A Formal Specification Framework for Developing Parallel Software based on the DEVS formalism

Bong-Joon Jung Young-Rak Seong Tag Gon Kim Kyu Ho Park

Department of Electrical Engineering
Korea Advanced Institute of Science and Technology
Taejon, Korea 305-701

Abstract

This paper proposes a formal specification framework for developing parallel software. The framework employs a parallel-software-extended DEVS (SDEVS) formalism to specify parallel software in modular, hierarchical fashion and to express specification-level parallelism. To execute developed SDEVS specification, it is automatically transformed into DEVS models for parallel simulation. During the transformation, the degree of parallelization is optimized to an underlying target architecture. To show the correctness of SDEVS, a parallel finite difference method (FDM) problem is specified and executed. The experimental results show that the SDEVS formalism specifies parallel software well-defined semantics and properly utilizes the inherent parallelism of the problem.

1 Introduction

Parallel programming is more difficult than sequential programming due to data decomposition, communication and synchronization among processors. Many parallelizing compilers and their execution environments have been proposed to reduce the difficulties. However, most of them adopted sequential programming languages as their parallelism representations for programmer's familiarity. For this reason, complex analyses and optimizations are needed to achieve effective performance from the sequential semantics.

Recently, researchers pay attention to formal or graphical specification methods for their high-level parallelism denotation and easy verification. David B. Skillicorn proposed an architecture-independent parallel computation methodology that is based on the Bird-Meertens formalism [4]. F. Vallejo et al. proposed a job level programming paradigm using extended Petri net on event-driven multiprocessor systems [6]. On the other hand, PAR-SDL adopted SDL (Specification and Description Language) which permits graphical specification and validation methods. They also developed an SDL-to-C translator that accepts most of SDL constructs and generates hardware independent C codes [3]. In our opinion, they need good parallelism abstractions and should use state-of-the-art software engineering techniques.

In this paper, we propose a new formal specification framework for developing parallel software. To describe software formally, we devise a parallel-software-extended DEVS (SDEVS) that is based on the DEVS (Discrete Event System Specification) formalism. Modular and hierarchical structures of SDEVS help us to specify complex software systems efficiently and to increase the reusability of developed software. The inherent parallelism of software is expressed explicitly by an abstracted denotation of SDEVS. A parallel DEVS simulation environment is used for the execution of SDEVS. Simulation for parallel software development allows additional fast prototyping and easy debugging merits [5].

After briefly describing the proposed framework overview, we explain the DEVS formalism and some backgrounds in Section 3. In Section 4 and 5, SDEVS and its language representation are described, respectively. Section 6 includes an SDEVS-to-DEVS transformation procedure. A parallel FDE equation and its experimental results are discussed in Section 7. Finally, we address our conclusion in the last section.

2 SDEVS framework

Figure 1 shows the framework of this paper. We can formally describe parallel software with three basic SDEVS models. An atomic model of SDEVS rep-

resents a basic sequential software module. A coupled model has the structural and coupling information among its children. A parallel model expresses inherent parallelism explicitly.

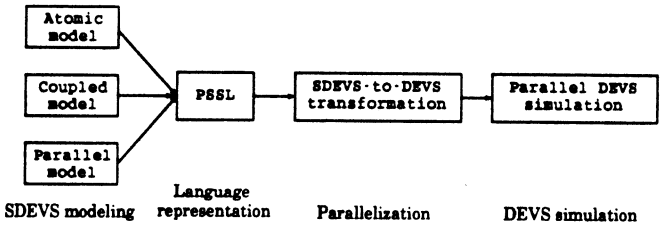


Figure 1: SDEVS framework

PSSL (parallel software specification language) is developed as a language representation of the SDEVS formalism. It concisely denotes SDEVS in well-defined fashion. Finally, the SDEVS-to-DEVS transformer automatically converts SDEVS specification into its equivalent DEVS models for parallel execution.

3 Backgrounds

DEVS formalism

A set-theoretic formalism, the DEVS formalism specifies discrete event systems in a hierarchical, modular form [1] [2]. In the DEVS formalism, one must specify the basic models from which larger ones are built and indicate how these models are connected together in hierarchical fashion. A basic model AM, also called an atomic model, is defined (1).

$$AM =< X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta > \qquad (1)$$

where,

X : External input events set;
Y : External Output events set;
S : States set;
δ_{int} $(S \rightarrow S)$: Internal transition function;
δ_{ext} $(Q \times X \rightarrow S)$: External transition function;
λ $(S \rightarrow Y)$: Output function;
ta $(S \rightarrow R^{+}_{0,\infty})$: Time advance function;

The set Q means the global state of a model or general state variables with time informations like (2).

$$Q = \{(s, \epsilon) | s \in S \; and \; 0 \leq e \leq ta(s)\} \qquad (2)$$

We can specify atomic model's dynamic nature with state variables and their four transition functions.

