Algorithms for efficient implementations of the DEVS & DSDEVS abstract simulators

Alexander Muzy
University of Corsica
Laboratory SPE CNRS UMR 6134
Campus Grossetti, BP 52
20250 Corti
+33 495 45 02 07
a.muzy@univ-corse.fr

James J. Nutaro
Oak Ridge National Laboratory
+01 865 241 1587
nutarojj@ornl.gov

1. Introduction

The Discrete EVent System Specification (DEVS) is a mathematical formalism for describing discrete event systems. DEVS is grounded in general systems theory. Within DEVS, discrete event systems are described by two kinds of structures. Atomic models describe the behavior of non-decomposable units via event-driven state transition functions. Each atomic model has a clearly defined set of inputs and outputs, and its state space is completely encapsulated. More complex models can be constructed from networks of atomic components. These network models, in turn, have well defined inputs and outputs, and they can be used as components within new network models. This allows for the modular and hierarchical construction of very large discrete event systems.

The hierarchical and modular structure of a DEVS model is reflected in the classical specification of the DEVS simulators [Zeigler et al., 2000]. Each atomic model is associated with a simulator object. The simulator is controlled by sending it messages such as "compute next state" and "compute next output", and it makes requests such as "get time of next event". A coordinator object is associated with each network model, and the coordinator can respond to the same types of messages as the simulator objects. The coordinator, as its name suggests, coordinates the execution of its component coordinators and simulators.

A direct implementation of the classical specification can be inefficient for simulating models that are comprised of numerous interacting subsystems [Muzy et al., 2002]. Memory inefficiencies result from using an excessive number of coordinator and simulator objects. Simulators are needed only for active components, whereas the standard description of the DEVS simulation algorithms calls for a simulator for every atomic model. Similarly, it is possible to replace the plethora of coordinator objects with a single coordinator object. This new coordinator uses recursion to manage the model hierarchy.

Time inefficiencies stem from the use of ensemble methods to determine the next event time and to route events through the model network (see, e.g., [Zeigler 2000]). This can be remedied by the use of more sophisticated event scheduling and event routing algorithms.

The efficiency of DEVS simulator implementations is beginning to receive significant attention [Hu and Zeigler, 2004; Lee and Kim, 2003; Wainer and Giambiasi, 2001]. Concerning cellular models, [Wainer and Giambiasi, 2001] shows that the simulation of cellular models can be improved by "flattening" the hierarchy of coordinator objects. In this flattened simulator, one coordinator manages all of the model's atomic components. This can significantly reduce the cost of event routing, and it eliminates the need for multiple coordinator objects. [Lee and Kim, 2003] describe a similar solution that computes and stores possible event routes at compile time. [Hu and Zeigler, 2004] describe an improved scheduling algorithm for cellular models that are simulated using a hierarchy of coordinators and simulators.

We propose a new simulator implementation for DEVS [Zeigler et al., 2000] and Parallel Dynamic Structure Discrete Event (DSDEVS) models [Barros, 1997]. The simulation architecture and communication protocol have been designed to improve efficiency by:

- eliminating unnecessary simulator and coordinator objects,
- speeding up event scheduling by only storing references to active models,
- eliminating unnecessary internal synchronization messages (i.e., *-messages and d-messages [Zeigler et al., 2000]), and
- avoiding unnecessary event routing messages (i.e., y-messages and x-messages).

The algorithms described in this paper have been implemented in the adevs simulation engine [Nutaro, 1999], and they have been applied successfully to several large scale simulation problems (see, e.g., [Nutaro, 2003], [Jammalamadaka, 2003], [Muzy, 2004]).

2. Background

A DEVS atomic model is described by a structure [Zeigler et al., 2000] $< X, S, Y, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda, ta >$. X is the set of input events, S is the state set, Y is the set of output events, δ_{int} is the internal transition function, δ_{ext} is the external transition function, δ_{conf} is the confluent transition function, λ is the output function, and ta is the time advance function. The transition functions are triggered by events, and they operate on a bag of inputs (denoted by X^b) and the state of the system when an event occurs.

