Simulation Speedup for DEVS Models by Composition-based Compilation

Wan Bok Lee and Tag Gon Kim

Department of Electrical Engineering & Computer Science
Korea Advanced Institute of Science and Technology (KAIST)
373-1 Kusong-dong, Yusong-ku, Daejeon 305-701, KOREA
E-mail: wblee@smslab.kaist.ac.kr, tkim@ee.kaist.ac.kr

Keywords: simulation speedup, DEVS formalism, performance evaluation.

ABSTRACT

The methods or algorithms which can accelerate the simulation speed are of great importance, since the modeling and simulation methodology for discrete event systems are used in many areas such as model validation/verification and performance evaluation. This paper proposes a method for simulation speedup for 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. Experimental results show that the simulation speed of transformed DEVS models is about 15 times faster than original ones.

1 INTRODUCTION

Various simulation models have been developed reflecting diverse simulation world views such as event scheduling, activity scanning and process interaction. Each model has its own unique strengths and weaknesses. For example, the models that are based on event scheduling are hard to understand and difficult to maintain them, but they can be executed at a high speed. Meanwhile, those models that are based on process interaction demonstrate a well-structured architecture and good quality of readability, but they require more execution time. Usually, both the simulation speed and well-structured model architecture seem to be incompatible.

The Discrete Event Systems Specification (DEVS) formalism, introduced by Zeigler[1], specifies discrete event models in a hierarchical, modular form. This hierarchical modeling offers such advantages as fast model development, model reuse, and easy model verification and validation[2]. These advantages are very helpful in modeling and simulation of large and complex systems. On the contrary, the simulation of a hierarchical DEVS model involves frequent message transmission to share the information of its component models. Thus, the simulation speed of DEVS models is relatively slow compared to that of the models based on event scheduling world view. 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[3]. However, this method requires additional cost in hardware as well as extra work for parallelization of existing sequential models. Hybrid simulation[4] 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[4]. Transformation of DEVS model structure for simulation speedup has been attempted in [5]. 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 eliminating the execution time overhead originated from repeated interpretation of complex data structure of a model in run-time[6]. 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.

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[1][7]. 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 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 trans-formation 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. The model of computation, DEVS formalism is briefly introduced in Section 2. In Section 3, run-time simulation overheads are classified and analyzed. The proposed mothed for simulation speedup is described in Section 4. After showing the experimental results in Section 5, conclusion follows in Section 6.

2 MODEL OF COMPUTATION

2.1 DEVS Formalism: A Brief Review

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 [7]:

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

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 $Q = \{(s,e) \mid s \in S \text{ and } 0 \le e \le ta(s)\}$, total state set of M;

$\delta_{int.} : S \rightarrow S$, an external transition function;

$\lambda : S \rightarrow Y$, 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 [7]:

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

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;

for each $i, j \in D \cup \{self\}$

$Z_{i,j} : Y_i \rightarrow X_j$, an i-to-j output event translation functions;

$SELECT : 2^D - \varnothing \rightarrow D$, tie-breaking function.

2.2 An Example Model

Fig. 1 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 $= < X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta >$

$X = \{in\}$;

$Y = \{out\}$;

$S = \{(n,pstat) \mid n \in I, pstat \in \{B,F\}\}$;

$\delta_{ext}((n,pstat),e,in) = (n+1,pstat)$;

$\delta_{ext}((n,B),e,ready) = (n,F)$;

$\delta_{int}((n,F)) = (n-1,B)$, if $n > 0$;

$\lambda((n,F)) = out$, if $n > 0$;

$ta(n,pstat) = 0$, if $n > 0$ and $pstat = F$;

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

$X = \{req\}$;

$Y = \{done\}$;

$S = \{(stat) \mid stat \in \{B,F\}\}$;

$\delta_{ext}(F,req) = B$;

$\delta_{int}(B) = F$;

$\lambda(B) = done$;

$ta(B) = service\_time$;

Once *Buff* and *Proc* are developed, the coupled model *PEL* can be specified by defining components and coupling

relations as defined in the coupled model. Thus, the *PEL* is formalized 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*.