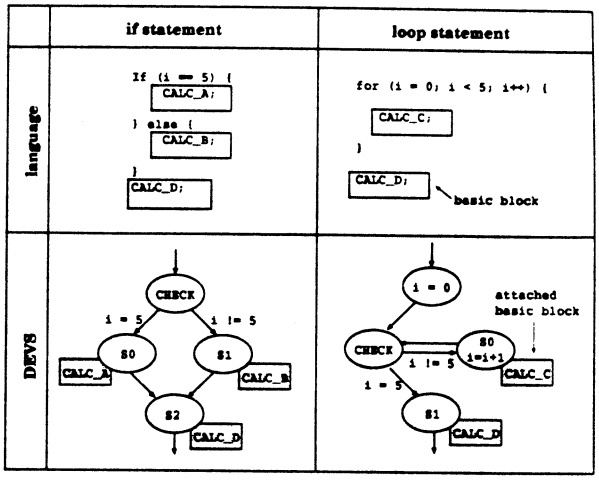


Figure 2: The relation between language and DEVS

When a model receives an external event, it determines its next state by executing the external transition function (δ_{ext}). The internal transition function (δ_{int}) has state transition information when an internal timeout event is occurred. The output function generates output events. Finally, the time advance function has the timeout value of a state.

The coupled model contains DEVS models as its children as well as the couplings between them. $SELECT$ determines a selection priority among children when they change their states simultaneously.

$$CM =< X, Y, M, EC, IC, SELECT > \qquad (3)$$

where,

X : Input events set;
Y : Output events set;
M : Children models;
IC $(\subseteq M_{out} \times M_{in})$: Internal coupling relation;
EC $(\subseteq CM_{out,in} \times M_{out,in})$: External coupling relation;
$SELECT$: Tie-breaking selector;

The DEVS models are modular because they communicate with others only through their I/O ports. In addition, the coupled model helps us to describe a complex system in a hierarchical manner.

DEVS as parallelism representation

We believe that DEVS is a good formalism for describing discrete event systems. A software system is considered as a kind of discrete event system. Thus,

we adopt the DEVS formalism for specifying the system. If we look at a software system in a pool of basic blocks and control flows that activate the blocks, the following mechanism will be used to represent software by DEVS. First, the control flows correspond to state transitions in DEVS. Second, the behavior of a software module is divided into basic blocks, and attached to model's states like Figure 2. They are activated whenever the model changes its state. Finally, events are used to communicate data between the software modules.

Because DEVS supports concurrent control flows, it is natural to specify functional parallelism using the DEVS formalism. To generate such concurrent control flows in DEVS, we use a multicast coupling scheme. In the scheme, an output event is multicasted into several connected input ports simultaneously. As a result, such multiple input events produce concurrent state transitions of receiver models. We can also specify data parallelism by spreading data when an event is multicasted.

DEVS simulation environments

Currently, several DEVS based tools have been developed for model development, execution, and simulation. DEVSim++ is an object-oriented DEVS simulation environment written in C++ language [10]. The environment supports user to develop hierarchical, modular discrete event models. P-DEVSim++ is a parallel extension of DEVSim++ [11]. It can simulate DEVS models developed by DEVSim++ in a distributed-memory architecture. For this, it partitions and maps DEVS models into processors while exploiting full parallelism of models.

4 SDEVS formalism

As mentioned in Section 1, SDEVS is an extension of the DEVS formalism. An SDEVS atomic model is similar to that of DEVS except *actions* set and *action function*. The *actions* set includes basic blocks of a software module. The *action function* acts a mapping function between the basic blocks and states of a model. The logical time of SDEVS is used to synchronize events because all SDEVS events are synchronized by the logical time.

$$AM =< X, Y, S, A, \delta_{int}, \delta_{ext}, \lambda, ta, \alpha > \qquad (4)$$

where,

X,Y : Input/Output events set;
S : States set;
A : Actions set;
δ_{int} $(S \rightarrow S)$: Internal transition function;
δ_{ext} $(Q \times X \rightarrow S)$: External transition function;
λ $(S \rightarrow Y)$: Output function;
ta $(S \rightarrow R^{+}_{0,\infty})$: Time advance function;
α $(S \rightarrow A)$: Action function;

Coupled model, which has coupling information between models, has the same characteristics as DEVS formalism.

