PAPER *Special Section on Concurrent System Technology*

Performance Evaluation of Concurrent System Using Formal Model: Simulation Speedup

Wan Bok LEE[†,††a)], ***Regular Member and*** **Tag Gon KIM**[††b)], ***Nonmember***

SUMMARY Analysis of concurrent systems, such as computer/communication networks and manufacturing systems, usually employs formal discrete event models. The analysis then includes model validation, property verification, and performance evaluation of such models. The DEVS (Discrete Event Systems Specification) formalism is a well-known formal modeling framework which supports specification of discrete event models in a hierarchical, modular manner. While validation and verification using formal models may not resort to discrete event simulation, accurate performance evaluation must employ discrete event simulation of formal models. Since formal models, such as DEVS models, explicitly represent communication semantics between component models, their simulation cost is much higher than using simulation languages with informal models. This paper proposes a method for simulation speedup in performance evaluation of concurrent systems using DEVS models. The method is viewed as a compiled simulation technique which eliminates runtime interpretation of communication paths between component models. The elimination has been done by a behavior-preserved transformation method, called model composition, which is based on the closed under coupling property in DEVS theory. Experimental results show that the simulation speed of transformed DEVS models is about 14 times faster than original ones.

key words: *simulation speedup, performance evaluation, formal modeling*

1. Introduction

Man-made dynamic systems, in which a set of processes runs concurrently in a cooperative manner, are no longer specified and then analyzed by classes of differential equations. The states of such systems are changed in response to occurrences of discrete events which take place at any time. Examples of such systems include computer networks, manufacturing systems and various forms of high-level controller systems. These concurrent systems might have safety critical abnormalities such as deadlock, race condition or violation of the mutual exclusive access that might result in enormous damages. Thus, the importance of proof of functional correctness of these systems arises. Formal models play an important role in such proof due to the fact that models are manipulated mathematically, thereby making it possible to develop automated tools. Although formal verification of large-scale systems is still practically impossible, automated analysis tools such as SPIN [1], SMV [2], or UPPAAL [3] have been successfully applied to some limited problems. Development of such tools to solve general problems is still left as an open problem.

In addition to the logical analysis described above performance evaluation and virtual prototyping are also performed in the development stages of a concurrent system. Thus a unified modeling framework in which a formal model is used for logical analysis, performance evaluation and virtual prototyping is most desirable. However, many formal models are only well suited in the stage of logical analysis; they could not be applied to the other stages such as performance evaluation and/or virtual prototyping. This limitation has led researchers to extend existing formalisms or to develop a new formalism which can be employed in all three stages of discrete event systems development. Examples include Statechart [4], Timed Petri Net (TPN) [5] and Real-time DEVS [11].

Discrete event simulation has been widely used for accurate performance evaluation of concurrent systems. Since such simulation requires heavy computation time, simulation speed is of great importance in practical performance evaluation. Usually, discrete event models for simulation are not specified in a formal manner. Instead they are specified informally, based on modeling world views, such as event-oriented, process-oriented and activity-oriented views, which can be simulated by corresponding simulation languages [7], [8]. Being informal, such models can not be directly used in formal verification/validation which requires mathematical manipulation of structure and/or behavior of models. The main advantage of using simulation languages with such informal models is easiness in model development and low computation time in simulation runs, compared with using formal models. On the contrary, formal models used in the logical analysis stage can be suitably used for simulation of performance evaluation. To do so significantly saves cost for development and verification of new models only for simulation purpose. An inherent drawback to use formal models in discrete event simulation is slow simulation speed due to repeated interpretation of complex model semantics

Manuscript received March 29, 2003.
Manuscript revised June 29, 2003.
Final manuscript received July 22, 2003.

[†]The author is with the Department of Information Security, Joongbu University, Chungnam, Korea.
[††]The authors are with the Department of Electrical Engineering and Computer Science, Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea.
a) E-mail: wblee@joongbu.ac.kr
b) E-mail: tkim@ee.kaist.ac.kr

in simulation run time. An example of such interpretation is that connection information between component models specified in model semantics is implemented as a message passing between the two during simulation run time.

The purpose of this paper is to address speedup of simulation run time using a formal model specified by the DEVS (Discrete Event Systems Specification) formalism [6]. Being based on set theory the DEVS formalism specifies discrete event models in a hierarchical, modular manner. The formalism has been widely used in simulation modeling and analysis of discrete event systems in the systems research area. Within the formalism two model classes are defined: atomic and coupled models. An atomic model represents the dynamics of a non-decomposable component of a discrete event system in a timed state transition; a coupled model is a collection of components, either atomic or coupled, the specification of which includes a list of components and their coupling relation. This paper proposes a method for simulation speedup in performance evaluation of concurrent systems using DEVS models. The method is viewed as a compiled simulation technique which eliminates run-time interpretation of communication paths between component models. The elimination has been done by a behavior-preserved transformation method, called model composition, which is based on the closed under coupling property in DEVS theory. Since a finally composed model, by applying a series of such transformations, has no interaction with any model simulation of such model markedly reduce simulation time without loss of accuracy.

The rest of this paper is organized as follows. Related works are presented in Sect. 2 and the model of computation, DEVS formalism is briefly introduced in Sect. 3. In Sect. 4, run-time simulation overheads are classified and analyzed. The proposed method for simulation speedup is described in Sect. 5. After showing the experimental results in Sect. 6, conclusion follows in Sect. 7.

2. Related Works

A DEVS-based design methodology is a unified framework on which the jobs of all the stages in a system development process can be performed. The framework, as shown in Fig. 1, provides a basis for modeling of discrete event systems with which logical analysis, performance evaluation, and implementation can be performed—all with the DEVS formalism [6], [13]. The logical analysis (or verification) checks lower-level operational specification against higher-level assertional specification [9]. Performance evaluation within the framework employs object-oriented simulation of hierarchical DEVS models for which simulation environments running on different languages are developed. Such environments include DEVSim++ [10] of a C++ based environment and DEVSimjava of a JAVA based environment. Implementation within the framework is no longer an error-prone transformation of a DEVS model to an executable program code. Instead, the model used in the designed phase itself is an implementation which runs in real-time on a real-time simulation engine [11], [12].

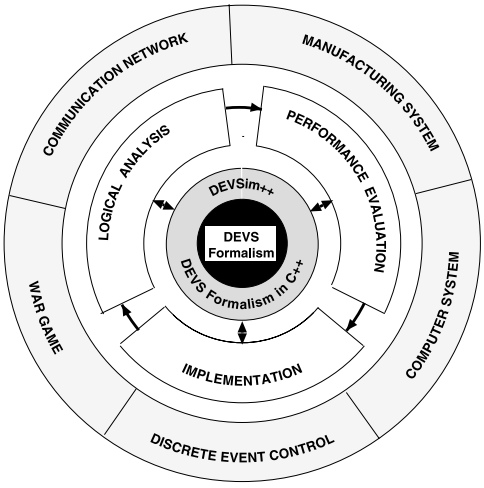

Fig. 1 DEVS methodology for system development.

