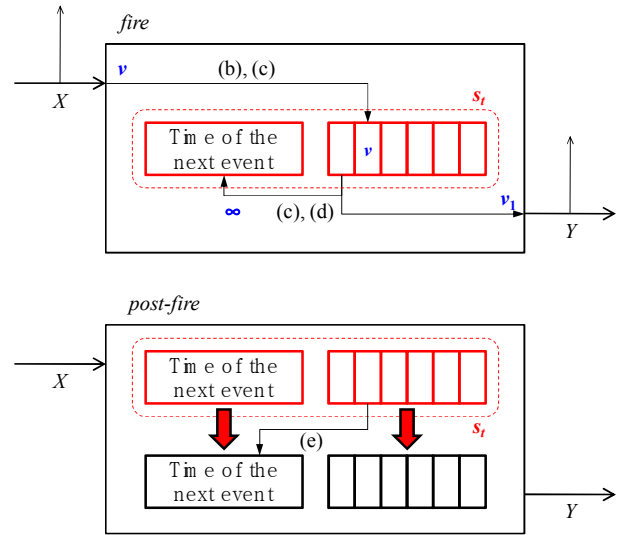


Figure 3. Internal behavior of server in fire and post-fire functions

After each execution of the fire function, the model makes an internal event scheduled to occur *serviceTime* later (i.e., at $t + serviceTime$) if the queue of the temporary state is not empty and *nextTimeFree* in the temporary state is infinite (see (e)). *nextTimeFree* = ∞ in the temporary state indicates that the model has just produces an output for the event in the fire function (cf. (c) and (d)) or has just received a token for the first time. Thus, if there are any tokens in the queue, the model starts to process the first of them. Otherwise, the model just updates its state from temporary state (see (f)). Note that *nextTimeFree* could not be finite in (f). If an external event has occurred, the queue could not be empty (cf. (b) and (c)). If an internal event has occurred, *nextTimeFree* is set as infinite (cf. (d)). While this model does not read the current state in the determination of the new state, models may use the state if necessary. The internal behavior in the fire and the post-fire functions is illustrated in Fig. 3.

E. Composite Actor Specification

A complex model consists of a large number of atomic models. Like atomic models, a model may also define an external interface, which is called its *hierarchical abstraction*. PDEMS also provides a means to represent such models. A composite model comprised of a number of atomic models is represented in the following form:

$$N = \langle X, Y, D, \{M_d\}, P, EIC, EOC, IC \rangle$$

where

X is the set of input tokens,

Y is the set of output tokens,

D is the set of the model references.

For each $d \in D$,

M_d is an atomic model,

$P = \{(d, p) \mid p \text{ is a unique non-negative integer in } N\}$ is the *priority of the model*.

$EIC \subseteq \{(N, d) \mid d \in D\}$ is the *external input coupling*,

$EOC \subseteq \{(d, N) \mid d \in D\}$ is the *external output coupling*,

$IC \subseteq \{(a, b) \mid a, b \in D, a \neq b\}$ is the *internal coupling*.

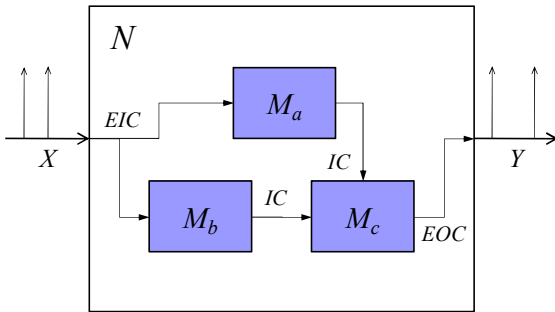


Figure 4. Elements in a coupled PDEMS

The interpretation of these elements is shown in Fig. 4. A composite PDEMS describes the structure of a DE model. The external input coupling connects the external inputs of the coupled mode to inputs of sub-models (e.g. a coupling from N to M_a in Fig. 4). The external output coupling connects outputs of sub-models to the outputs of the coupled model (e.g. a coupling from M_c to Y). The internal coupling connects outputs of sub-models to inputs of sub-models (e.g. a coupling from M_b to M_c). However, *no direct feedback loops* are allowed (i.e. $a \neq b$ in IC).

