The DEVS Formalism: Hierarchical Modular Systems Specification in C++

Tag Gon Kim and Sung Bong Park
Dept of Electrical Engineering
Korea Advanced Institute of Science and Technology
Daejon 305-701, Korea

ABSTRACT

Zeigler's DEVS (Discrete Event Systems Specification) formalism supports specification of discrete event models in a hierarchical, modular manner. The formalism makes it possible to design new simulation languages/environments with better understood and sounder semantics. This paper describes a realization of the DEVS formalism in C++, called DEVSIM++ running on a Sparc-1. An object-oriented modeling and simulation environment, DEVSIM++ exploits sound semantics from the DEVS formalism and object-oriented expressive power and execution speed from C++. Thus, DEVSIM++ supports the development of discrete event models within the DEVS framework and simulating them in C++.

1. INTRODUCTION

The compatibility between object-oriented programming paradigms and discrete event world view formalisms has been well noted [O'Keefe 1986]. In such programming systems, an object, an instance of class, is a package of a high-level data structure and associated operations. Since an object can represent a real world counter part, correspondence between an object in the programming system and a model component in the real world can be well established.

Zeigler's DEVS (Discrete Event Systems Specification) formalism [Zeigler 1984] supports specification of discrete event models in a hierarchical, modular manner. The DEVS formalism makes it possible to design new simulation languages/environments with better understood and sounder semantics. Moreover, semantics in the formalism is highly compatible with object-oriented specification of simulation models. In fact, realizations of the DEVS formalism in LISP-based object-oriented environments are already in place. Two such realizations are DEVS-Scheme [Kim 1990] implemented in Scheme (a LISP dialect) and DEVS-CLOS [Farrow 1991] implemented in Common LISP. While the LISP-based realizations provided powerful expressive means for modeling, simulation execution time, bounded to the speed of the underlying language of LISP, was not fast enough for practical applications. Thus, a new realization with a language exploiting trade-offs between expressive power and simulation performance was motivated. One such language can be C++ [Stroustrup 1986].

This paper describes a realization of the DEVS formalism in C++ called DEVSIM++. An object-oriented modeling and simulation environment, DEVSIM++ exploits a sound semantics from the DEVS formalism and an object-oriented expressive power and execution speed from C++. Thus, DEVSIM++ supports the development of discrete event models using C++ within the DEVS framework. This paper is organized as follows. Section 2 presents a brief review of the DEVS formalism. Sections 3 presents outlines of the DEVSIM++ realization including the class organization. Examples are given in section 4 and summary in section 5.

2. THE DEVS FORMALISM IN BRIEF

A set-theoretic formalism, the DEVS formalism [Zeigler 1984][Concepcion 1988], specifies discrete event models in a hierarchical, 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 hierarchical fashion. A basic model, called an *atomic model* (or *atomic DEVS*), has specification for dynamics of the model. An atomic model M is specified as:

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

X : input events set;
S : sequential states set;
Y: output events set;
$\delta_{int} : S \rightarrow S$: internal transition function;
$\delta_{ext} : Q \times X \rightarrow S$: external transition function;
$Q = \{ (s, e) \mid s \in S, 0 \le e \le ta(s) \}$: tatal state of M;
$\lambda : S \rightarrow Y$: output function;
ta: S $\rightarrow$ Real : time advanced function.

The second form of the model, called a *coupled model* (or *coupled DEVS*), tells how to couple (connect) several component models together to form a new model. This latter model can itself be employed as a component in a larger coupled model thus giving rise to construction of complex models in hierarchical fashion. A coupled model DN is defined as:

$$DN = < D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, SELECT >$$

D : component names set;
for each i in D,
  M_i : DEVS for component i in D;
  I_i : set of influencees of i ;
for each j in I_i,
  $Z_{i,j} : Y_i \rightarrow X_j$: i-to-j output translation function;
SELECT: subsets of D $\rightarrow$ D : tie-breaking selector.

Detail descriptions for the definitions of the atomic and coupled DEVS can be found in [Zeigler 1984].

The *abstract simulator* introduced in [Zeigler 1984] is a conceptual device capable of interpreting the dynamics of DEVS models. Two classes of the abstract simulator have been defined: one for the *atomic DEVS* and the other for the *coupled DEVS*. Within DEVS-Scheme a pair of a model and an abstract simulator is created and association of the two is made for simulation.

The job of an abstract simulator for an atomic model is to schedule the next event time and execute model's transition functions and output function timely as simulation proceeds. The responsibilities of the abstract simulator for a coupled DEVS model is to synchronize the component abstract simulators for scheduling the next event time and to route external event messages to component simulators. The complete algorithms for both abstract simulators and proof for the correctness for the algorithms can be founded in [Zeigler 1984].