As one of important features of this paper, SDEVS parallel model represents the parallelism of a given problem. It is an abstracted model that contains scalable behavior of an atomic model and a multicast coupling scheme. The scalable nature is used for model decomposition during parallelization. All the external couplings are also converted into multicast couplings. We can exploit functional parallelism by using the multicast scheme. To describe the abstraction formally, we add two kinds of constructs; *parallel coordinates* (PC) and *coupling vector* (CV). An SDEVS parallel model is defined in (5).

$$PM =< X, Y, S, PC, A, \delta int, \delta ext, \lambda, ta, \alpha, ICV > \qquad (5)$$

where,

X, Y : Input/Output events set;
S : State set;
PC : Parallel coordinate set;
A : Actions set;
δint $(S \rightarrow S)$: Internal transition function;
δext $(Q \times X \rightarrow S)$: External transition function;
λ $(S \rightarrow Y)$: Output function;
ta $(S \rightarrow R^{+}_{0,\infty})$: Output function;
CV $(PM[PC]_{in} \times PM[PC]_{out})$: Coupling vector;
α $(S \rightarrow A)$: Action function;

The PC is a set of parallel coordinates referenced in the context of an SDEVS parallel model. Using the PC in an array index variable causes decompositions along the referenced dimension. The loops that have PC are also decomposed. When data dependencies exist among decomposed data, data transfers are occurred through the couplings defined by CV.

Example of SDEVS

Suppose an image processing software module that receives 100×100 image, processes it, and then, sends a result for the next processing. To simplify the example, we set one parallel coordinate in vertical direc-

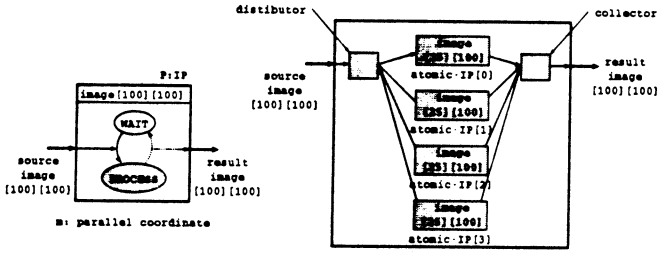
tion of the image. Figure 3 shows the graphical representation of the parallel model P:IP with its equivalent block diagram. The figure shows that the P:IP model is converted into four atomic models during parallelization procedure. To distribute and collect the original image, a distributor and a collector models are inserted, respectively.

There are two states : WAIT and PROCESS in the model. It waits for an input image in the WAIT state. The image is processed and sent to another model in the PROCESS state. An action function named process_image is attached to PROCESS state for actual image processing.

$$IP =< X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta, IC, ICV, I, \alpha > \quad (6)$$

where,

$X = \{image\_in\}$,
$Y = \{image\_out\}$;
$S = \{phase, image\epsilon[y = 100][x = 100]\}$;
$phase = \{WAIT, PROCESS\}$;
$\delta_{int}(PROCESS) = WAIT$;
$\delta_{ext}(WAIT, image\_in) = PROCESS$;
$\lambda(CALC) : image\_out$:
$ta(WAIT) = \infty$;
$ta(PROCESS) = 1$;
$\alpha(PROCESS) = \{process\_image\epsilon\}$;
$I = \{y\}$;



(a) Image processing parallel model (b) Equivalent block diagram

Figure 3: The equivalent block diagram of P:IP parallel model

5 PSSL

PSSL (parallel software specification language) is developed as a representation of the SDEVS formalism. It supports various basic types including abstract data type (ADT). Almost all keywords have one-to-one correspondences with SDEVS to describe the formalism exactly. Moreover, the syntax of the language is as concise as possible for easy writing and reading.

Basic declaration and statements

PSSL basic declarations are used to define parallel coordinates, variables, and events. It supports basic types such as integer, float as well as structured types. Furthermore, an abstract data type (ADT) is supported for a message type definition. To efficiently denote scientific and engineering calculation problems, PSSL supports full arithmetic and logical operators.

There are three statement types in PSSL. First, test statements are used to identify last accessed ports and current states. Second, PSSL also supports conditional branch and loop statements. Finally, an assignment statement can be used to update variables. These statements can be used at all characteristic functions and action definitions.

