DEVSIF Composer: A Synthesis Tool for Fast Interpretation of Simulation Models

Wan-Bok Lee, *Member, KIMICS*

Abstract— The methods or algorithms which can accelerate simulation speed became of great importance, as the modeling and simulation methodology for discrete event systems is used in many areas such as model validation/verification and performance evaluation. This paper proposes a tool named, DEVSIF composer. The tool is made of an automated compiled simulation technology and it builds a new composed model which can be executed much fast by composing the component models together. Models are described by our new specification language DEVSIF, which is compatible with object-oriented language and supports representation of a hierarchical model structure. Experimental results demonstrates that DEVSIF composer enhances the simulation speed of a transformed DEVS model 5 times faster than that of the original ones in average.

Index Terms— Simulation speedup, DEVS, composition, performance evaluation

I. INTRODUCTION

Systems design is an iterative process which involves modeling, simulation and analysis of candidate systems. If sub-systems and algorithms within larger systems are to be developed a discrete event modeling and simulation method can be employed[1].

Once a discrete event model for such a sub-system or an algorithm is developed a logical/behavior analysis as well as a performance evaluation of the model is performed before implementation. Thus, the design process transits among three phases, namely logical analysis, performance evaluation and implementation each of which often employs a different model.

If transitioning between such different models is performed manually, it would be a major hurdle to a seamless design process. A unified modeling framework which provides a basis to specify models at different phases in common semantics would overcome such a hurdle.

This paper describes a tool named DEVSIF composer which was implemented to accelerate the simulation speedup in performance evaluation of systems using DEVS models. The method is viewed as a compiled simulation technique[2] that eliminates run-time interpretation of communication paths among component models[3]. 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.

First of all, to make a process as a tool we should formalize the task and the process in a mathematical form. Thus, a modeling language named DEVSIF need to be defined first. Whenever we specify a simulation model using the DEVSIF language, a newly composed model can be built. It is our main contribution that we implemented a composition tool, named DEVSIF composer, which can automatically compose several component models as a single atomic model just within a second. The composed model reveals no massage passing activities at all, and it can be executed several times faster.

The rest of the paper is organized as follows. In Section 2, our modeling specification, DEVSIF will be explained. Section 3 describes the simulation mechanism and the execution overheads in brief. The proposed tool for simulation speedup is described in Section 4. After evaluating the experimental results in Section 5, Section 6 reveals the concluding remark.

II. DEVSIF SPECIFICATION

The DEVS formalism, which is a set-theoretic 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) the way in which these models are connected together in a hierarchical fashion. A basic model, called an atomic model, has specifications for the dynamics of the model. Detailed description about DEVS formalism is found in [4].

DEVSIF is a modeling language which can represent models of discrete event systems in a formal way[5]. The formal expression makes it easy to analyze, simulate and execute the model [5]. It is an extension of DEVS spec language [6], which is devised for the behavioral analysis of the model with no timing information. A similar specification language, called openDEVS, was defined in [7] which has three characteristics: preservation of the DEVS models information, object-oriented modeling, and model type-check. The DEVSIF specification includes all these features.

Manuscript received February 17, 2008.

Wan-Bok Lee is with the Department of Game Design, Kongju National University, Kongju, Chungnam, 314-712, Korea (Tel: +82-41-850-6098, Fax: +82-41-850-6150, Email: wblee@kongju.ac.kr)

If models are developed in DEVSIF, modelers can have several advantages. The most advantageous is that a model developed in DEVSIF can be utilized in the code generation phase to generate various simulation models depending on the target simulation environment such as DEVSim++[8][9] or DEVSim-Java environment. It is another merit that the designed model can be automatically composed by the help of a tool, DEVSIF composer. Both the composed model and the original one reveal the same behavior while the simulation run-time of the composed one is less than the other.

A. DEVSIF System Development Framework

Fig. 1 shows our DEVSIF-based modeling and simulation flow. The first step is system modeling phase. A modeler can specify a system in a DEVSIF specification. Then, the developed model needs to be verified or validated. Simulation may be used in this step. But this step need not employ a long time of simulation rather than it is only sufficient to check whether the model is correctly built. Usually simulation takes much computation time and power. If simulation speed is a critical problem or a fast execution is desirable, then the model can be composed at the next step. Code generator in DEVSIF frame-work can generate directly the executable code from a pre-developed DESIF specification. Finally, we can perform a massive simulation and get various experimental results.