Simulation, which has been mainly used for the purpose of performance analysis, may also be used as a means of verification method. In particular, when the system has many components and its overall state space is large, then the simulation becomes the only applicable method of verifying the system in most cases. As simulation requires much computation time and cost in general, the methods or algorithms which can accelerate its execution speed is of great importance. For a decade, several approaches have been developed to accelerate simulation speed of DEVS models.

Parallel or distributed simulation is one of the solutions for such a problem [15] However, this method requires additional cost in hardware as well as extra work for parallelization of existing sequential models. Hybrid simulation [16] is another approach to tackle the problem. This method is based on a transformation of the steady state behavior of a DEVS model into an equivalent analytic model. Although the method can be effectively used for simulation of large and complex systems the transformation is applicable under some assumptions on processes [16]. Transformation of DEVS model structure for simulation speedup has been attempted in [14]. The approach is to reduce message traffic during simulation run by transformation of hierarchical DEVS model structure into fatter one. The approach achieved speedup of about 25% faster compared with simulation time using the original model.

Compiled simulation technique, which has been successfully applied in the area of logic simulation, could significantly reduce simulation time by eliminat-

ing the execution time overhead originated from repeated interpretation of complex data structure of a model in run-time [7]. Although the method can increase simulation speed significantly, the work has not been applied to simulation of a formal model. The model in the work was just C-language based programming code and the compilation process was not defined in a formal way.

3. Model of Computation

3.1 DEVS Formalism: A Brief Introduction

A set-theoretic formalism, the DEVS formalism, specifies discrete event models in a hierarchical and modular form. Within the formalism, one must specify 1) the basic models from which larger ones are built, and 2) how these models are connected together in a hierarchical fashion. A basic model, called an atomic model, has specifications for the dynamics of the model. An atomic model AM is specified by a 7-tuple [13]:

$$AM = < X,\ S,\ Y,\ \delta_{ext},\ \delta_{int},\ \lambda,\ ta >,$$

where
- X : a set of input events;
- S : a set of sequential states;
- Y : a set of output events;
- $\delta_{ext.}$: $Q \times X \rightarrow S$, an external transition function, where $Q = \{(s,e) | s \in S \ and \ 0 \leq e \leq ta(s)\}$, total state set of M;
- $\delta_{int.}$: $S \rightarrow S$, an external transition function;
- λ : $S \rightarrow Y$, an output function;
- ta : $S \rightarrow R^{+}_{0,\infty}$ (non-negative real number), time advance function.

The second form of the DEVS model, called a coupled model (or coupled DEVS), is a specification of the hierarchical model structure. It describes how to couple component models together to form a new model. This latter model can itself be employed as a component in a larger coupled model, thereby giving rise to the construction of complex models in a hierarchical fashion. Formally, a coupled model CM is defined as [13]:

$$CM = < X_{self}, Y_{self},\ D,\ \{M_i\},\ \{I_i\},\ \{Z_{i,j}\},\ SELECT >,$$

where
- X_{self} : a set of input events;
- Y_{self} : a set of output events;
- D : a set of component names;
- for each $i \in D$,
  - M_i : a component DEVS model, atomic or coupled;
  - I_i : the influences of i;
- for each $i, j \in D \cup \{self\}$
  - $Z_{self,j} : X_{self} \rightarrow X_j$, $\forall j \in D$, external input coupling;
  - $Z_{i,self} : Y_i \rightarrow X_{self}$, $\forall i \in D$, external output coupling;
  - $Z_{i,j} : Y_i \rightarrow X_j$, $\forall i, j \in D$, an i-to-j internal coupling;
- $SELECT : 2^D$ - $\emptyset \rightarrow D$, tie-breaking function.

3.2 An Example Model

Figure 2 shows a simple model of a buffer and a cascaded processor, named *Single Server Queue (SSQ)*. The model is commonly used as a component in computer and/or communication systems. *EF* is an environment model supplying stimuli to *PEL* and collecting outputs. *Buff* controls the flow of incoming problems, or packets, which are to be sent to *Proc*. The input port *in* of *Buff* is for receiving incoming problems and the input port *ready* of *Buff* receives an acknowledge signal from *Proc* indicating that *Proc* is free. *Buff* releases problems one by one as *Proc* is free, while *Proc* holds each problem for some time units and outputs to *Buff*. From the informal description of structure and behavior, we formally specify the *Buff* and *Proc* model as follows:

Buff = < X, Y, S, $\delta_{ext,}\delta_{int,}\lambda$, ta >
- $X = \{in\}$
- $Y = \{out\}$
- $S = \{(n, pstat) | n \in I, pstat \in \{B, F\}\}$
- $\delta_{ext}((n, pstat), e, in) = (n + 1, pstat)$
- $\delta_{ext}((n, B), e, ready) = (n, F)$
- $\delta_{int}((n, F)) = (n - 1, B)$, if $n > 0$
- $\lambda((n, F)) = out$, if $n > 0$
- $ta(n, pstat) = 0$, if $n > 0$ and $pstat = F$

Proc = < X, Y, S, $\delta_{ext,}\delta_{int,}\lambda$, ta >
- $X = \{req\}$
- $Y = \{done\}$
- $S = \{(stat) | stat \in \{B, F\}\}$
- $\delta_{ext}(F, req) = B$
- $\delta_{int}(B) = F$
- $\lambda(B) = done$
- $ta(B) = service\_time$

Once *Buff* and *Proc* are developed, the coupled

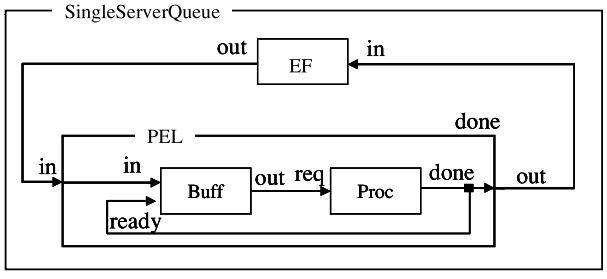

Fig. 2 Example DEVS model, *Single Server Queue*.

model *PEL* can be specified by defining components and coupling relations as defined in the coupled model. Thus, the *PEL* is specified as:

$PEL = < X,\ Y,\ D,\ \{M_i\},\ \{I_i\},\ \{Z_{i,j}\},\ SELECT >$
$X_{self} = \{in\}$
$Y_{self} = \{out\}$
$D = \{Buff, Proc\}$
$I_{Buff} = \{Proc\}; I_{Proc} = \{Buff, PEL\}$
$I_{PEL} = \{Buff\}$
$Z_{PEL,Buff}(in) = in; Z_{Buff,Proc}(out) = req$
$Z_{Proc,Buff}(done) = ready$
$Z_{Proc,PEL}(done) = out$
$SELECT(Buf, Proc) = Proc$

Similarly, the overall model *Single Server Queue* can be specified by connecting the two component models, *EF* and *PEL*.