- test statements : PORT, STATE
- control branch statements: IF, LOOP
- assignment statement

Definition and description sections

There are six definition sections in PSSL. We can specify parallel coordinates, message types, state variables, I/O ports, and actions by using these sections.

- PARALLEL section defines a parallel coordinate.
- MESSAGE section defines message types.
- STATE_VAR section defines state variables.
- INPUT_PORT section.
- OUTPUT_PORT section.
- ACTIONS section defines actions.

The remainder non-definition sections are used to describe SDEVS behavior of a model. The characteristic functions of atomic model and the couplings of coupled model are specified in these sections.

- EXTERNAL_TRANSITION section
- INTERNAL_TRANSITION section
- OUTPUT section
- TIME_ADVANCE section
- COUPLINGS section defines coupling informations.

Example of PSSL

Let's consider the S:IP model specified in Section 3. We can describe PSSL code for the parallel model without any parallelization details like Figure 5.

```
PARALLEL_MODEL ip
COORDINATE
    m = 100;
MESSAGE
    struct {
        float image[m][10];
    } Image_t;
INPUT_PORT
    Image_t image_in;
OUTPUT_PORT
    Image_t image_out;
STATE_VAR
    int phase = 0;
    float Image[m][10];
EXTERNAL_TRANSITION
    STATE phase == 0 :
        PORT image_in :
            IN msg(image_in);
            ACTION update_image(msg);
            phase = 1;
        END_PORT;
    END_STATE;
INTERNAL_TRANSITION
    STATE phase == 1 :
        phase = 0;
    END_STATE;
OUTPUT
    STATE phase == 1 :
        NEW Image_t msg;
        ACTION process(Image, msg);
        OUT msg(image_out);
    END_STATE;
TIME_ADVANCE
    STATE phase == 1 : TIME 1; END_STATE;
    STATE DEFAULT : TIME Infinity; END_STATE;
ACTIONS
    FUNC update_image
    ARG  float[m][10];
    BODY
        // real calculations
        ....

    END_FUNC;

    // other action definitions
    ....

END_MODEL;
```

Figure 4: A PSSL code of image processing

6 Parallelization

Developed SDEVS specification is converted to equivalent DEVS models for parallel execution. The SDEVS-to-DEVS transformer reads PSSL codes, analyses each models, then, generates P-DEVSim++ codes. During the transformation, all actions are converted to C++ functions. An SDEVS atomic model is converted to its DEVS model by adding C++ function calls in its transition functions. For an SDEVS parallel model, the degree of model decomposition should be evaluated before model conversion.

At the first glance, it seems to be true that the larger number of decomposition are taken, the more parallelism we achieve. However, the maximum degree of parallelism is decided by the following overhead of target systems. In a shared-memory multiprocessor, communication time to send or receive events between models is negligible. So, the degree of decomposition is determined by the the number of processors and the overheads of system calls such as fork and RPC operation. If we concentrate on the distributed-memory architecture, we must consider the communication overhead of the system. In this paper, we only consider the transformation in a distributed-memory architecture.

System parameters

The SDEVS-to-DEVS transformer use two system parameters to efficiently parameterize the communication overhead. It is well known that basic communication setup time exists in real communications. The time is mainly composed of software overhead and remains constant for any data size. For a small packet size, the setup time often takes a major factor in communication time. So, we estimate the communication time with a constant setup time and actual data transmission time that is proportional to data size.

Definition 4.1 *granularity factor* ρ
A ratio between the unit communication and computation time of a target system.

$$\rho = \frac{t_{comm}}{t_{comp}} \tag{7}$$

Definition 4.2 *Setup factor* σ
Communication setup time measured in unit communication time.

$$\sigma = \frac{S}{t_{comm}} \tag{8}$$

where S means average setup time.

Both of the factors are important measures because they determine the grain size of parallel problems. We think that it is a good approach not to take care of the grain size of a problem.

Cost function

Using ρ and σ, we devise a cost function to determine the degree of model decomposition. The cost function is defined as the total estimated execution time of transformed DEVS models. The estimated execution time of a parallel model is composed of three parts shown Figure 5. They are the time to distribute and collect the events (DIST(i), COLL(i)), the time to calculate data (COMP(i)), and the time to communicate events with each decomposed models (D(l,i)).