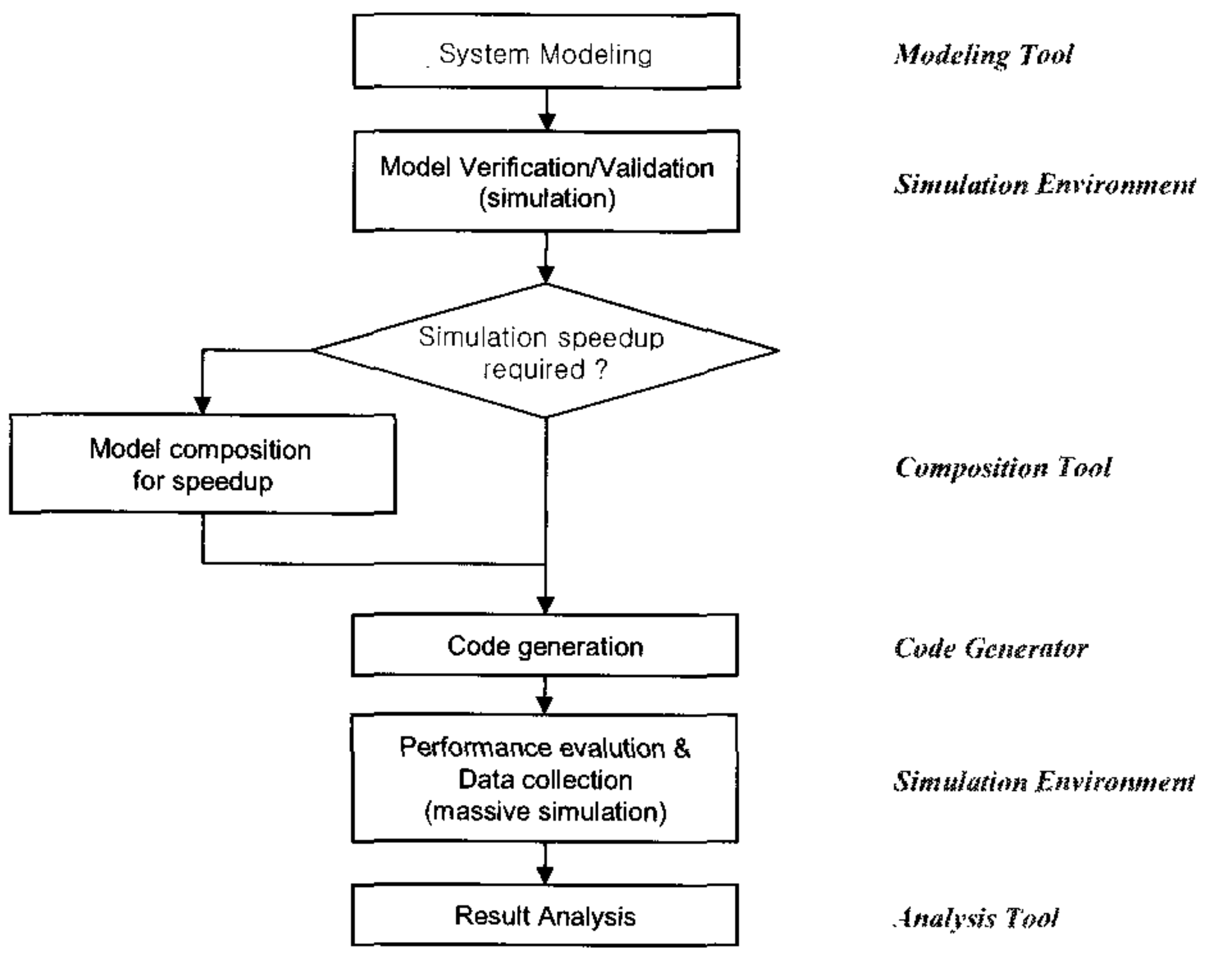

System Modeling
Modeling Tool
Model Verification/Validation (simulation)
Simulation Environment
Simulation speedup required ?
Model composition for speedup
Composition Tool
Code generation
Code Generator
Performance evalution & Data collection (massive simulation)
Simulation Environment
Result Analysis
Analysis Tool

Fig. 1 DEVSIF system development framework.