4. Simulation Overhead Analysis

Simulation of DEVS models requires a simulation algorithm which interprets dynamics of model's specification. In DEVS theory, the algorithm is implemented as abstract processors each of which is associated with a component of an overall hierarchical DEVS model in an one-to-one manner. There are two types of processors: a *simulator* for an atomic model and a *coordinator* for a coupled model. Simulation of DEVS models is performed in such a way that event scheduling and message passing between such component models are done in a hierarchical manner.

Basically the abstract simulator algorithm for DEVS models, which is introduced in Appendix A, repeatedly performs two tasks: 1) an *event synchronization task*, and 2) a *scheduling task*. In the event synchronization task, a *next scheduled component,* which has the earliest next schedule time among the components, generates an output event with a state transition specified in its internal state transition function. At the same time, all the components whose input events are coupled with the output event are influenced, i.e. they change their state variables specified in their external state transition functions. To find such components a coupling scheme specified in a coupled model is to be referred. Those influenced components and the influencing component are simply named as *influencees* and *influencer*, respectively. On the other hand, the *scheduling* task, which follows the event synchronization task, determines the next scheduled component and its activation time.

The above two tasks are performed by means of the four types of messages such as (x), (y), (*) and (done) messages. In the event synchronization task, (x) or (y) messages are created from an influencer and are translated as (x) messages to the influencees noticing that new external input events have been arrived. Then both of the influencer and the influencees perform state transitions. Similarly, all the influencees and the influencer send (done) messages to relevant components, as soon as they finish a state transition. The next scheduled times of components are carried on (done) messages. (*) message is delivered to the next scheduled component that is going to do the next event synchronization task. In summary, (x) and (y) are related to the event translation task while (done) and (*) are to the scheduling task.

To evaluate how much time is taken for each task in a simulation process, we first divided the overall simulation run-time into three parts: 1) overhead for updating state variables, 2) overhead for event translation 3) overhead for scheduling. Noticeable in this classification is that the time taken in the event synchronization

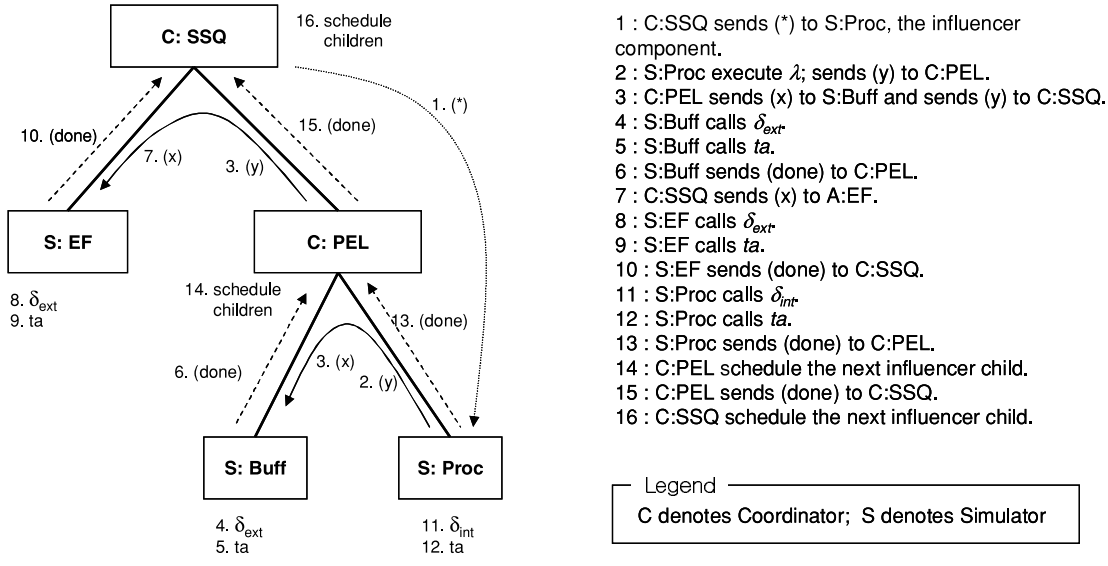

Fig. 3 A process of model interpretation.

Table 1 Classification of internal tasks.

	State Variables Updating	Event Translation	Scheduling
Original model	4,8,11	2,3,7	1,5,6,9,10,12,13,14,15,16
Composed model	4,8,11		1,5,9,12,16

Table 2 Time taken for each overhead.

Ex. Model	State Variables Updating (sec.)	Event Translation (sec.)	Scheduling (sec.)	Total Time (sec.)
SSQ	1.6 (6%)	8.9 (32%)	17.8 (63%)	28.3 (100%)
CSMA/CD	2.1 (7%)	8.5 (29%)	18.6 (64%)	29.2 (100%)
CHCS	2.2 (7%)	8.9 (27%)	21.6 (66%)	32.7 (100%)

task is divided into two parts: the overhead for updating state variables and the overhead for event translation. Among the three overheads, the message passing activity has no relation with the first overhead, but it has close relations with the second and the third overheads as explained before.

To show the interpretation process of the abstract simulator algorithm in Appendix A, we revisit the example in Fig. 2. Let's assume that the current influencer is *Proc* and *Proc* sends an output event, *done* to the other atomic models. Then, this event is translated as an input event, *in* of *EF* and as *ready* of *Buff*. In this process, (y) and (x) messages are generated and routed internally as in Fig. 3. As these messages are passed, some other tasks happen during the simulation as shown in Fig. 3. They can be categorized into three overheads as shown in Table 1. Much of the internal tasks may be removed if a compiled simulation technique, called composition, is applied. This will be explained in the next section.

To measure the time taken for each overhead in a simulation run, we experimented with three example models, as in Table 2, with an object-oriented simulation environment of DEVSim++ [10]. The examples will be explained in more detail in a later section. Experiments were done 100 times and mean values were taken. The *GNU* profile tool, *gprof* was used in order to measure the time taken for each overhead. The experiment revealed that only about 7% of the overall time has been spent for updating state variables, whereas the other time has been used for the scheduling and the event translation tasks.

5. Composition Method for Simulation Speedup

5.1 Composition

The two dominant overheads, i.e. event translation and scheduling overheads, are closely related to the mechanism of passing the four types of messages. They can be significantly reduced by applying a kind of compiled simulation technique, *composition*. Composition is a process of merging all the component models belonging to a coupled model. As the composed model is also an atomic model, we can get a finally composed model of the form of an atomic model after applying a series of composition.