3. CLASS ORGANIZATION IN DEVSIM++

DEVSIM++ realizes the DEVS formalism for modeling and associated abstract simulator concepts for simulation, all in C++. For modeling, current DEVSIM++ provides the modeler with facilities for specification of atomic models and coupled models in a digraph form within the DEVS framework. We call such models definition *Atomic_models* and *Coupled_models* in DEVSIM++, respectively. Thus, within the DEVSIM++ environment, the modeler needs to specify transition functions, time advanced functions, and output functions for atomic models and components and their coupling schemes for digraph models. For simulation, DEVSIM++ implements hierarchical scheduling algorithms in abstract simulators of atomic and coupled models defined by the modelers. The approach above is similar to one taken in DEVS-Scheme [Kim 1990].

3.1 Classes for General Facilities

DEVSIM++ employs the NIH class library (NIHCL) [Gorlen 1990]. The class Object, the root class of NIHCL, supports general facilities for operations and queries on object including class name, class description, class comparison, and others. The universal class in DEVSIM++ is the class Entities and defined as a subclass of the class Object. The inheritance mechanism in C++ ensures that general facilities shared for all classes need only be defined in the two classes once and for all. Some facilities in the class Object are class description, comparing objects, and printing objects. The class Entities provides such facilities as constructing objects, destroying objects, and displaying objects in a class. Fig. 1 shows class hierarchy in DEVSIM++.

3.2 Classes Models and Processors

Models and *Processors* in DEVSIM++, the main subclass of entities, provide the basic constructs needed for modelling and simulation. The class Models is further specialized into the major classes Atomic_models and Coupled_models that realize atomic models and coupled models in the DEVS formalism, respectively. The class Processors, a virtual device capable of interpreting models' dynamics, implements the abstract simulator concepts associated with the DEVS formalism. Processors has three specializations: *Simulators* for Atomic_models, *Co_ordinators* for Coupled_models, and *Root_co_ordinators* for handling all needs for simulation.

3.3 Classe Atomic_models

Atomic_models realizes the atomic level of the DEVS model formalism. As shown in Fig. 2, it has instance variables corresponding to each of the parts of the formalism and associated methods operating on them. The instance variables *int_transfn, ext_transfn, outputfn,* and *time_advancefn* realize its internal transition function, external transition function, output function, and time advance function, respectively. While omitting details of the modeler's specification, we shall show an example of atomic model specification in section 4.

3.4 Class Coupled_models

Coupled_models is the realization of coupled models definition in the DEVS formalism which embodies the hierarchical construction of modular models. As shown in Fig. 3, definition of a coupled DEVS model includes specification of its component models (called its children) and the desired communication links called coupling scheme. Accordingly, Coupled_models provides methods to specify component models and their couplings (Such methods will be in section 4.2). Other methods in Coupled_models include *get_children, get_influencees,* and *get_receivers* that can be used for message passings in simulation.

3.5 Class Processors

Simulation of DEVS models developed in DEVSIM++ is managed by means of communication among three subclasses of Processors: Simulators, Co_ordinators, and Root_co_ordinators. Such subclasses realize the abstract simulator principles developed as part of the theory. Simulators and Co_ordinators are assigned to handle Atomic_models and Coupled_models in a one-to-one manner, respectively. Instance variables in Models and Processors records such model-processor pairing. A Root_co_ordinators is linked to the Co_ordinators of the outmost coupled model to manage the overall simulation. Details of pseudo-code for simulation management in each subclass can be found in [Zeigler 1987].

4. MODELS DEVELOPMENT IN DEVSIM++

To explain the development of an atomic model, consider a simple model of a buffer and a cascaded processor shown in Fig. 4 (a). The model is commonly used in computer and/or communication systems. BUF controls flow of incoming problems or packets which are to be sent to PROC. The input port "in" of BUF is for receiving incoming problems and the input port "ready" of BUF receives an acknowledge signal from PROC indicating that PROC is

free. BUF releases problems one by one as PROC is free while PROC holds each problem for some time units and outputs it. At the same time, PROC sends an acknowledge signal to BUF.

4.1 Atomic Model BUF

To specify an atomic model BUF within DEVSIM++, one first needs to create an object BUF of Atomic_models. Then one specifies four components in the DEVS formalism in C++:

external transition function: BUF_ext_transfn(s,e,x);
internal transition function: BUF_int_trnasfn(s);
output function: BUF_outputfn(s);
time advance function: BUF_time_advancefn(s).

For input/output events in input/output sets (x in the external transition function above is an input), DEVSIM++ specifies a pair of port-value for each event in an explicit form of (port, value). Thus, an external event x = (port1, value1) signals the fact that an input port port1 is receiving a value value1. Similarly, an output event y = (port2, value2) indicates sending of the event at an output port port2 with a value value2.