B. DEVSIF Representation

DEVSIF has three parts to describe a simulation model, which are interface, atomic model, and coupled model. Interface part specifies a set of input/output events that is common to an atomic model and a coupled model. A DEVSIF model pre-serves model information in the DEVS formalism and supports object-oriented feature. The general DEVSIF of atomic or coupled models are as follows:

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

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

Fig. 2 DEVSIF specification.

Both the atomic and the coupled DEVSIF specify their interface description and behavioral specification separately. The input event set and the output event set are defined in the interface area.

The behavioral specification for atomic DEVSIF consists of seven attributes. The four characteristic functions in an atomic model of DEVS are mapped in a one-to-one manner to those specifications in DEVSIF such *as internal transition, external transition, output function and time advance*. And the *initial condition* defines the initial state of each state variable. The *member* section enables us to specify quite complex form of functions in a modular manner. Local functions in *member* section are similar with the private member function in object-oriented language. Without the *member* specification, some of the transition rules might be described in a quite long length.

The coupled DEVSIF specifies 5 attributes: *components, external input coupling, external output coupling, internal coupling*, and *select*. Each of the attributes has the same semantics as that of the corresponding items in a coupled DEVS specification. But, as the semantics of *select* function has not been concretely specified in the formalism, we concretized its meaning as priority order. The first item in the list *select* of a DEVSIF has the highest priority while the last one has the lowest priority. If *select* is not specified in the DEVSIF specification, priority order is the same as the sequence of their appearance in a *component* section.

Furthermore, some other useful features are also provided in DEVSIF. The DEVSIF can be linked with other object codes. Thus some of the routines can be developed in C or C++ language. And they can be linked together to form an overall model. It is another advantageous feature of DEVSIF that one DEVSIF specification can include the other DEVSIF by using the keyword '*include*'. Consequently, a large and complex system can be divided into several sub systems and may be modeled as separate DEVSIF files. But they can constitute an overall system including the pre-developed models.

C. Example

Example DEVSIF specifications for the atomic model *BUF* and for the coupled model *SSQ* is described in Fig. 2. The detailed behavior of the model is found at [3].

```
interface Buff
   inputs: { in, ready}
   outputs: { out}
end Buff;
atomic model Buff
   state variables:
      proc_status : {IDLE, BUSY};
      q_length : integer;
   initial condition:
      proc_status := IDLE;
      q_length := 0;
   internal transition:
      ((q_length>0)&&(proc_status==IDLE)) => {
         q_length:=q_length-1;proc_status:= BUSY;
      }
   external transition:
      (proc_status==BUSY) * ready => { proc_status:= IDLE; }
      (proc_status==IDLE) * in => { q_length:=q_length+1; }
      (proc_status==BUSY) * in => { q_length:=q_length+1;continue; }
   output function:
      ((q_length>0)&&(proc_status==IDLE)) => out;
   time advance:
      ((q_length>0)&&(proc_status==IDLE)) => 0;
      ((q_length>0)&&(proc_status==BUSY)) => infinity;
      (q_length==0) => infinity;
end Buff;

interface SSQ
   inputs: {}
   outputs: {}
end SSQ;
coupled model SSQ
   component: {
      Genr : Generator,
      Transd : Transducer,
      Buf : Buff,
      Proc : Processor
      }
   external input coupling: {  }
   external output coupling: {  }
   internal coupling: {
      Genr.out -> Buf.in,
      Buf.out -> Proc.in,
      Proc.done -> Buf.ready,
      Proc.done -> Transd.done,
      Transd.stop -> Genr.stop
      }
end SSQ;
```

Fig. 3 DEVSIF example.

III. SIMULATION OVERHEAD

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.

Lee and Kim [3] has shown that the composition method could be successfully applied to increase the simulation speed. Our tool, DEVSIF composer, can build a composed model automatically just within a second from the conventional DEVS components.

IV. DEVSIF COMPOSER

DEVSIF Composer is a tool that does the composition operation on DEVSIF models. It is expected that the composed model would be executed faster than the original one, since it does not employ message passing activity. However, it is a disadvantage of composition that the composed model shows bad readability and is difficult to maintain them for later modification.

