DEVSIF : RELATIONAL ALGEBRAIC DEVS INTERMEDIATE FORMAT

Ki Jung Hong*, Tag Gon Kim*, and In Sup Kwon**

*Department of Electrical Engineering & Computer Science
KAIST
Taejon, KOREA

**Pyungchang Computer & Communication Inc., Seoul, KOREA

KEYWORDS

DEVSIF (DEVS Intermediate Format), DEVS formalism, DEVS Model Reuse, Relational Algebraic DEVS

ABSTRACT

The DEVS formalism has been widely used in modeling and simulation of various discrete event systems such as computer/communication systems. However, DEVS models developed in one simulation environment may not be directly reused in the other environment. To be useful, a means to represent DEVS models in an environment-independent format needs to be devised. This paper proposes a language-independent DEVS modeling format, called DEVSIF (DEVS Intermediate Format) whose semantics is based on relational algebra. DEVSIF can be automatically converted into DEVS models executable in various DEVS simulation environments such as DEVSim++ and DEVSim-Java. An example of modeling/simulation based on DEVSIF demonstrates effectiveness of the proposed simulation method.

1 INTRODUCTION

The DEVS formalism has been widely used in modeling and simulation of various discrete event systems such as computer, communication, manufacturing systems. However, DEVS models developed in one simulation environment may not be directly reused in the other environment. To be useful, a means to represent DEVS models in an environment-independent format needs to be devised.

This paper proposes the framework for the design and simulation of the language-independent DEVS model, called DEVSIF whose semantics is based on relational algebra. DEVSIF is an extension of DEVS spec language (Hong and Kim 1996), which is devised for the behavioral analysis of the model with no timing information. A similar specification language, called openDEVS, was defined in (Thomas et.al. 1996) which has three characteristics: preservation of the DEVS models information, object-oriented modeling, and model type-check. The proposed DEVSIF specification includes all these features. More importantly, a translator is developed which converts a DEVSIF model to a relational algebraic model for easy reuse and maintenance. The relational algebraic DEVSIF model, called the RADESIF model, can be stored in a database or a file-system, then loaded, searched, and modified in a simulation environment for reuse. To automatically generate simulation code for a specific simulation environment, we developed code generators for C++ DEVS models or Java DEVS models. Such simulation models can be directly executable in the DEVSim++ or DEVSimJava simulation environment.

This paper is organized as follows. The DEVSIF framework is introduced in more detail at section 2, a complete example of the framework is presented at section 3, and we conclude the paper in section 4.

2 DEVSIF BASED SIMULATION FRAMEWORK

Figure 1 shows a simulation framework based on the proposed DEVSIF methodology. Within the framework, a modeler uses DEVSIF specification for modeling of discrete event systems. The specification is translated into a relational algebraic DEVS model by a DEVSIF translator, which then is stored in a relational database or a equivalent file system. Now, the stored model can be reused in the code generation phase to generate various simulation models depending on the target simulation environment. Moreover, the stored model can be directly simulated by an appropriate interpreter designed by a general purpose

language or a database language. Of course, such interpretation sacrifices execution speed. The following subsections discuss the theoretical basis for DEVSIF in conjunction with the DEVS formalism and Relational Algebra.