F. Example: Inspection

The *Inspection Model* (Fig. 5), included in the Ptolemy demos, illustrates a famous paradox in probability called the inspection paradox [6]. The inspection model can be represented in PDEMS as follows:

$$N = \langle X, Y, D, \{M_d\}, P, EIC, EOC, IC \rangle$$

where

$X = Y = \emptyset$,

$D = \{\text{Poisson Bus Arrivals, Regular Bus Arrivals, Passenger Arrivals, Poisson Waiting Time, Regular Waiting Time, SingleEvent0, Report Poisson Average, Report Regular Average}\}$,

the set of atomic models = $\{M_{\text{Poisson Bus Arrivals}}, M_{\text{Regular Bus Arrivals}}, M_{\text{Passenger Arrivals}}, M_{\text{Poisson Waiting Time}}, M_{\text{Regular Waiting Time}}, M_{\text{SingleEvent0}}, M_{\text{Report Poisson Average}}, M_{\text{Report Regular Average}}\}$,

$P = \{(\text{Poisson Bus Arrivals}, 1), (\text{Regular Bus Arrivals}, 2), (\text{Passenger Arrivals}, 3), (\text{Poisson Waiting Time}, 4), (\text{Regular Waiting Time}, 5), (\text{SingleEvent0}, 6), (\text{Report Poisson Average}, 7), (\text{Report Regular Average}, 8)\}$,

$EIC = EOC = \emptyset$,

$IC = \{(\text{Poisson Bus Arrivals}, \text{Poisson Waiting Time}), (\text{Regular Bus Arrivals}, \text{Regular Waiting Time}), (\text{Passenger Arrivals}, \text{Poisson Waiting Time}), (\text{Passenger Arrivals}, \text{Regular Waiting Time}), (\text{Poisson Waiting Time}, \text{Report Poisson Average}), (\text{Regular Waiting Time}, \text{Report Regular Average}), (\text{SingleEvent0}, \text{Report Poisson Average}), (\text{SingleEvent0}, \text{Report Regular Average})\}$.

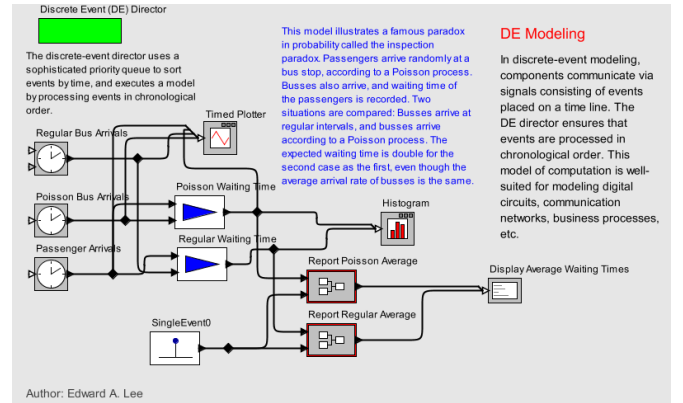


Figure 5. Inspection demo of Ptolemy II

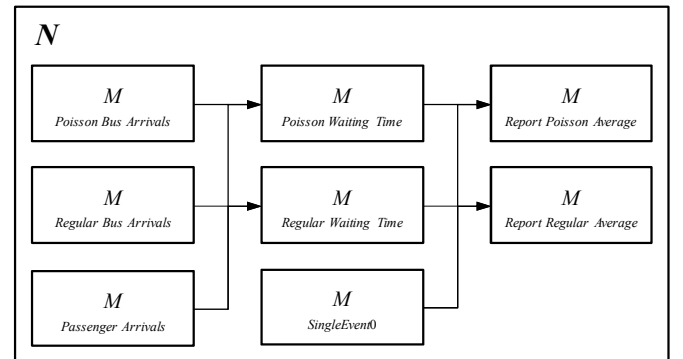


Figure 6. Composited PDEMS for inspection demo

Since the model is the top-level model, it does not have any inputs and outputs. Thus, there is no connection between atomic models and the coupled model (i.e. $EIC = EOC = \emptyset$). To simplify the representation, we have excluded three models related to output analysis: *Timed Plotter*, *Histogram*, and *Display Average Waiting Times*. Thus, the model contains eight atomic models. Their names and (arbitrary) priorities are listed in D and P , respectively. Couplings between them are specified IC . The outputs of three generator models – $M_{\text{Poisson Bus Arrivals}}$, $M_{\text{Regular Bus Arrivals}}$, and $M_{\text{Passenger Arrivals}}$ – are connected to the inputs of two waiting time recorder models: $M_{\text{Poisson Waiting Time}}$ and $M_{\text{Regular Waiting Time}}$. The outputs of the two recorder models are connected to the inputs of two average computation models:

$M_{Report\ Poisson\ Average}$ and $M_{Report\ Regular\ Average}$. The output of $M_{SingleEvent0}$, which generates a single signal to report simulation results, is connected to the inputs of the two computation models. The model is illustrated as Fig. 6.