DEVSIF composer currently runs on Linux platform. The tool operate fast enough as the composition is just a process of merging the parts of the transition rules of atomic models into a combined transition function. In contrast, the composition operation in the area of static analysis area of model checking needs to explore all the reachable states from the initial states, which requires very expensive computation power in fact.

A. Implementation of Composition Process

The composition process consists of three step procedures.

The first step is the process of flattening all. The hierarchical model structure becomes flattened after this step. In this process, the priority order between the components are decreases as it appear later at the *select* statement.

Renaming process is the next step. If two or more of instances are created from the same atomic model, then the names of their member state variables are the same. To avoid referencing collision, all the symbol labels used for state variables need to be renamed in this step.

Finally, the third step is the process of merging the state transition functions which are influenced by the same event. While, the scheduling job is done at coordinators in a hierarchical manner at the original model, the composed model should have its own scheduling capability. Thus additional scheduling mechanism is required and there is a class library which supports scheduling task.

Following this procedure, a composed model can be constructed.

B. Considerations for Efficient Code Generation

Following points were taken into consideration such that DEVSIF composer can generate more efficient code.

1) Using absolute time for scheduling rather than relative time

Whenever an event is executed, all the influenced state variables are updated. These state variables include not only the state variables of components but also the variables used to store the elapsed time of components. Notice that the total state space is the Cartesian product of (s_i, e_i) space. e_i is a clock variables used to store the elapsed time of i'th component after it made a state transition. Whenever an event happens each e_i tends to be modified.

Specifically, if the component of the event is an influencee or an influencing one, its value is updated by a time advance function. Otherwise, it is decremented by the amount of elapsed time. This mechanism has a drawback in that almost every clock variables need to be updated whenever an event occurs. We can overcome this drawback if we manage the scheduling times of components as absolute values rather than relative ones. The next influencing component has the minimum of the left time to next schedule.

Representing the scheduling time in an absolute time-base makes no difference to this scheduling job. Moreover, both the elapsed time e_i and the left scheduling time σ_i can be computed from the absolute time representation. As a result, the generated code for scheduling job became based on an absolute time-base. It is expected that this mechanism would be more efficient when the number of composed components increases.

2) Linking with external scheduling library

The composed model must have its own local scheduling capability to choose the next event happening. This is resolved by the internally reserved function, *Local_sched()* which resides in an external library, its source code was built of C-language. Currently, the *Local_sched()* function is implemented based-on linked list data structure and requires the other auxiliary state variables such as *comp_no*, *tNext*, and *prio* as argument variables. *comp_no* denotes the number of components that will be aggregated and the variables *tNext* and *prio* means the next schedule time and the static priority of each component. The next schedule time stored in *tNext* is specified an absolute time. Because, these variables are defined in DEVSIF specification it needs to be shared with the external scheduling library. Thus, they are defined as shared type in DEVSIF specification. *Local_sched()* selects the index of an element that has the minimum value of *tNext*. If two or more components have the same minimum value, then it selects one which has the least value of priority. The element corresponds to the least element of partial order relation.

3) Enhancing code readability by using member functions

The composed model has many state variables and state transitions. This complexity becomes larger as the more components are attended. Thus, the composed model might show bad readability and it may be difficult to understand. Each merged function is a combination of the characteristic functions of the influencing or the influences. To enhance the readability, the code generator defines all the related characteristic functions as member functions. And the functions are called at the merged characteristic function describing that they have an influence relation with a dedicated influencer.

After composition, the generated code reflects the influence relations among the components. For example, when the model *Proc* generates an output event *done*, an internal transition rule of *Processor* and an influenced external transition rule of the *Buff* are invoked. They are placed in the same block in a composed transition rule and executed sequentially. Because there takes place no message passing in this process, it is expected the simulation run-time would be much less. Quantitative measure of the simulation speedup would be given in the next section.

V. EXPERIMENTAL RESULTS

The simulation speedup has been evaluated by the use of a discrete event simulation environment, DEVSim++[8]. Three kinds of example models were tested on an IBM 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. 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*. The third model, Columbian Health Care System (CHCS) has been experimented. The model has been used in various studies[10].