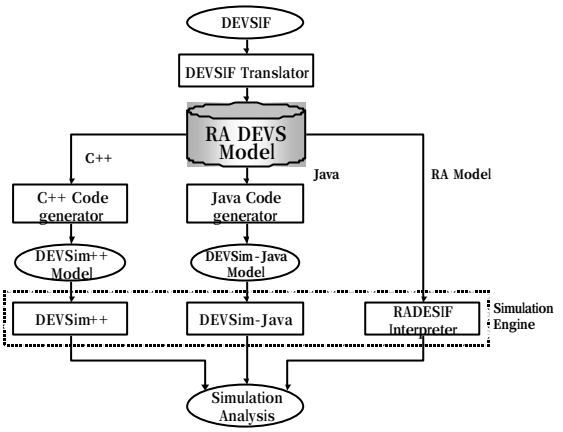

Figure 1: Simulation Methodology Using DEVSIF

2.1 DEVS Formalism: A Brief Introduction

The DEVS Formalism specifies a model in a hierarchical, modular form. A discrete event system consists of entities whose dynamics are described as a set of procedure rules. Such rules control the interactions among the communicating entities. The communicating entities and the procedure rules can be decomposed into the smaller ones with the modeling semantics. These decomposed components are directly mapped to the atomic models, from which larger ones are built. A basic model, called an atomic model, is not further decomposed specification of the dynamic behavior of a component. Formally, an atomic model AM is specified as (Zeigler 1984) :

$$AM = < X, S, Y, d_{int}, d_{ext}, l, ta >$$
X :input events set;
S :sequential state set;
Y :output events set;
$d_{int} : S \rightarrow S$:internal transition function;
$d_{ext} : Q \times X \rightarrow S$:external transition function;
$l : S \rightarrow Y$:output function;
$ta : S \rightarrow Real$:time advance function;
$Q = \{(s, e) \mid s \in S, 0 \le e \le ta(s)\}$
:total state of AM (e :elapsed time).

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 new model can be employed as another 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 (Zeigler 1984):

$$CM = < X, Y, \{M_i\}, EIC, EOC, IC, SELECT >$$
X :input events set;
Y :output events set;
$\{M_i\}$:DEVS components set;
$EIC \subseteq X \times \bigcup_i X_i$
:external input coupling relation;
$EOC \subseteq \bigcup_i Y_i \times Y$
:external output coupling relation;
$IC \subseteq \bigcup_i Y \times \bigcup_i X_i$
:internal coupling relation;
$SELECT : 2^{\{M_i\}} - \varnothing \rightarrow \{M_i\}$: tie breaking selector.

A detailed discussion about the DEVS formalism and modeling is found in (Zeigler 1984):

2.2 Relational Algebra

Relational algebra (RA) is based on set theory. Set is described as { … }, and structure is described by using < … >. After now on, we will describe a set of structured information by using $\{< \ldots >_i\}$.

2.2.1 Basic Operator

There are six fundamental operations that serve to define relational algebra (Silberschatz 1997). Let R and S be two relations over sets of attributes R' and S', respectively.
Let $X \subseteq R'$. Then, the following are basic operators in RA.

(1) Rename r : Given a relational-algebra expression E, the expression r_x(E) returns the result of expression E under the name x.
(2) Selection s : $s_F(R) = \{t \mid F(t) \wedge t \in R\}$
(3) Projection Π : $\Pi_X(R) = \{t[X] \mid t \in R\}$
(4) Union $\cup$: $R \cup S = \{t \mid t \in R \vee t \in S\}$
(5) Difference – : $R - S = \{t \mid t \in R \wedge t \notin S\}$
(6) Cartesian product $\times$: $R \times S = \{<r,s> \mid r \in R \wedge s \in S\}$

(7) Natural join $\bowtie$: $R \bowtie S = \prod_{R \cup S'} (s_{\wedge_i (R.A_i = S.A_i)} R \times S)$

2.2.2 DEVS Models in RA : RA DEVS

By using relation algebra, we can specify discrete event models with preservation of the DEVS semantics. More specifically, the algebra has an expressive power equivalent to the DEVS formalism in specification of both an atomic DEVS model and a coupled DEVS model. We now define atomic and coupled DEVS models in relational algebra.