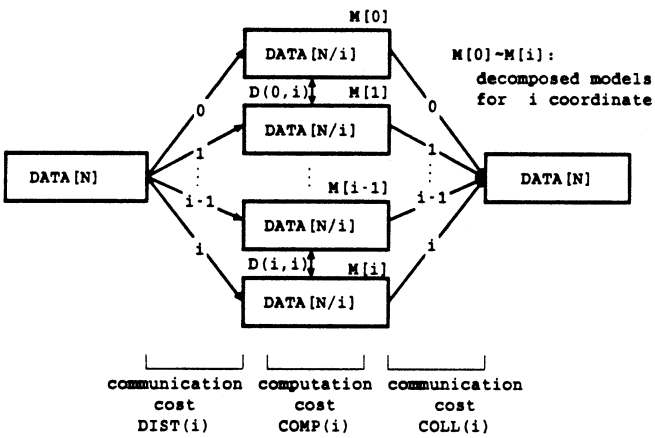


Figure 5: The block diagram to construct CF(i)

$$\begin{aligned} CF(i) &= DIST(i) + COLL(i) + COMP(i) + IC(i) \\ &= 2(\tfrac{V_i}{i} \times t_{comm} + S) \times i + (\tfrac{P_i}{i} \times t_{comp}) \\ &\quad + \textstyle\sum_l D(l)_i \times t_{comm} \end{aligned} \tag{9}$$

where, V_i means the data size to compute along the parallel coordinate i. We can get this value by profiling all action functions used in a parallel model. P_i means total computation time along i coordinate. $D(l)_i$ represents the data size to transfer through link l for i coordinate.

The equation further can be reduced to (10). The equation shows that the first term is constant and the second and the third terms have trade-off relation for i. So, there might exist an optimal i value that minimizes the cost function.

$$\frac{CF(i)}{t_{comm}} = 2(V_i + \sigma \times i) + \tfrac{1}{\rho}\tfrac{P_i}{i} + \textstyle\sum_l D(l)_i \tag{10}$$

Example of parallelization

Consider the simple `P:IP` model shown in Section 3. An estimation of P_m values are achieved through profiling **process_image** action function. We use the following model and system parameters.

- $I = \{m\}$
- $V_m = 100$, 100 data per m coordinate
- $P_m = 9000$, 90 machine cycles per one calculation
- $\sum_l D(l)_i = 0$, No CV
- $\rho = 12$
- $\sigma = \frac{75}{3}$
- $Proc = 4$

The cost function is reduced to (11). The function is minimized when $m = 3.6$. Therefore, we can get the minimum execution time when the parallel model is decomposed to three atomic models.

$$\frac{CF(m)}{comm} = 2(100 + \frac{75}{3} \times m) + \frac{9000}{12m} \tag{11}$$

7 Experiments

An parallel FDE (finite difference equation) solver is specified in SDEVS and executed in P-DEVSim++ running on a KAICUBE, a hypercube computer developed at the KAIST CORE lab. The FDE is an equation to solve the two-dimensional Laplace equation. When we solve the problem by the FDM (Finite Difference Method), it divides the problem domain into finite squares. Equation (12) shows that the value of any square ϕ is calculated from its four neighbors.

$$4\phi_{xy} - \phi_{x-x'} - \phi_{y-y'} - \phi_{x+x'} - \phi_{y+y'} = 0 \tag{12}$$

There are several methods for parallelizing the FDM [7]. We use a domain decomposition technique to parallelize the problem. In this case, mesh-like nearest neighbor communications are produced from data dependencies between the decomposed data. We choose **x, y** parallel coordinates to parallelize the problem. To consider data dependencies along **x, y** coordinates, we also set two communications vectors: (0, 1), (1, 0). The two CV of the FDE parallel model generates a mesh coupling between decomposed models shown in Figure 6.

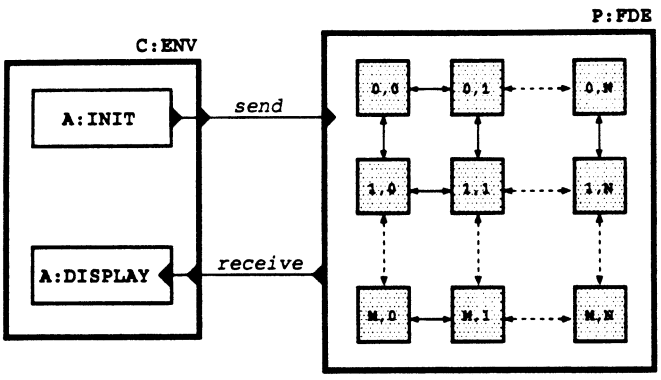


Figure 6: The global SDEVS architecture of FDE

The figure shows the global model structure of the problem. A SDEVS coupled model `C:ENV` has two atomic models as its children. The `A:INIT` model initializes the input data. After that, it sends the data to the `P:FDE` parallel model. The `A:DISPLAY` model accepts results from `P:FDE` model and displays the result. The `P:FDE` parallel model accepts data, processes it, then, sends to `A:DISPLAY` model.