Table 1 indicates the experimental results. Experiments were done for several cases varying the number of components. We measured two cases of simulation time for each model. The first is the simulation time of the original model, the second is that of the model obtained by the composition tool, DEVSIF composer. As shown in Table 1, the composition improved the execution speed approximately 5.3 ((8.1+5.4+4.6+4.5+3.9+5.7+5.1+ 5.0) / 8) times on average. Table 1 shows that the simulation speedup has been decreased very slightly as the components of the overall system are increased. This fact is because that our external scheduling library was implemented as a linked list of which computation time is linear proportional to the number of components.

Table 1 Simulation speedup

Example name	Type	Sim. Time (sec.)	Speedup
SSQ	Original	66.6	1.0
	Composed	8.1	8.1
CSMA/CD3	Original	121.0	1.0
	Composed	22.6	5.4
CSMA/CD6	Original	214.6	1.0
	Composed	46.6	4.6
CSMA/CD9	Original	334.2	1.0
	Composed	74.8	4.5
CSMA/CD12	Original	431.9	1.0
	Composed	111.0	3.9
CHCS8	Original	8.0	1.0
	Composed	1.4	5.7
CHCS14	Original	15.5	1.0
	Composed	3.0	5.1
CHCS22	Original	28.4	1.0
	Composed	5.7	5.0

VI. CONCLUSION

This paper introduces an automated compiled simulation tool, DEVSIF composer, which improves significant speedup for discrete event simulation. The tool is an implementation of model composition which was introduced at [3]. Since message passing time among the components was shortened after applying the tool, we could easily accelerate the simulation speed.

By experimenting three kinds of example models, we found that the composed model could be simulated approximately 5 times faster that the original model. And because the synthesis time taken by the tool was less than a second, the tool must be a very effective one.

However the data structure adopted at the tool is based on linked list which would be inefficient when the number of components becomes larger. In this case, more elegant data structures or algorithms can be applied such as MIN heap[11].

ACKNOWLEDGMENT

This work was supported in part by Kongju National University (KNU), Project No. 2007-0097.

REFERENCES

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

[2] 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.

[3] W. B. Lee, T. G Kim, "Performance Evaluation of Concurrent System Using Formal Model: Simulation Speedup", *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E86-A, no. 11, pp: 755-2766, Nov. 2003.

[4] 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.

[5] K. J. Hong and T. G Kim, "DEVSIF : Relational Algebraic DEVS Intermediate Format," *Proceedings of AIS'2000*, Tucson, Arizona, U.S.A., 2000.

[6] G P. Hong and Tag G Kim, "A Framework for Verifying Discrete Event Models Within a DEVS-Based System Development Methodology," *Transactions of the Society for Computer Simulation*, vol. 13, no. 1, pp.19-34, 1996.

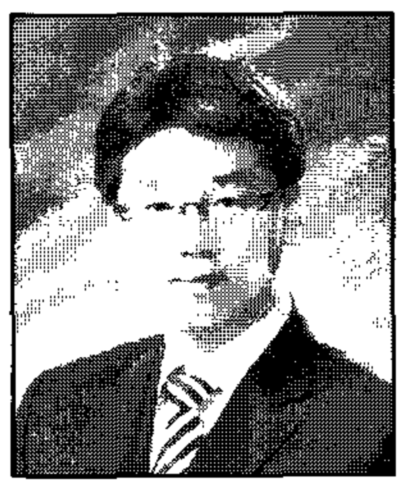
[7] C. Thomas, H. Luckhoff, and T. G Kim, "OpenDEVS: A Proposal for a Standardized DEVS Model Exchange Format", *Proc. of AIS '96*, pp. 371-377.

[8] 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.

[9] 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.

[10] D. Baezner, G Lomow, and B.W. Unger. "Sim++: The Transition to Distributed Simulation." *In Proceedings of the SCS Multiconference on Distributed Simulation*, 1990.

[11] J. W. J. Williams. Algorithm 232 - Heapsort, Communications of the ACM 7(6): 347–348, 1964.



Wan-Bok Lee
received the B.E, M.E, and Ph.D. degrees in electrical engineering from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea, in 1993, 1995, and 2004, respectively. He is presently an Assistant Professor at the Department of Game Design, Kongju National University, Kongju, Chungnam, Korea. His research interests include game programming, methodology for modeling and simulation of discrete event systems, and information security.