First, a relational algebraic atomic model AM' is specified as:

$AM' = \langle Model', X', S', Y', I', d'_{int}, d'_{ext}, l', ta' \rangle$
$Model' = \langle id, parent\_id, model\_name \rangle$
$X' = \{\langle id, event\_name, type\_id \rangle_i\}$
$S' = \{\langle id, state\_name, type\_id \rangle_i\}$
$Y' = \{\langle id, event\_name, type\_id \rangle_i\}$
$I' = \{\langle id, func \rangle_i\}$
$d'_{int} = \{\langle id, func, act\_func \rangle_i\}$
$d'_{ext} = \{\langle id, func, x, act\_func \rangle_i\}$
$l' = \{\langle id, func, y, act\_func \rangle_i\}$
$ta' = \{\langle id, func, type, ta\_val, ta\_id \rangle_i\}$
$ta'.type \in$ {INFINITY, SIGMA, RANDOM }
$func' = \{\langle id, func\_id, op, e1, e2 \rangle_i\}$
$func'.op \in$ {PLUS,ASSIGN,…}

Next, a relational algebraic coupled model CM' is specified as:

$CM' = \langle Model', X', Y', M', EIC', EOC', IC', SELECT' \rangle$
$Model' = \langle id, parent\_id, model\_name \rangle$
$X' = \{\langle id, event\_name, type\_id \rangle_i\}$
$Y' = \{\langle id, event\_name, type\_id \rangle_i\}$
$M' = \{\langle id, child\_id, model\_id, child\_name \rangle_i\}$
$EIC' = \{\langle id, model\_id, x, child\_id, child\_x \rangle_i\}$
$EOC' = \{\langle id, child\_id, child\_y, model\_id, y \rangle_i\}$
$IC' = \{\langle id, src\_child\_id, src\_child\_y, dst\_child\_id, dst\_child\_x \rangle_i\}$
$SELECT' = \{\langle id, child\_id, priority \rangle_i\}$

To prove the equivalence between RA and DEVS formalisms in expressive power, isomorphism between two formalisms needs to be established. A mapping from a RA model into a DEVS model *H* satisfies with the following equations:

$$AM = H_{AM}(AM')$$
$$CM = H_{CM}(CM')$$

Conversely, a DEVS to RA mapping function, *G*, should be satisfied with the following equations:

$$AM' = G_{AM}(AM)$$
$$CM' = G_{CM}(CM)$$

Since *AM* and *AM'* have a similar structure, *G* and *H* is easily introduced as shown in a following example in terms of the input event set *X* and the internal transition function d_{int}:

$H_{AM}.X = P_{(event\_name,type\_id)}(AM'.Model' \bowtie AM'.X')$
$H_{AM}.d_{int} = d_{int}\_I(P_{(func)}(AM'.Model' \bowtie AM'.d'_{int}))$
$\wedge\, d_{int}\_I(P_{(act\_func)}(AM'.Model' \bowtie AM'.d'_{int}))$
$d_{int}\_I : S' \rightarrow S'$
$G_{AM}.X = \textbf{\textit{Translator}}(AM)/X$
$G_{AM}.d_{int} = \textbf{\textit{Translator}}(AM)/d_{int}$

The translation procedure is a series of composition of the function G.

2.3 DEVSIF : DEVS Intermediate Format

The DEVS intermediate format is developed for the formal expression of a model of a discrete event system. The formal expression makes it easy at once to analyze, simulate and execute the model.

2.3.1 Syntax and semantics of DEVSIF

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, an atomic model is described in an extended BNF format as:

```
interface model_name  [:parent_model_name]
      input : {…}
      output : {…}
end model_name;
atomic model model_name [:parent_model_name]
      state variables :[var_name in 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;
```

In DEVSIF, a coupled model is described as:

```
interface model_name  [:parent_model_name]
      input : {…}
      output : {…}
end model_name;
coupled model model_name [:parent_model_name]
      component : {[child_name in model_name;]+}
      external input coupling :
          {[model_name.input_event->
          child_name.child_input_event;]*}
      external output coupling :
          {[child_name.child_output_event ->
          model_name.output_event;]*}
      internal coupling :
          {[src_child_name.src_output_event->
          dst_child_name.dst_input_event;]*}
      [select : [{[child_name;]*}]]
end model_name;
```

Since the DEVSIF model should preserve the same semantics as the DEVS model, DEVSIF is considered to satisfy this preservation. *input*, *output*, *state variables*, *internal transition*, *external transition*, *output function*, *time advance* implicate *X*, *S*, *Y*, d_{int}, d_{ext}, *l*, *ta* respectively. Actually, *Initial condition* is defined to simulate the model, and is the

extended attribute of the DEVS formalism. To reduce errors in model specification, the DEVSIF translator employs a strong type-check and ill-structure check, thus supporting stable DEVS model design.