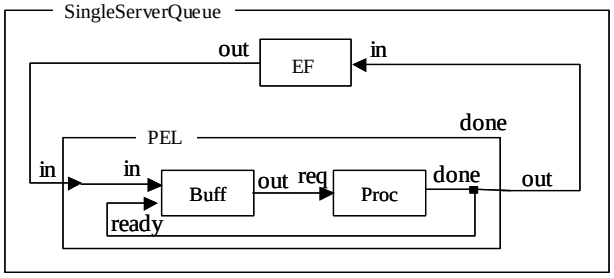

Fig. 1 Example DEVS model, *Single Server Queue*

2.3 DEVSIF Representation

DEVSIF is a modeling language developed at KAIST to represent models of discrete event systems in a formal way. The formal expression makes it easy to analyze, simulate and execute the model[8].

DEVSIF has three parts to describe the overall model, which are *interface*, *atomic model*, and *coupled model*. interface part specifies a set of input/output events which is common to a atomic model and a coupled model. A DEVSIF model preserves model information in the DEVS formalism and supports object-oriented feature. In DEVSIF, atomic or coupled models are described in an extended BNF format as:

interface model_name
    inputs : {…}
    outputs : {…}
end model_name;
atomic model model_name
    state variables : [var_name : type_def;]*
    initial condition : [expr]*;
    internal transition : [(expr) => {expr};]*
    external transition :
        [(expr) * input_event => {[expr;]+};]*
    output function : [(expr) => expr;]*
    time advance : [(expr) => expr;]*
end model_name;

interface model_name
    inputs : {…}
    outputs : {…}
end model_name;
coupled model model_name
    component : {[child_name : model_name]+}
    external input coupling :
        {[model_name.input_event->
        child_name.input_event;]*}
    external output coupling :
        {[model_name.output_event->
        child_name. output_event;]*}
    internal coupling :
        {[src_child_name.output_event->
        dest_child_name.output_event;]*}
    [**select** : [{[child_name;]*}]]
end model_name;

3 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 simulators each of which is associated with a component of an overall hierarchical DEVS model in an one-to-one manner. Thus, simulation of DEVS models is performed such a way that event scheduling and message passing between such component models are done in a hierarchical manner.

Basically the abstract simulators algorithm[1] for DEVS models 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 internal 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 external state transition functions. To find such components a coupling scheme specified in coupled models are 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 the (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 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.

Table 1 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%)

To measure the time taken for each overhead in a simulation run, we experimented with two example models, as shown in Table 1, with an object-oriented simulation environment of DEVSim++[9]. 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.

4 COMPOSITION METHOD FOR SIMULATION SPEEDUP

4.1 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, called *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.

DEFINITION 1: Let a coupled model $CM = < X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, SELECT >$ and each component be an atomic model as:

$M_i = < X_i, Y_i, S_i, \delta_{exi,i}\ \delta_{ini,i}\ \lambda_i, ta_i >, \forall\ i \in D.$

Then a composed model M is defined as follows:

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

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;

(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 $IMM(..., \sigma_i, ...) = \{i \in D | \sigma_i = \sigma_{min}\}$

(6) $\delta_{ext}(s,e,x) = (..., (s_i',e_i'), ..., i^{*'})$ 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

$i^{*'} = sched(..., \sigma_i, ...) = SELECT(IMM(..., \sigma_i, ...))$

(7) $\delta_{int}(s) = (..., (si',ei'), ..., i^{*'})$ 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

$i^{*'} = sched(..., \sigma_i, ...) = SELECT(IMM(..., \sigma_i, ...))$

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 an external state transition function of the composed model. The next scheduled component i^* is determined by the function *sched* as specified in DEFINITION 1.

One important fact about the composition is that the event translation process is interpreted prior to run-time and its result is employed to construct the state transition function of the composed model. Thus, the composed model deploys no message passing in run-time. 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.

The behavior of a DEVS model, can be understood as 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. 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.

4.1 Composition Example

The component models of the coupled model *PEL* as in Fig.1, *Buff* and *Proc* can be composed into a new atomic model, PEL^{comp} like following.

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

$X = \{in\}$;

$Y = \{done\}$;

$S = \{(n,pstat,\sigma_{Buff}),(stat,\sigma_{Proc}),i^* \mid i^* \in \{Buff,Proc\}\}$;

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 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 Section 3. 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.

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 scheduled times.

5 EXPERIMENTAL RESULTS

The simulation speedup has been evaluated by the use of a discrete event simulation environment, DEVSim++[9]. Two example models were tested on a 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 Section 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*.