Notice that our composition method is different with the conventional composition operation in the area of static analysis. The state explosion problem, which is one of the most serious problems of static analysis, does not happen in our composition. While the composition of static analysis generates all the reachable states and build a reachability tree, our composition just merges the procedure rules of each component. The next states need not to be explored during our composition. They can be determined at run-time by executing a procedure rule, i.e. a state transition function. Therefore no state explosion problem happens. Simulation model is sufficient if it can determine the next state, while the analysis model should determine all possible next states.

Figure 4 illustrates the effect of our composition. When the model of normal hierarchical structure is to be simulated, it need to be interpreted during the simulation run-time thus the four types of messages happen constantly. Moreover the four characteristic functions of each component should be called in the middle of simulation run one by one. On the contrary, the composed model does not employ any model interpretation in run-time; every characteristic function is called at once and message passing does not take place in run-time. Following is the definition of our composition operation.

Definition 1 (Composition): Let a coupled model $CM = < X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, SELECT >$ and each component, M_i be an atomic model such that $M_i =< X_i, Y_i, S_i, \delta_{exi,i}\delta_{ini,i}\ \lambda_i, ta_i > and\ i \in D$. Then a composed model M is defined as follows:

$M =< X, Y, S, \delta_{ext},\delta_{int},\lambda, ta >$ where each of the tuple is determined as:
(1) $X = X_{self}$;
(2) $Y = Y_{self}$;
(3) $S = \times_{i \in D} Q_i \times D$, where
$Q_i = \{(s_i, \sigma_i) | s_i \in S_i, 0 \le \sigma_i \le ta_i(s_i)\}$ and $i^* \in D$ is the name of the next scheduled component;

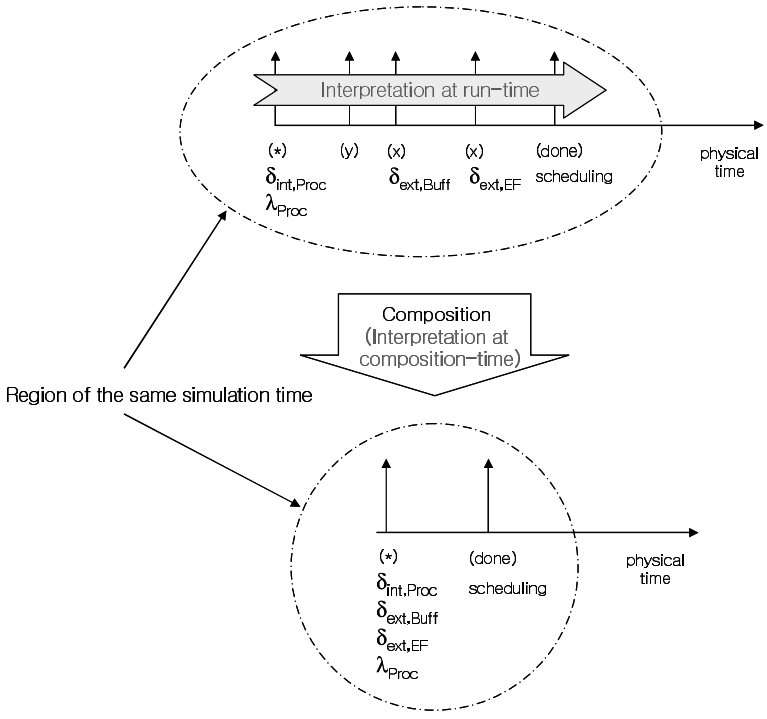

Fig. 4 Hierarchical simulation (top) processes five events and compiled simulation (bottom) processes two events at the same simulation time.

(4) $ta(s) = min\{\sigma_i | i \in D\}$;
(5) $\lambda(s) = Z_{i^*,self}(\lambda_{i^*}(s_{i^*}))$, if $self \in I_{i^*}$;
Let $\sigma_{min} = min\{\sigma_i | i \in D\}, e_i = ta_i(s) - \sigma_i$, and
$IMM(\ldots,\sigma_i,\ldots) = \{i \in D | \sigma_i = \sigma_{min}\ \}$.
(6) $\delta_{ext}(s,e,x) = (\ldots,(s_i',e_i'),\ldots,i^{*'})$ where
$(s_i',e_i') = (\delta_{ext,j}(s_j,e_j+e,Z_{self,j}(x),0)$,
for $j \in I_{self}$,
$(s_i',e_i') = (s_j,e_j+e)$, otherwise,
$i^{*'} = sched(\ldots,\sigma_i,\ldots)$
$= SELECT(IMM(\ldots,\sigma_i,\ldots))$;
(7) $\delta_{int}(s) = (\ldots,(si',ei'),\ldots,i^{*'})$ where
$(s_i',e_i') = (\delta_{int,j}(s_j),0)$, for $j = i^*$,
$(s_i',e_i') = (\delta_{ext,j}(s_j,e_j+ta(s),Z_{i^*,j}(\lambda_{i^*}(s_{i^*})),0)$,
for $j \in I_{i^*}$,
$(s_i',e_i') = (s_j,,e_j+ta(s))$, otherwise,
$i^{*'} = sched(\ldots,\sigma_i,\ldots)$
$= SELECT(IMM(\ldots,\sigma_i,\ldots))$;

The scheduling related information is stored and managed by a simulation engine. However, since the composition produces the resultant model as an atomic model, the information for scheduling came to be embedded in the composed model. The state variables σ_i's and i^* were appended for this need. The variable σ_i denotes the time left until the next scheduled time for each component $i \in D$ and i^* indicates the next scheduled component which is responsible for next event occurrence. Because all the state variables can be changed in a state transition function, σ_i's and i^* are modified in either an internal or external state transition function of the composed model. The next scheduled component i^* is determined by the function *sched* as specified in Definition 1(7).

One important fact about the composition is that the event translation process is interpreted prior to runtime and its result is employed to construct the state transition function of the composed model. Thus, the composed model deploys no message passing in runtime. Another important fact is that the scheduling task is resolved in a state transition function only by referring the local state variables (σ_i's), thus its computation time can be reduced significantly. In contrast, the scheduling task of the original model requires certain amounts of processing time, because the task deploys frequent message passing among the components in a distributed manner to share the scheduling information of the components. In short, the composition is an operation of constructing a compiled model out of the component models by abstracting the event translation task away and embedding an internal scheduling mechanism which can be computed in a shorter time.

Definition 2 (Behavior): The behavior of a DEVS model, ω is a record of the timed state transitions of the form $\omega = (s_0,t_0),(s_1,t_1),\ldots(s_i,t_i),\ldots$ where s_i is the changed state at time t_i and s_0 is the initial state.

Theorem 1: The behavior of a composed model is the same as that of the original model. That is composition does not change the behavior of a model.

The proof is shown in Appendix B.

5.2 Composition Example

The component models of the coupled model *PEL*, *Buff* and *Proc* can be composed into a new atomic model, PEL^{comp} as follows.