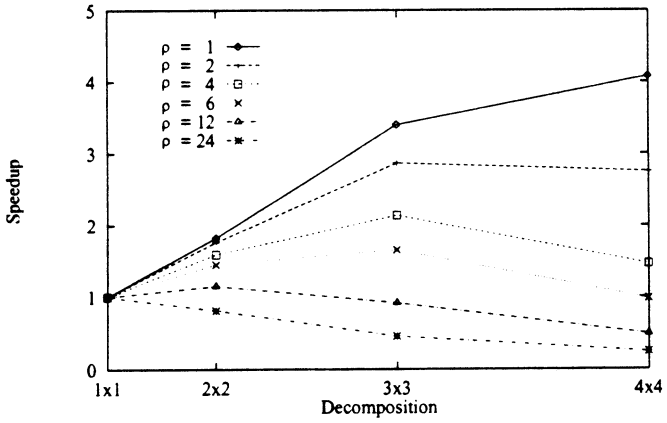


Figure 7: Speedup vs. various ρ values

The FDE example is executed for several decompositions with changing ρ. Figure 7 shows some important results. First, an optimal decomposition exists for a certain ρ. Second, the optimal decomposition moves to smaller sizes for a system that has larger ρ. In a high ρ system ($\rho = 24$), the expansion is meanless for the FDE problem. Finally, the maximum speedup point nearly follows the devised cost function in Section 4.

The facts show that the SDEVS formalism specifies software in well-defined semantics. It also properly utilizes the inherent parallelism of the parallel software. Furthermore, the devised cost function determines an optimal model decomposition for each parallel coordinates.

8 Conclusion

SDEVS is an extended DEVS formalism for describing parallel software in a hierarchical, modular manner. We can describe an architecture-independent parallelism at the specification-level within the formalism. An SDEVS atomic model represents a basic sequential software module. A coupled model has the hierarchical structure of a software system and the coupling information between models. A parallel model is an abstracted model that contains scalable behavior of a natomic model and a multicast coupling scheme. Through the SDEVS-to-DEVS transformation, a parallel model is converted into DEVS models with model decomposition. The amount of the decomposition is automatically evaluated during the transformation using target system dependent parameters. An experiment of a parallel FDM problem shows that SDEVS can properly utilizes the inherent parallelism of a problem and generates correct results.

References

[1] Bernard P. Zeigler, "Multifaceted Modeling and Discrete Event Simulation," ACADEMIC PRESS, 1984.

[2] Bernard P. Zeigler, "Theory of Modeling and Simulation," ROBERT E. KRIEGER PUBLISHING COMPANY, INC, 1985.

[3] Christoph Steigner, Ralf Joostema, Christian Groove, "PAR-SDL: Software design and implementation transputer systems," in *Transputer Application and Systems*, 1993, pp. 1083–1095.

[4] David B. Skillicorn, "Architecture-Independent Parallel Computation," in *Computer*, Vol. 23, No. 12, Dec., 1990, pp. 38–50.

[5] Eric A. Brewer and William E. Weihl, "Developing Parallel Applications Using High-Performance Simulation," *Workshop on Parallel and Distributed Debugging*, 1993.

[6] F. Vallejo, J. A. Gregorio, M. Gonzalez Harbour, and J. M. Drake, "Shared Memory Multiprocessor Operating System with an Extended Petri Net

Model," in *IEEE transactions on parallel and distributed systems*, Vol. 5, No. 7, 1994, pp. 179–762.

[7] Ian G. Angus, Geoffrey C. Fox, Jai Sam Kim, David W. Walker, "Solving Problems On Concurrent Processors Volume II," Prentice-Hall, 1990.

[8] Kai Hwang, "Advanced Computer Architecture," MacGRAW-HILL,1993.

[9] Kenneth J. Turner, "Using Formal Description Techniques," Wiley, 1993.

[10] Sung-Bong Park, "DEVSim++: A Semantic Based Environment for Object-Oriented Modeling of Discrete Event Systems," M.S Thesis, KAIST, 1993.

[11] Yeong Rak Seong, "Parallel Simulation of Hierarchical, Modular Discrete Event Models," Ph.D. Thesis, KAIST, 1994.