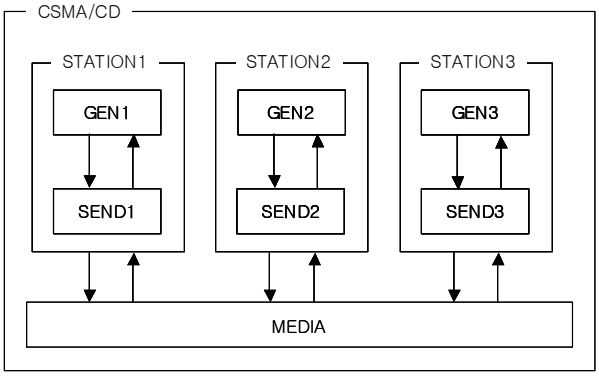

Fig. 2 Overall structure of CSMA/CD

The second model, *CSMA/CD* consists of three coupled models *STATION* and an atomic model *MEDIA*. *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[10].

Table 2 Simulation speedup

Experiment Model.		Time taken for each overhead (sec.)			Total time (sec.)	Speedup
		State var. update	Event translation	Scheduling		
SSQ	Original model	1.6	8.9	17.8	28.3	1.0
	Composed model	0.3	0.0	1.2	1.5	18.5
CSMA/CD	Original model	2.1	8.5	18.6	29.2	1.0
	Composed model	0.7	0.0	1.7	2.5	11.9

Table 2 shows the experimental results when the composition was applied for the two example models. To observe how much the simulation speed is accelerated as the proposed method is applied, we measured two simulation times for each model. The first is the simulation time of the original model, the second is that of the model after composition. As shown in Table 2, the composition improved the execution speed approximately 15.2 ((18.5 + 11.9) / 2) times faster than that of the original model. Most of all, the event translation overhead was completely eliminated at the third experiments and 92% (1 - (1.2/17.8 + 1.7/18.6) / 2) 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.

6 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 two example models, we identified that the activity of message passing is closely related to some of the heavy simulation overheads.

To reduce those dominant overheads, we proposed a compilation method named composition. The composition combines the component models into a single model. This operation eliminates the run-time activity of message passing. When the composition was applied, the simulation speed was faster 15 times on average 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

ACKNOWLEDGEMENTS

The authors are grateful to K. J. Hong for his fruitful discussion and support. Hong gave much help to extend the syntax of DEVSIF modeling language such that the composed model became to be represented as another DEVSIF of a compact form.

REFERENCES

[1] B. P. Zeigler, *Multifacetted Modelling and Discrete Event Simulation*. Academic Press. 1984.

[2] R. Sargent, “Hierarchical modeling for discrete event simulation (panel)” *In Proceedings of the 1993 Winter Simulation Conference, p569.*

[3] R. M. Fujimoto. ”Optimistic Approaches to Parallel Discrete Event Simulation” *Trans. of The Society for Computer Simulation,* vol. 7, no. 2, pp.153-191, 1990.

[4] Myung S. Ahn and Tag G. Kim, “A Framework for Hybrid Modeling/Simulation of Discrete Event Systems,” *AIS'94*, pp. 199-205, Dec. 1994, Gainesville, FL.

[5] Y. G. Kim and T. G. Kim, “Optimization of Model Execution Time in the DEVSim++ Environment”. *In Proc. of 1997 European Simulation Symposium*, pp.215-219, Oct. 1997, Passau, Germany.

[6] D. M. Lewis, “A hierarchical compiled code even-driven logic simulator,” *IEEE Transactions on Computer-Aided Design*, vol. 10, no. 6, pp.726-737, 1991.

[7] A. I. Concepcion, and B. F. Zeigler, “DEVS Formalism: A Framework for Hierarchical Model Development”, *IEEE Trans. on Software Engineering*, vol. 14, no. 2, Feb. 1988.

[8] Ki Jung Hong and Tag Gon Kim, “DEVSIF : Relational Algebraic DEVS Intermediate Format,” *Proceedings of AIS'2000*, Tucson, Arizona, U.S.A., 2000.

[9] T. G. Kim, and S. B. Park, “The DEVS Formalism: Hierarchical Modular Systems Specification in C++” *In Proceedings of the 1992 European Simulation Multiconference,* pp.152-156, 1992.

[10] *Supplements to Carrier Sense Multiple Access with Collision Detection*, Institue of Electrical and Electronics Engineers Inc., 1988.