$PEL^{comp} =< X,\ Y,\ S,\ \delta_{ext}, \delta_{int}, \lambda,\ ta >$
$X = \{in\}$
$Y = \{done\}$
$S = \{(n, pstat, \sigma_{Buff}), (stat, \sigma_{Proc}), i^*\} |$
$i^* \in \{Buff, Proc\}\}$
Let $s = (q_{Buff}, q_{Proc}, i^*), s' = (q'_{Buff}, q_{Proc}{}', i^{*'})$,
$q_{Buff} = (n, pstat, \sigma_{Buff}), q_{Proc} = (stat, \sigma_{Proc})$,
and $q_{Buff}{}' = (n', pstat', \sigma'_{Buff})$,
$q_{Proc}{}' = (stat', \sigma'_{Proc})$ for notational convenience.
$\delta_{ext}((q_{Buff}, q_{Proc}, i^*), e, in) = (q'_{Buff}, q'_{Proc}, i^{*'})$
where $q'_{Buff} = (n+1, pstat, ta_{Buff}(n, pstat))$,
$q_{Proc}{}' = (stat, \sigma_{Proc} - e)$, and
$i^{*'} = sched(ta_{Buff}(n, pstat), \sigma_{Proc} - e)$
$\delta_{int}(q_{Buff}, q_{Proc}, i^*) = (q'_{Buff}, q'_{Proc}, i^{*'})$
where $q_{Buff}{}' = (n-1, B, ta_{Buff}(n, pstat))$,
$q_{Proc}{}' = (B, ta_{Proc}(stat))$,
and $i^{*'} = sched(ta_{Buff}(n, pstat), ta_{Proc}(stat)$,
if $n > 0 \wedge pstat = F \wedge i^* = Buff$
$q_{Buff}{}' = (n, pstat, \sigma_{Buff} - min(\sigma_{Buff}, \sigma_{Proc}))$,
$q_{Proc}{}' = (F, \infty)$,
and $i^{*'} = Buff$, if $stat = B \wedge i^* = Proc$
$\lambda(q_{Buff}, (B, \sigma_{Proc}), Proc) = done$
$ta((n, pstat, \sigma_{Buff}), q_{Proc}, Buff) = 0$,
if $n > 0 \wedge pstat = F$
$ta(q_{Buff}, (B, \sigma_{Proc}), Proc) = service\_time$

The input and the output events set, X and Y, are the same as those of the coupled model *PEL*. The states set of the composed model is determined as a product set of the total state of each component. The element i^* in the composed state denotes the next scheduled component which will activate the next event.

The input event *in* only affects *Buff*; thus the state variable n, which was originated from the component *Buff,* is incremented and its next scheduled time is updated, while the state variable *stat* from the component *Proc* is not changed and its left time to the next scheduled time is decremented. Considering the case where the output event *out* is generated from *Buff* and is delivered to its influencee *Proc*, the event translation task is performed as described in Sect. 4. In this case, two characteristic functions of *Buff* are executed in the model before composition. The output function (λ_{Buff}) is invoked producing the output event *out* and an internal state transition function ($\delta_{int,Buff}$) is executed modifying its two state variables n and *pstat*. At the same time the simulator translates this event as an input event *req* of *Proc*. Thus *Proc* executes its external state transition function ($\delta_{ext,Proc}$) modifying the state variable *stat*. In contrast, all these operations becomes abstracted in the composed model. In the composed model the state changes are made in a single state transition function δ_{int}, and the event translation task does not take place. This is the reason why the composition can accelerate the simulation speed. The case where *Proc* generates an output event *done* is similar to the previous case.

Algorithm 1 Simulation algorithm for composed model.

```
while (t_N < ∞) do
  t := t_N
  s := δ_int(s)
  t_N := t + ta(s);
end while
```

In addition, the scheduling job, which resolves i^* for the next execution, is performed in a state transition functions which may be either of δ_{ext} or δ_{int}. The function *sched* is called to determine the next i^*. Because the function *sched* only refers local state variables such as σ_{Buff} and σ_{Proc}, its execution time is expected to be much shorter than that of the original model. In fact, the original model deploys many message passings among the components to inform their next schedule times.

5.3 Simulation Algorithm for the Composed Model

Besides the fact that the simulation speed of a composed model can be increased significantly, there are additional two more points of improvements. During the process of simulation, the next event is executed if the processor of a model has received a message from the run-controller of a simulation kernel [6]. But this message passing is an expensive operation. Therefore, if both the function of the run-controller and the processor are integrated, the simulation can be proceeded more faster as there happens no message passing. This is one point of improvement. The other improvement is the fact that there is no need to call the output function of the composed model. In fact, the output function need not be executed in the composed model, because the model has no interaction with its environment. The Algorithm 1 achieves these two points of improvements.

Theorem 2: When a finally composed model is executed by the Algorithm 1, the same behavior is produced as is obtained by the abstract simulator algorithm [6].

The proof is shown in Appendix C.

6. Experimental Results

The simulation speedup has been evaluated by the use of a discrete event simulation environment, DEVSim++ [10]. Three example models were tested on a

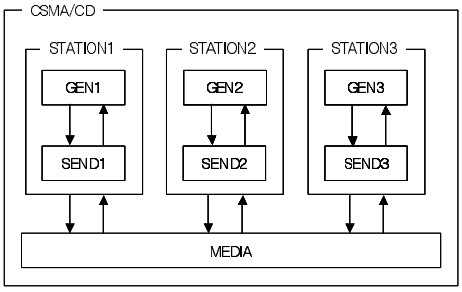

Fig. 5 Overall structure of CSMA/CD.

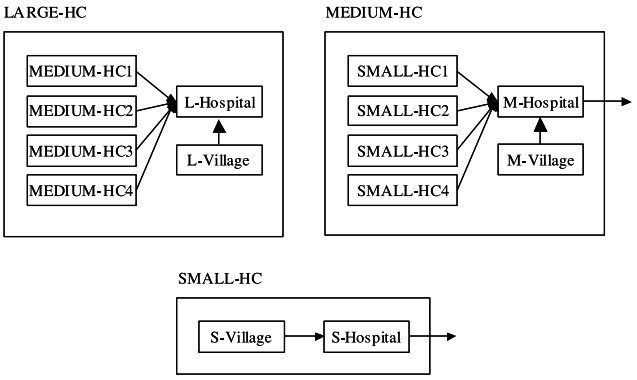

Fig. 6 The model of Columbian Health Care System.

Pentium-II PC. The GNU profile tool, *gprof* was used to measure the simulation time.

The first example model is *Single Server Queue,* which consists of four atomic models. Two of them are *Buff* and *Proc* which have already been specified in Sect. 3.2. The other component models are *GENR* and *TRANSD*. The model *GENR* generates problems and sends them to *Buff* and *TRANSD* collects outputs from *Proc*.