G. Hierarchical Composition

The hierarchical abstraction of Ptolemy II enables modelers to construct complex models hierarchically. A composite model comprised of atomic models can be used as an atomic model in a higher-level. For example, *Report Poisson Average* and *Report Regular Average* models in the inspection demo are hierarchical abstractions of composite models as shown in Fig. 7. Each report model collects a running average of the incoming data, and when a trigger is received, samples that data, converts it to a string, and appends it to a caption to report the final result [6].

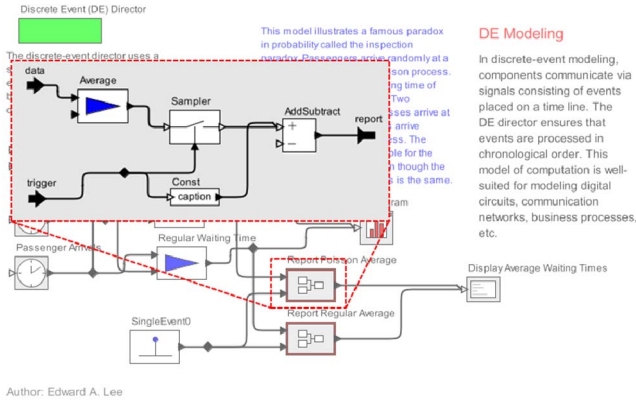


Figure 7. Hierarchical abstraction in inspection demo of Ptolemy II

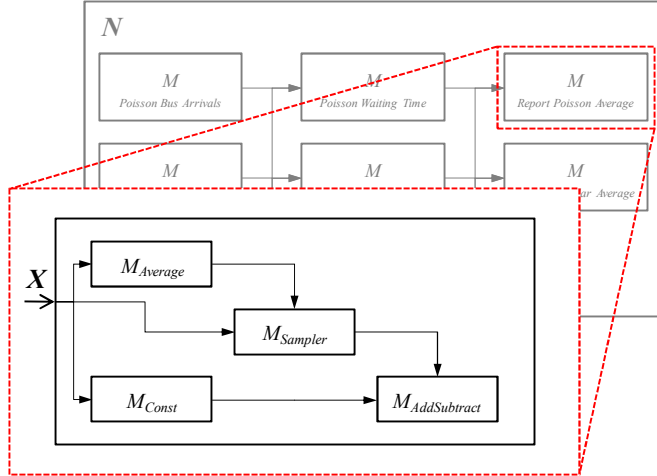


Figure 8. Hierarchical representation of inspection demo in PDEMS

Although a composite PDEMS primarily describes the structure of a DE model, we can intuitively know that it can describe the behavior of the model, like an atomic PDEMS. The input and output sets are explicitly specified in the representation. The state of a composite PDEMS is the Cartesian product of all the state of its atomic models. Thus, its state transitions are based on the functions of the models. External input tokens are consumed by the models connected

in *EIC* and output tokens are produced by the models connected in *EOC*. The time of the next internal event is the minimum value among those of the models. Therefore, PDEMS can represent models hierarchically. Fig. 8 shows a hierarchical representation of the inspection demo.

III. DISCRETE EVENT SYSTEM SPECIFICATION (DEVS) & PDEMS

In this section, we compare the proposed specification and the discrete event system specification (DEVS) which is a well-known formalism in discrete event simulation.

A. DEVS Overview

The DEVS formalism [8,9] is a theoretically well-grounded means of expressing modular discrete event simulation models developed by Zeigler. An atomic DEVS model is defined as a 7-tuple:

$$DEVS = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle,$$

where

- X is the set of input events,
- Y is the set of output events,
- S is the set of sequential states,
- δ_{ext} is the external transition function,
- δ_{int} is the internal transition function,
- λ is the output function,
- ta is the time advanced function.