After specifying the functions above, one needs to assign them to the instance variables of Atomic_models BUF as:

```
BUF.set_ext_transfn(BUF_ext_transfn);
BUF.set_int_transfn(BUF_int_transfn);
BUF.set_outputfn(BUF_outputfn);
BUF.set_time_advancefn(BUF_time_advancefn).
```

Fig. 4(b) shows specification of the atomic model BUF. One can develop the atomic model PROC in the same manner as used in BUF.

4.2 Coupled model PEL

Once BUF and PROC are developed, the coupled model PEL can be specified by defining components and coupling scheme as defined in the DEVS formalism. Specification of components employs a method *add_children* of Coupled_models.

Specification of coupling scheme designates three sets of communication links: internal coupling and external couplings consisting of external output and external input coupling. In DEVSIM++, one can specify such coupling scheme using a method *add_coupling* provided by Coupled_models in DEVSIM++. Fig. 4(c) shows specification of the coupled model PEL.

5. SUMMARY

We described a realization of the DEVS formalism in C++. Current status of the realization DEVSIM++ is that it only supports specification of atomic models and coupled models in a directed graph form. However, since DEVSIM++ separates the model specification from simulation algorithms, the class Coupled_models can be easily specialized into subclasses, suited to different application domains, without modifying current simulation algorithm. Detail methodology of such specialization in DEVS-Scheme was discussed in [Kim 1991]. In DEVS-Scheme, the class *coupled-models* of DEVS-Scheme is specialized into the classes *digraph-models* and *kernel-models*, which, in turn, specialized into the classes *broadcast-models, hypercube-models, cellular-models*, and so on. We experienced that DEVSIM++ improved simulation performance markedly without losing much expressive power, compared with DEVS-Scheme. Simulation runs for various discrete event models are underway for testing DEVSIM++.

REFERENCES

[Concepcion 1988] A.I. Concepcion and B.P. Zeigler, "DEVS formalism: A framework for hierarchical model development," *IEEE Trans. on Software Engineering*, Vol. 14, No. 2, pp 228-241, Feb., 1988.

[Farrow 1991] J.M. Farrow and S. Sevinc, "Modelling Tools for A Common LISP Object-oriented System Environment," *Proc in The Second Annual Conf on AI, Simulation and Planning in High Autonomy Systems*, IEEE Press, 1991.

[Gorlen 1990] K.E. Gorden, S.M. Orlow, and P.S. Plexico, *Data Abstraction and Object-oriented Programming in C++*: John Wiley & Sons, New York, NY, 1990.

[Stroustrup 1985] B. Stroustrup, *The C++ Progamming Language*: Addison-Wesley Pub. Co, Reading, MA, 1986.

[Kim 1990] Tag G. Kim and B.P. Zeigler, "The DEVS-Scheme Simulation and Modelling Environment," Chapter 2 in *Knowledge Based Simulation: Methodology and Application* (eds: Paul A. Fishwick and Richard B. Modjeski) Springer Verlag., Inc., pp. 20-35, 1990.

[Kim 1991] Tag G. Kim, "Hierarchical Development of Model Classes in The DEVS-Scheme Simulation Environment," *Expert Systems with Applications*, Vol. 3, No. 3, pp. 343-351, 1991.

[O'keefe 1986] R. O'keefe, "Simulation and Expert Systems-A Taxonomy and Some Examples,"*Simulation* 41:6, pp. 10-16, 1986.

[Zeigler 1984] B.P. Zeigler, *Multifacetted Modeling and Discrete Event Simulation*: Academic Press, Orlando, FL, 1984.

[Zeigler 1987] B.P. Zeigler, "Hierarchical Modular Discrete Event Modeling in Object-oriented Environment," *Simulation*, 1987.

- Object
--- Entities
----- Models
------- Atomic_models
------- Coupled_models
----- Processors
------- Root_co_ordinators
------- Co_ordinators
------- Simulators

Fig. 1. Class Hierarchy in DEVSIM++.

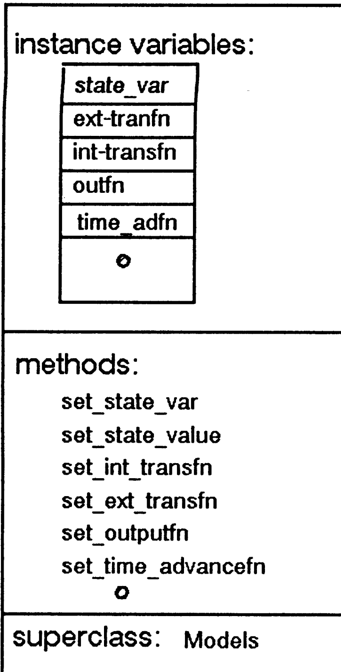

Fig. 2. Class Atomic_models.