The second model, *CSMA/CD* consists of three coupled models *STATION* and an atomic model *MEDIA*, as shown in Fig. 5. *STATION* refers to a network node, which is connected to the physical network media and has two atomic models *GEN* and *SEND*. *GENR* merely generates data and *SEND* sends data to *MEDIA* if the transmission line is available. *SEND* goes to a jamming state when it receives a collision message and tries to resend the collided data after back-off time. *MEDIA* broadcasts the collision message to the *STATION* models when more than one *STATION* tries to send data simultaneously (IEEE std 802.3).

The final model, Columbian Health Care System (*CHCS*) has been exeprimented. The model has been used in various studies [17]. The model is one of a multi-tiered health care system of villages and health centers. There is one health center for each village. When villagers become ill, they travel to their local health center for assessment and are treated when possible. If they cannot be treated locally, patients are referred up the hierarchy to a next health center, where the assessment/treatment/referral process is repeated. Upon arriving at a health center, a patient is enqueued until a health care worker becomes available. It is assumed that patients can always be treated at the top-level hospital of the health care system. Figure 6 depicts the hierarchically constructed *CHCS* model.

Table 3 shows the effect of the composition for the *SSQ* model. The first column of this table shows the set of the components which are composed into a model. As the many components are merged, the number of messages are reduced and a lesser simulation time is required. In particular, when the simulation algorithm for the composed model was applied, the message was not generated at all, as shown at the last row in Table 3. Figure 7 shows the relation of the total number of messages and the simulation time. They are almost linearly proportional to each other. Thus we can perceive that the overhead of updating state variables is negligibly small and the activity of message passing is closely related with the other overheads.

Because the composition does not reveal state explosion problem, a large size DEVS model can be effectively composed and will reveal increased simulation speed. Furthermore, the composition of all the components in the *SSQ* takes just a few milliseconds. The composed models can be simulated fast, but they have disadvantages also in that they show bad readability and are difficult to maintain at a later time. Therefore, we can use the composition very effectively when the model is of release-version.

Table 4 shows the experimental results when both the composition and the proposed simulation algorithm were applied for the three example models. To observe how much simulation speed is accelerated as the proposed methods are applied, we measured three cases of simulation times for each model. The first is the simulation time of the original model, the second is that of the model after composition, and the third was measured after both the composition and the proposed simulation algorithm was applied. As shown in Table 4, the composition improved the execution speed approximately 6 $((7.2 + 5.4 + 5.4)/3)$ times faster than that of the original model. Furthermore, when the proposed simulation algorithm was applied, the speedup was as much as 2.3 times faster. Thus, the total simulation speedup was approximately 14 times faster on average. Most of all, the event translation overhead was completely eliminated at the third experiments and 91% $(1 - (1.2/17.8 + 1.7/18.6 + 2.2/21.6)/3)$ of the scheduling overhead was reduced after composition. Finally, we can observe that the two dominant simulation overheads, that is the event translation overhead and the scheduling overhead, can be significantly reduced as the methods are applied.

The overhead of updating state variables was slightly reduced after composition. Originally, state

Table 3 Relation between the message number and the simulation time.

Composed model set	Algo. 1 applied ?	Message number					Simulation time (sec.)
		(x)	(y)	(*)	(done)	Total	
{}	no	1600009	1200007	2400014	4000023	9200053	28.3
{Buff, Proc.}	no	800005	800005	2400014	3200019	7200043	20.7
{Gen, Buff, Proc.}	no	400003	400003	2400014	2800017	6000037	15.9
{Gen, Buff, Proc., Trans.}	no	0	0	1200007	1200007	2400014	4.0
{Gen, Buff, Proc., Trans.}	yes	0	0	0	0	0	1.5

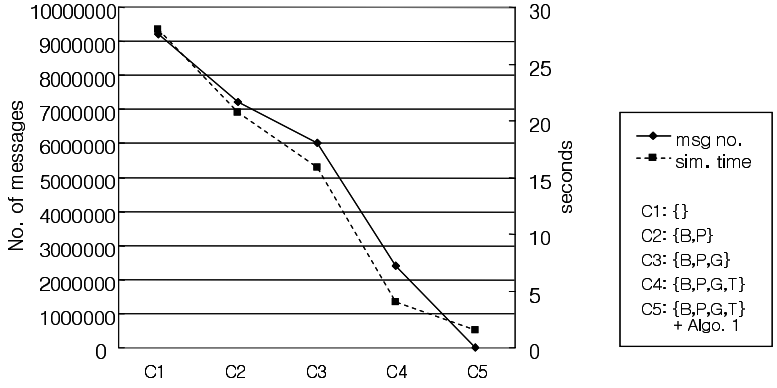

Fig. 7 Total message number vs. simulation time for several configurations.

Table 4 Simulation speedup.

Experiments		Time taken for each overhead (sec.)			Total time (sec.)	Cumulative speedup
		State var. update	Event translation	Scheduling		
SSQ	Original model	1.6	8.9	17.8	28.3	1.0
	Composition	0.5	0.2	3.2	3.9	7.2
	Composition + Algo.1	0.3	0.0	1.2	1.5	18.5
CSMA/CD	Original model	2.1	8.5	18.6	29.2	1.0
	Composition	0.8	0.3	4.3	5.4	5.4
	Composition + Algo.1	0.7	0.0	1.7	2.5	11.9
CHCS	Original model	2.2	8.9	21.6	32.7	1.0
	Composition	1.1	0.3	4.8	6.1	5.4
	Composition + Algo.1	0.8	0.0	2.2	3.0	11.0

Table 5 Simulation speedup with respect to the number of stations in the CSMA/CD model.

Experiments		Time taken for each overhead (sec.)			Total time (sec.)	speedup
		State var. update	Event translation	Scheduling		
CSMA/CD with 2 stations	Original model	1.1	5.6	12.1	18.8	1.0
	Composition + Algo.1	0.4	0.0	1.1	1.5	12.6
CSMA/CD with 3 stations	Original model	2.1	8.5	18.6	29.2	1.0
	Composition + Algo.1	0.7	0.0	1.7	2.5	11.9
CSMA/CD with 4 stations	Original model	3.3	11.8	25.7	40.7	1.0
	Composition + Algo.1	1.3	0.0	3.0	4.3	9.5
CSMA/CD with 5 stations	Original model	3.4	10.9	26.1	40.4	1.0
	Composition + Algo.1	1.3	0.0	3.1	4.4	9.1

transition functions of components are called one by one as the model is interpreted at run-time. But, for the composed model the state transition functions are pre-merged at composition-time (compile-time) and are called all at a time as shown in Fig. 4. Therefore, the number of function calls made to update state variables becomes small, as the model is composed. Consequently, the time taken for updating state variables is reduced with the composition.