The DEVS environment supports building models in a hierarchical and modular manner, where the term “modular” means the description of a model in such a way that it has recognized input and output ports through which all interaction with the external world is mediated. This property enables the hierarchical construction of models so that complex models can be easily developed.

B. DEVS & PDEMS

Atomic models represented in PDEMS can be also represented in DEVS and vice versa since both of them model discrete event systems. The input and output sets of the former models are identical to those of the latter models. Their state sets are probably identical to those of DEVS models, except for *sigma*. While each atomic DEVS refers to a local clock, the global clock is used in PDEMS. For an external event without any internal events, $fire(s_0, t_0, x) = (\emptyset, s_t)$ and $postfire(s_0, t_0, s_t) = (s_1, t_1)$ in PDEMS are corresponding to $\delta_{ext}(s_0, e, x) = s_1$ and $ta(s_2) = t_1 - t_0$ in DEVS. That is, they transition from state s_0 to state s_1 for the event and will internally transition $t_1 - t_0$ later. For an internal event without any external events, $fire(s_0, t_0, \emptyset) = (y, s_t)$ and $postfire(s_0, t_0, s_t) = (s_1, t_1)$ in PDEMS are corresponding to $\lambda(s_0) = y$, $\delta_{int}(s_0) = s_1$, and $ta(s_2) = t_1 - t_0$ in DEVS. For an external event and internal event at the same time, $fire(s_0, e, x) = (y, s_t)$ and $postfire(s_0, t_0, s_t) = (s_1, t_1)$ in PDEMS are corresponding to $\lambda(s_0) = y$, $\delta_{con}(s_0, x) = s_1$, and $ta(s_1) = t_1 - t_0$ in DEVS.

IV. CONCLUSIONS AND FUTURE WORK

This paper presents PDEMS that can represent DE models of Ptolemy II in a more formal form. The atomic model specification describes the behavior of a model, while the structure of a model can be represented by the composite model specification. PDEMS supports the hierarchical abstraction of Ptolemy II in which a composite model can be used as an atomic model at a higher-level.

While the current form of the specification may be incomplete, it may be used as a basis for the formal verifications of DE models after further refinement. We will develop a Java-to-PDEMS converting tool (Fig. 1(a)) and a PDEMS-to-code converting tool (Fig. 1(b)), based on a text-based specification of PDEMS. We will then study on formal verification and theorem proving techniques for PDEMS. We will also study on formal specifications and verifications of Ptolemy II domains, such as the continuous time, process network, and finite state machine.

REFERENCES

- [1] J. Eker, J. Janneck, E.A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S.R. Sachs, and Y. Xiong. "Taming Heterogeneity – the Ptolemy Approach," *Proceedings of the IEEE*, vol. 91, no.1, pp. 127-144, Jan. 2003.
- [2] C. Brooks, E.A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng, "Heterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to Ptolemy II)," EECS Department, University of California, Berkeley, UCB/EECS-2008-28, Apr. 2008.
- [3] C. Brooks and E.A. Lee, "The Center for Hybrid and Embedded Software Systems." <http://chess.eecs.berkeley.edu/charter.pdf>
- [4] A.R. Girard, A.S. Howell, and J.K. Hedrick, "Model-Driven Hybrid and Embedded Software for Automotive Applications." http://www.cse.wustl.edu/~cdgill/RTAS04/modes_ucb.pdf
- [5] R. Alur, F. Ivancic, J. Kim, I. Lee, and O. Sokolsky, "Generating Embedded Software from Hierarchical Hybrid Models," *ACM SIGPLAN Notices*, vol. 38, no. 7, pp. 171-182, Jul. 2003.
- [6] The Ptolemy Project. <http://ptolemy.eecs.berkeley.edu/>
- [7] L. Muliadi, "Discrete Event Modeling in Ptolemy II," MS Report, Dept. of EECS, University of California, Berkeley, CA 94720, May 1999.
- [8] B.P. Zeigler, T.G. Kim, and H. Praehofer, *Theory of Modeling and Simulation*, 2nd Ed., Academic Press, 2000.
- [9] B.P. Zeigler. *Object-oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic Systems*, Academic Press, 1990.
- [10] C. Brooks, E.A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng, "Heterogeneous Concurrent Modeling and Design in Java (Volume 3: Ptolemy II Domains)," EECS Department, University of California, Berkeley, UCB/EECS-2008-37, Apr. 2008.