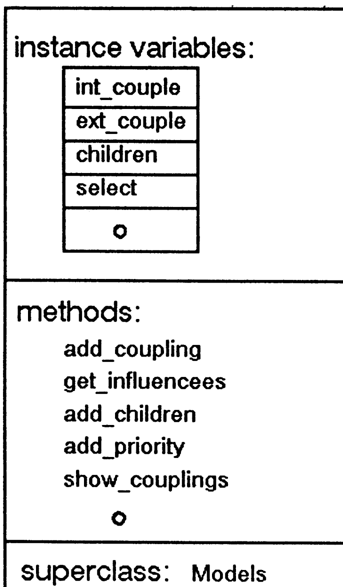

Fig. 3. Class Coupled_models.

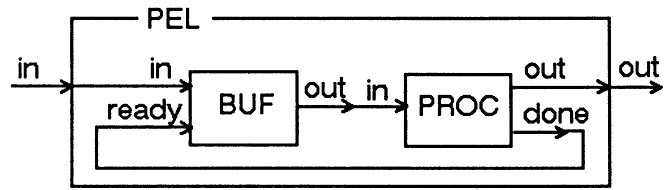

Fig. 4 (a). A Coupled Model PEL.

```
// Atomic_models BUF in DEVSIM++ //
// state vars: proc_free, phase, buffer  //
// input ports: "in", "ready"             //
// output port: "out"                    //

make_pair (Atomic_models, BUF);

 // external trasition function //

void BUF_ext_transfn(State_vars& s, timeType& e,
                         Messages& message)
{
   String& inport = *message.get_port();
   OrderedCltn& buf =
              *(OrderedCltn*)s.get_value("buffer");
   if(inport == "in")
   {
      if ((*(Integer*)s.get_value("proc_free") == true)
         && !buf.isEmpty())
      {
         s.set_value("phase", send);
      }
      Integer* i = new Integer(*(Integer*)
                   message.get_value());
      buf.add(*i);
   }
   else if(inport == "ready")
      s.set_value("proc_free", true);
   else
      exit(-1);
}

// internal transition function //

void BUF_int_transfn(State_vars& s)
{
   if(*(Integer*)s.get_value("phase") == send)
   {
      s.set_value("phase", passivate);
      s.set_value("proc_free", false);
      OrderedCltn& buf =
              *(OrderedCltn*)s.get_value("buffer");
      Integer* i = (Integer*)buf[0];
      buf.remove(*i);
      delete i;
   }
}
```

```
// output function //

void BUF_outputfn(State_vars& s)
{
  if(*(Integer*)s.get_value("phase") == send)
  {
    if(buf.isEmpty())
    {
      cout << "Error: The function should not be
                          called\n";
      exit(-1);
    }
    OrderedCltn& buf =
              *(OrderedCltn*)s.get_value("buffer");
    Integer* i = (Integer*)buf[0];
    message.set_port("out");
    message.set_value(*i);
  }
}

// time advance function //

timeType BUF_time_advancefn(State_vars& s)
{
  OrderedCltn& buf =
            *(OrderedCltn*)s.get_value("buffer");
  if(*(Integer*)s.get_value("phase") == passivate &&
    *(Integer*)s.get_value("proc_free") == true    &&
    !buf.isEmpty())
    return 0;      // delay time for send data to process
  else if(*(Integer*)s.get_value("phase") == send &&
      *(Integer*)s.get_value("proc_free") == true &&
      !buf.isEmpty())
    return hold;         // hold
  else
    return infinity;
}

// assign functions to instance variables //

void assign_BUF_functions()
{
  BUF.set_ext_transfn(BUF_ext_transfn);
  BUF.set_int_transfn(BUF_int_transfn);
  BUF.set_outputfn(BUF_outputfn);
  BUF.set_time_advancefn(BUF_time_advancefn);
}
```

Fig. 4 (b). DEVSIM++ code for Atomic Model BUF.

```
// Coupled_models PEL in DEVSIM++ //
// components: BUF, PROC          //
// input port: "in"               //
// output port: "out"             //

make_pair (Coupled_models, PEL);

// add children and coupling in PEL //

void assign_PEL_comp_coupl()
{
  PEL.add_children(BUF, PROC);
  PEL.add_coupling(PEL, "in", BUF, "in");
  PEL.add_coupling(BUF, "out", PROC, "in");
  PEL.add_coupling(PROC, "done", BUF, "ready");
  PEL.add_coupling(PROC, "out", PEL, "out");
  PEL.add_priority(BUF, PROC);
}
```

Fig. 4 (c). DEVSIM++ code for Coupled Model PEL.