Table 5 shows the speedup variation with respect to the varying number of stations in the *CSMA/CD* model. As shown in Table 5, the number of stations has little relation with the simulation speedup. For the above four models, the event translation and the scheduling overhead hold about 29% ((5.6/18.8 + 8.5/29.2 + 11.8/40.7 + 10.9/40.4)/4) and 64% ((12.1/18.8 + 18.6/29.2 + 25.7/40.7 + 26.1/40.4)/4) of overall simulation run-time. Composition removes

the event translation overhead completely and unburdens the scheduling overhead by a factor of 90% $(1 - (1.1/12.1 + 1.7/18.6 + 3.0/25.7 + 3.1/26.1)/4)$ in average. And as this factor is almost independent with the component numbers, it is expected that the composition would still be effective even for larger models.

7. Conclusion

This paper introduces a composition method of DEVS models. The method is a compiled simulation technique that progresses significant speedup for discrete event simulation. To analyze the most critical overhead of the abstract simulator algorithm for the DEVS models, we first classified the overall simulation process into three overheads and measured the time taken for each overhead. By experimenting three example models, we identified that the activity of message passing is closely related to heavy simulation overheads.

To reduce those dominant overheads, we proposed two methods. One is composition, which combines the component models into a single model. This operation eliminates the run-time activity of message passing. The other is to apply an efficient simulation algorithm for the composed model. When the two methods were applied, the simulation speed was approximately 14 times faster and the activity of message passing did not take place. As the method is defined in a formal way it is promising to implement an automated tool which can always benefit the simulation of multi-component DEVSs to be executed faster. And, this is a great difference compared with the previous works on compiled simulation technique

References

[1] G.J. Holzmann, "The model checker SPIN," IEEE Trans. Softw. Eng., vol.23, no.5, pp.279–295, May 1997.
[2] K. McMillan, SMV—Symbolic Model Checking, Kluwer Academic Publishers, 1993.
[3] J. Bengtsson, K.G. Larsen, F. Larsson, P. Pettersson, and W. Yi, "UPPAAL—a tool suite for the automatic verification of real-time systems," Proc. Hybrid Systems III. LNCS, 1066, pp.232–243, Spriger-Verlag, 1996.
[4] D. Harel, "Statechart: A visual formalism for complex systems," Science of Computer Programming, vol.8, pp.231–274, 1987.
[5] M.A. Holliday and M.Y. Vernon, "A generalized timed Petri net model for performance analysis," IEEE Trans. Softw. Eng., vol.13, no.12, pp.1297–1310, Dec. 1987.
[6] B.P. Zeigler, Multifacetted Modelling and Discrete Event Simulation, Academic Press, 1984.
[7] D.M. Lewis, "A hierarchical compiled code even-driven logic simulator," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.10, no.6, pp.726–737, 1991.
[8] A.M. Law and C.S. Larmey, An Introduction to Simulation Using SIMSCRIPT II.5, CACI, Sept. 1984.
[9] G.P. Hong and T.G. Kim, "A framework for verifying discrete event models within a DEVS-based system development methodology," Trans. Soc. Comput. Simul. (USA), vol.13, no.1, pp.19–34, 1996.
[10] T.G. Kim and S.B. Park, "The DEVS formalism: Hierarchical modular systems specification in C++," Proc. 1992 European Simulation Multiconference, pp.152–156, 1992.
[11] J.S. Hong, H.S. Song, T.G. Kim, and K.H. Park, "A real-time discrete event system specification formalism for seamless real-time software development," Discrete Event Dynamic Systems, vol.7, pp.355–375, 1997.
[12] S.M. Cho and T.G. Kim, "Real-time simulation framework for RT–DEVS models," Trans. Soc. Comput. Simul. (USA), vol.18, no.4, Dec. 2001.
[13] A.I. Concepcion and B.F. Zeigler, "DEVS formalism: A framework for hierarchical model development," IEEE Trans. Softw. Eng., vol.14, no.2, Feb. 1988.
[14] Y.G. Kim and T.G. Kim, "Optimization of model execution time in the DEVSim++ environment," Proc. 1997 European Simulation Symposium, pp.215–219, Passau, Germany, Oct. 1997.
[15] R.M. Fujimoto, "Optimistic approaches to parallel discrete event simulation," Trans. Soc. Comput. Simul. (USA), vol.7, no.2, pp.153–191, 1990.
[16] M.S. Ahn and T.G. Kim, "A framework for hybrid modeling/simulation of discrete event systems," AIS'94, pp.199–205, Gainesville, FL, Dec. 1994.
[17] D. Baezner, G. Lomow, and B.W. Unger, "Sim++: The transition to distributed simulation," Proc. SCS Multiconference on Distributed Simulation, 1990.

Appendix A: Abstract Simulator Algorithm

Zeigler proposed the abstract simulator concept [6] for simulation of DEVS models. The simulator associates each model to a virtual processor that interprets the dynamics specified by the formalism, in a one-to-one manner. A simulation proceeds by means of message passing among the processors, not among DEVS models. There are two types of processors: a *simulator* for an atomic model and a *coordinator* for a coupled model. A special kind of coordinator called a *root coordinator* is associated with no models and takes responsibility for advancing simulation time. The simulators and coordinators are linked by the coupling information of the corresponding coupled models, thus forming the same hierarchical structure as that of the models. Each processor simulates a system by sending and/or receiving four types of messages: (x), (y), (*), and (done) messages.

The job of an abstract simulator for an atomic model is to determine the next scheduling time, t_N and to request the associated atomic model to execute its transition functions and output function in a timely manner. A simulator receives and processes (x) messages and (*) messages. As a result, it produces (y) messages and (done) messages. Because a simulator associates to an atomic model, which is always a leaf node of a system decomposition tree, it cannot receive (y) and (done) messages. The event handling algorithms for the abstract simulator are presented in Algorithm 2.

The responsibility of a coordinator for a coupled model is to synchronize the component abstract sim-

Algorithm 2 Message handling algorithm for the abstract simulator.

```
Simulator:when_rcv_(x,t)
  if t_L ≤ t ≤ t_N then
    e := t − t_L;
    s := δ_ext(s,e,x);
    t_L := t;
    t_N := t_L + ta(s);
    send (done,t_N) to the parent coordinator;
  end if
Simulator:when_rcv_(*,t)
  if t= t_N then
    y := λ(s);
    send (y,t) to the parent coordinator;
    s := δ_int(s);
    t_L := t;
    t_N := t_L + ta(s);
    send (done,t_N) to the parent coordinator;
  end if
```

ulators for scheduling the next event time and routing external event messages to component simulators. Scheduling and event routing are performed in a hierarchical manner. A coordinator receives and processes (x), (*), and (done) messages. As a result, it produces (y), and (done) messages. The event handling algorithms for the coordinator are presented in Algorithm 3.