Through the code-generation from RADESIF to DEVSim++ and DEVSim-java, the model is freely combined with a desired simulation environment.

3 A EXAMPLE: CSMA/CD

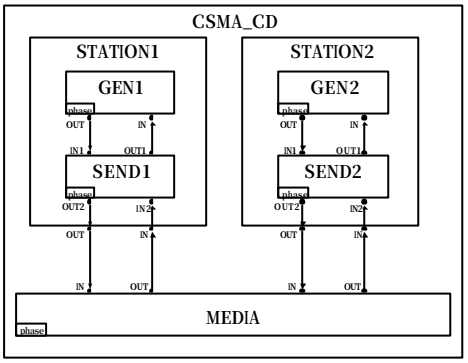

Figure 2: The overall structure of CSMA/CD

This section presents an example to show the complete modeling and simulation process using the proposed framework. The CSMA/CD protocol has the two major facilities for the collision detection and retransmission mechanism. The overall CSMA/CD model consists of the coupled model STATION and the atomic model MEDIA. STATION means the network node, which is connected to the physical network media and has two atomic models GEN and SEND. GEN merely generates data when SEND transmits data successfully or a collision occurs in MEDIA. SEND checks the status of the transmission line by sending an inquiry message to MEDIA. If the transmission media is available, it sends data to MEDIA. SEND goes to the jamming state when it receives a collision message and tries to resend the collided date after back-off time. MEDIA broadcasts its status to STATIONs when it receives an inquiry message. MEDIA broadcasts the collision message to the STATIONs when more than one STATIONs try to send data simultaneously (IEEE std 802.3).

3.1 Modeling of CSMA/CD

The whole CSMA/CD model has too many components to be presented in this paper. So, simply, take a SEND model, which is the core atomic model in CSMA/CD, for the explanation of the complete process. SEND model shows the collision detection and retransmission mechanism in CSMA/CD. SENSE state means the collision detection.

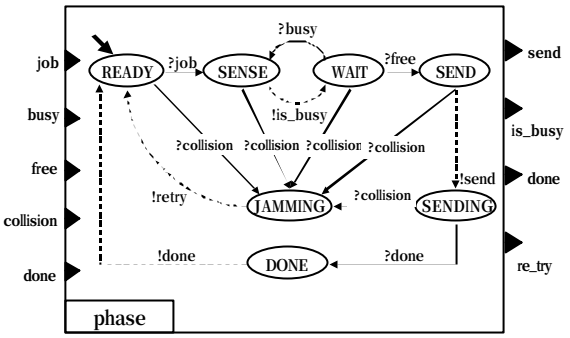

Figure 3: DEVS Model of SEND in CSMA/CD

JAMMING state processes the retransmission requirement. Next, the DEVSIF model is presented for SEND model.

```
interface SEND
     input : {job,busy,free,collision,done}
     output : {send,is_busy,done,re_try}
end SEND
atomic model SEND
     state variables :
         phase in {READY, SENSE, WAIT, SEND,
                   SENDING, DONE, JAMMING};
     initial condition : phase := READY;
     internal transition :
         (phase=SENSE)=>{phase := WAIT;}
         (phase=SEND)=>{phase := SENDING;}
         (phase=JAMMING)=>{phase:=READY;}
         (phase=DONE)=>{phase:=READY;}
     external transition :
         (phase=READY)*job=>{phase:=SENSE;}
                   …
     output function :
         (phase=SENSE)=>is_busy
                   …
     time advance :
         (phase=READY)=>infinity
                   …
end SEND
```