Algorithm 3 Message handling algorithms for the coordinator.

```
Coordinator:when_rcv_(x,t)
  if t_L ≤ t ≤ t_N then
    for all affected processor p^I do
      children := children ∪ { p^i };
      send (x,t) to p^i;
    end for
    wait until children is empty;
    t_L := t;
    t_N := min { t_N of all child processors };
    send (done,t_N) to the parent coordinator;
  end if
Coordinator:when_rcv_(*,t)
  if t = t_N then
    p*:= select(imminent processors)
    children := children ∪ { p*}
    send (*,t) to p*;
    t_L := t;
    t_N := min { t_N of all child processors };
    send (done,t_N) to the parent coordinator;
  end if
Coordinator:when_rcv_(y,t)
  for each affected child processor p^i; do
    children := children ∪ { p^i }
    send (x,t) to p^i;
  end for
  if the other processor p^j is affected; then
    send (y,t) to the parent coordinator;
  end if
Coordinator:when_rcv_(done,t)
  children := children \ { source processor of the
  (done,t) }
  if children is empty;  then
    t_L := t;
    t_N := min { t_N of all child processors };
    send (done,t_N) to the parent coordinator;
  end if
```

Appendix B: Proof of Theorem 1

Proof: Let a coupled model be $CM = < X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, SELECT >$ and each of the components be an atomic model as $M_i =< X_i, Y_i, S_i, \delta_{exi,i}\delta_{ini,i}\lambda_i, ta_i >, \forall\ i \in D$. And let its composed model M be as $M =< X, Y, S, \delta_{ext,}\delta_{int,}\lambda, ta >$

By the abstract simulator algorithm of Algorithm 2 and Algorithm 3, simulation proceeds when a processor accepts a (*) or a (x) message. Thus, it is sufficient to prove that both the composed and the original model reveal the same responses whenever they receive one of these messages, that is they update the state variables with the same values at the same time and reports the same messages to the parent processor.

- **Case 1)** Let's assume that the same (*) has arrived at both models.
  First, CM selects p^* at line 2 of Coordinator::when_rcv_(*,t) in Algorithm 3. This p^* is exactly the same as the i^* in Definition 1, since the *sched* function in Definition 1 applies the same procedures to determine i^*. Once the p^* is determined p^*'s when_rcv_(*,t) is invoked by the message generated at the line 4 of Coorinator:when_rcv_(*,t), then δ_{exi,i^*} changes s_{i^*} at the line 4 of Simulator:when_rcv_(*,t) and (y) message is generated from the line 3 of Simulator:when_rcv_(*,t). The generated (y) message invokes Coordinator:when_rcv_(y,t) in Algorithm 3. The invoked routine does the following two tasks: 1) The routine updates all the state variables of every influenced components as specified at the line 3 of Coordinator:when_rcv_(y,t) and at the Simulator:when_rcv_(x,t). 2) It routes the (y) message to the parent processor if it also influences the component outside of the coupled model, CM. In summary, the component which recieved the (*) message executes its δ_{int}, and all the influenced children executes their δ_{ext}'s. And if the output event generated from the output function of i^* influences other components outside the CM, a new (y) message will be passed to the parent coordinator. This sequential task completely conforms with the actions specified in Definition 1(7) and Definition 1(5). Thus, the changed state values and the generated (y) messages of CM are exactly the same with those of the composed model, M. After all these event translation tasks finish, CM reports the next scheduled time at line 6,7 of Coordinator::when_rcv(*,t) in Algorithm 3. This value is also the same as that of M, specified in the Def-

inition 1(4), which is determined at the line 6 of Simulator:when_rcv_(*,t) in Algorithm 2.

- **Case 2)** Let's assume that the same (x,t) has arrived at CM and M.
  In a similar way as that of case 1), all the corresponding state variables of CM and M are changed with the same values and the next scheduled times being the same.

□

Appendix C: Proof of Theorem 2

Proof: Let the timed state transition $w_1 = (s_{1,0}, t_{1,0})\ldots(s_{1,i}, t_{1,i})\ \ldots$ be the behavior generated by the abstract simulator algorithm and $w_2 = (s_{2,0}, t_{2,0})\ldots(s_{2,i}, t_{2,i})\ \ldots$ be the behavior generated by the Algorithm 1. The theorem can then be proved by an induction.

- *The first step:* $(s_{1,0}, t_{1,0}) = (s_{2,0}, t_{2,0})$.
  Since the same model is simulated, the initial states $s_{1,0}$ and $s_{2,0}$ are the same. And their initial times $t_{1,0}$ and $t_{2,0}$ are all zero.
- *Induction step:* Let $(s_{1,i}, t_{1,i})$ and $(s_{2,i}, t_{2,i})$ be the same. Then it is sufficient to show that $(s_{1,i+1}, t_{i+1,i})$ and $(s_{2,i+1}, t_{2,i+1})$ are the same.
  By the procedure Simulator:when_rcv_(*,t) in the abstract simulator algorithm, $s_{1,i+1}$ is determined as $\delta_{int}(s_{1,i})$. and $s_{2,i+1}$ is determined as $\delta_{int}(s_{2,i})$ by the Algorithm 1. As $s_{1,i}$ and $s_{2,i}$ are the same, so are $s_{1,i+1}$and $s_{2,i+1}$. The next scheduled time is advanced by the amount of $ta(s_{1,i+1})$ for abstract simulator algorithm, that is $ta(s_{2,i+1})$ for the case of the Algorithm 1. Since $s_{1,i+1}$and $s_{2,i+1}$are the same, so are $ta(s_{1,i+1})$ and $ta(s_{2,i+1})$. Thus the next schedule times are the same.

□

Wan Bok Lee received the B.E and M.E degrees in electrical engineering from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea, in 1993 and 1995, respectively. He is presently a Ph.D. candidate at the Department of Electrical Engineering and Computer Science, KAIST, and a Full-time Instructor at Information Security Department of Joongbu University, Chungnam, Korea since 2003. His research interests include methodology for modeling and simulation of discrete event systems, formal method, model checking, and information security.

Tag Gon Kim received his Ph.D. in computer engineering with specialization in systems modeling/simulation from University of Arizona, Tucson, AZ, 1988. He was a Full-time Instructor at Communication Engineering Department of Bookyung National University, Pusan, Korea between 1980 and 1983, and an Assistant Professor at Electrical and Computer Engineering at University of Kansas, Lawrence, Kansas, U.S.A. from 1989 to 1991. He joined at Electrical Engineering Department of KAIST, Tajeon, Korea in Fall, 1991 as an Assistant Professor and has been a Full Professor at EECS Department since Fall, 1998. His research interests include methodological aspects of systems modeling simulation, analysis of computer/communication networks, and development of simulation environments. He has published more than 100 papers on systems modeling, simulation and analysis in international journals/conference proceedings. He is a co-author (with B.P. Zeigler and H. Praehofer) of Theory of Modeling and Simulation (2nd ed.), Academic Press, 2000. He was the Editor-in-Chief of SIMULATION:Trans of SCS published by Society for Computer Simulation International (SCS). He is a senior member of IEEE and SCS and a member of ACM and Eta Kappa Nu. Dr. Kim is a Certified Modeling and Simulation Professional by US National Training Systems Association.