From figure 3, we first construct the atomic model SEND such as the above DEVSIF description. Figure 3 shows the overall structure of the CSMA/CD model for specifying the collision detection mechanism. The output of the DEVSIF translator is shown as: Table1

From Table1, we can generate code for the specific simulation engine such as DEVSim-java. DEVSim-java code-generation is shown as :

```
import DEVSim.*;
class MSEND extends AtomicModel{
     final int SC_READY = 0;
     final int SC_SENSE = 1;
     …
     int phase;
     MSEND() {
```

Table 1: RADESIF of SEND in CSMA/CD

Model Name	Table Name	Attribute Name		
SEND	*X'*	id	event name	type id
		1	job	integer
		1	busy	integer
		1	free	integer
		1	collision	integer
		1	done	integer
	Y'	id	event name	type id
		1	send	integer
		1	is_busy	integer
		1	done	integer
		1	retry	integer
	S'	id	state_name	type id
		1	phase	integer
	l'	id	func	
		1	0	
	d_{int}'	id	func	act_func
		1	3	6
		…		
	d_{ext}'	id	func	*x'* act_func
		1	27	job 30
		…		
	●'	id	func	*y'* act_func
		1	87	is_busy 90
		…		
	ta'	id	func	type ta_val ta_id
		1	111	infinity <NULL> <NULL>
		…		
	func	id	func_id	o p e1 e2
		1	0	ASSIGN 1 2
		1	1	STATE phase <NULL>
		1	2	INTVAL 0 <NULL>
		…		

```
        /* Define input events set */
        addInports(1,"job");
        addInports(1,"busy");
        …
        /* Define output events set */
        addOutports(1,"send");
        addOutports(1,"is_busy");
        …
        /* operation of initial conditiln */
        phase = SC_READY;
    }
    public void extTransfn(StateVars s,
                  double e,Message message) {
        String ev = message.getPort();
        if((phase==SC_READY)
            &&(ev.equals("job")) {
            phase = SC_SENSE;
        } else if
        …
    }
    public void intTransfn(StateVar s) {
        if (phase == SC_SENSE) {
            phase = SC_WAIT;
        } else if
            …
    }
    public void outputfn(StateVars s,
            Messages message) {
        if (phase == SC_SENSE) {
            message.setPortVal("is_busy",
                new Integer(1));
        } else if
            …
    }
    public double timeAdvancefn(StateVars s) {
        if (phase == SC_READY) {
            return Infinity;
        } else if
            …
    }
};
```

The next example shows the coupled model description. The DEVSIF example of STATION coupled model in CSMA/CD is described as:

```
interface STATION
      input : {busy,free,collision,done}
      output : {send,is_busy}
end STATION
coupled model STATION
      component : { CGEN in GEN; CSEND in SEND;}
      external input coupling : {
          STATION.busy->CSEND.busy;
          STATION.free->CSEND.free;
          STATION.collision->CSEND.collision;
          STATION.done->CSEND.done;
      }
      external output coupling : {
          CSEND.send->STATION.send;
          CSEND.is_busy->STATION.is_busy;
      }
      internal coupling : {
          CGEN.job->CSEND.job;
          CSEND.re_try->CGEN.re_try;
          CSEND.done->CGEN.done;
      }
end STATION
```

The RADESIF example and it's generated code, DEVSim-java model, of the above description is the following next:

Table 2: RADESIF of STATION in CSMA/CD

Model Name	Table Name	Attributes				
STATION	*X'*	id	event_name	type_id		
		3	busy	integer		
		3	free	integer		
		3	collision	integer		
		3	done	integer		
	Y'	id	event_name	type_id		
		3	send	integer		
		3	is_busy	integer		
	M'	id	child_id	model_id	child_name	
		3	0	0	CGEN	
		3	1	1	CSEND	
	EIC'	id	model_id	x	child_id	child_x
		3	3	busy	1	busy
		3	3	free	1	free
		3	3	collision	1	collision
		3	3	done	1	done
	EOC'	id	child_id	child_y	model_id	y
		3	1	send	3	send
		3	1	is_busy	3	is_busy
	IC'	id	src_child_id	src_child_y	dst_child_id	dst_child_x
		3	0	job	1	job
		3	1	retry	0	retry
		3	1	done	0	done
	SELECT'	child_name				

```
class MSTATION extends CoupledModel{
      MSTATION() {
          AtomicModel CGEN = new MGEN();
          AtomicModel CSEND = new MSEND();
          /* Define input events set */
          addInports(1,"busy");
          addInports(1,"free");
          …
          /* Define output events set */
          addOutports(1,"send");
          addOutports(1,"is_busy");
          addChildren(CGEN);
          addChildren(CSEND);
          /* EIC , EOC , IC */
          addCoupling(this,"busy",CSEND,"busy");
          …
          addCoupling(CSEND,"send",this,"send");
          …
          addCoupling(CGEN,"job",CSEND,"job");
          …
      }
};
```

3.2 Simulation Result

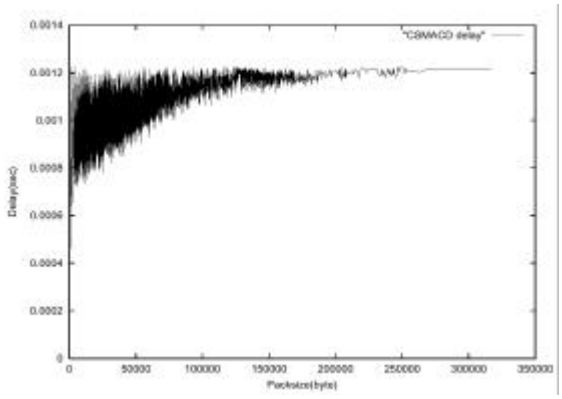

Figure 4: CSMA/CD : Simulation Result for delay

The performance index of CSMA/CD model is delay, which is defined as average time delay per average sending data size (byte). Generally, the delay is an important performance index. The above figure shows such an example of CSMA/CD modeled by the DEVSIF framework.

4 CONCLUSION

This paper proposes the DEVSIF framework of modeling and simulation of discrete event systems. Within the framework, DEVS models are converted into relational algebraic models, each with modeling semantics preserved. The main purpose of the DEVSIF methodology is to directly reuse of DEVS models in various simulation environments. This methodology exploits the DEVSIF language and its associated translator. A set of tools are developed for translation of DEVS models in a database and automatic generation of simulation models from the database. An example of performance evaluation of a CSMA/CD model within the proposed framework shows effectiveness of the framework

REFERENCES

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

Tag Gon Kim, 1997, *DEVSim++ User's Manual: C++ Based Simulation with Hierarchical, Modular DEVS Models*, ftp://sim.kaist.ac.kr/pub/DEVSim++-1.0/, Systems Modeling Simulation Lab., KAIST, Taejon, Korea.

Silberschatz, A., Henry F. Korth, and S. Sudarshan, 1997, *Database System Concepts*, McGraw-Hill Companies, Inc.

Zeigler, B. P. 1984, *Multifacetted Modeling and Dicrete Event Simulation*, Orlando, FL, Academic Press.

Muchinick, Steven S. 1997, *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers, Inc.

Thomas, C., H. Luckhoff and T.G. Kim, 1996, "OpenDEVS: A Proposal for a Standarized DEVS Model Exchange Format", *Proc. Of AIS'96* , pp.371-7.

IEEE standard 802.3, 1988, *Supplements to Carrier Sense Multiple Access with Collision Detection*, Institute of Electrical and Electronics Engineers Inc.

AUTHOR BIOGRAPHIES

Ki Jung Hong received the BS and MS degrees in Korea Advanced Institute of Science and Technology, Taejon, Korea in 1995 and 1997 respectively. His research interests are the real-time system design, the rapid prototyping with co-design framework, and the internet route arbiter.

Tag Gon Kim received the BSEE and MSEE degrees from Pusan National University, Korea,