A DSDEVS network is described by a structure [Barros, 1997] $< X_{DSDEVS}, Y_{DSDEVS}, \chi, M_\chi >$. X_{DSDEVS} is the set network of

input events, Y_{DSDEVS} is the set of network output events, χ is the name of a special atomic model called the executive model, and M_χ is the executive model. The executive model is a special atomic model described by $< X_\chi, Y_\chi, S_\chi, \gamma, \Sigma^*, \delta_{ext,\chi}, \delta_{\text{int},\chi}, \delta_{conf,\chi}, ta_\chi, \lambda_\chi >$. X_χ and Y_χ are the input and output sets of the executive model, S_χ is the set of states, $\gamma : S_\chi \rightarrow \Sigma^*$ is the structure function, Σ^* is the set of network structures, and $\delta_{ext,\chi}$, $\delta_{int,\chi}$ and $\delta_{conf,\chi}$ are the executive model's external, internal, and confluent transition functions. ta_χ and λ_χ are the executive model's time advance and output functions. In effect, the state of the executive model describes the network structure. Structure changes can occur when the executive model changes its state.

3. Algorithms

Algorithm 1 describes the Root Coordinator (*Root*) that is used to drive a *Simulator*. The simulation starts by initializing the root coordinator's only child (i.e., the *Simulator*). The *Root* executes the main three functions of the *Simulator* until there are no more active models or the final simulation time t_{end} is reached. The first function, *compute-and-route-output()*, computes the output function of the imminent atomic models at the time of next event, t_n, and then routes any component output events. The second function, *atomic-model-delta-functions()*, computes the state transition function of the atomic models that are active (i.e., imminent or have received input) at t_n. The third function, *executive-delta-functions()*, computes the next states of the executive models that are active at t_n.

```
variables:
 tend   // total simulation time
 child // the Simulator

compute the initializations() function of the child
While (tn of the child < tend)
    compute the compute-and-route-output() function of the child
    compute the atomic-model-delta-functions() function of the child
    compute the network-executive-delta-functions() function of the child
endWhile
```

Algorithm 1. Root

Algorithm 2 describes the main variables used in the *Simulator* to implement the *compute-and-route-output()*, *atomic-model-delta-functions()*, and *network-executive-delta-functions()* functions. A description of each variable is included in the algorithm as a comment. Subscripts are used to denote a variable associated with a specific component model (e.g., D_1 indicates the set of current component models for executive model number 1).

```
Variables:
 parent                    // parent coupled model reference
 q                         // model state
 tn                        // time of next event
 tl                        // time of last event
 scheduler                 // priority queue of pairs (d,tn,d)
 total-imminent-set        // set of imminent executive and atomic models
 receiver-set              // set of final event receivers (executive models and atomic models)
 executive-imminent-set    // set of imminent executive models
 D                         // set of current models
 D'                        // set of next set of models
 D'- D                     // set of new models
 D - D'                    // set of removed models
 D ∩ D'                    // set of models that have migrated from one network model to another
 A                         // set of all atomic models
```

Algorithm 2. Simulator variables

Algorithm 3 presents the function that sets up the initial states, simulation times, and model structures. The function *initializations()* initializes atomic model and network executive model states. Models with non-infinite next event times are then added to the scheduler. A similar function, called *structure-initializations()*, is used to initialize models that are added as the result of structure changes occurring during a simulation run.

```
Function initializations()
   t_l ← 0
   construct initial atomic model set A
   For each d ∈ A Do
      q_d ← q_0,d
      t_n,d ← ta_d(q_d)
      t_l,d ← t_l
      If(t_n,d < ∞) Then
         add (d,t_n,d) to the scheduler
      endIf
   endFor
endFunction initializations

Function structure-initializations()
   For each d ∈ (D'- D) – (D ∩ D') Do
      if d is an atomic model then add d to A
      q_d ← q_0,d
      t_n,d ← ta_d(q_d)+t_l
      t_l,d ← t_l
      If(t_n,d < ∞) Then
         add (d,t_n,d) to the scheduler
      endIf
   endFor
endFunction structure-initializations()
```

Algorithm 3. Simulator: initialisations

The procedure for computing the next states of active atomic components is shown as Algorithm 4. First, all of the models in the *total-imminent-set* are scanned. If the model is an executive model, then it is added to the *executive-imminent-set*. These active executive models can cause a change in the network structure, and so they are evaluated only after the non-network executive model state changes have been computed. Doing this avoids problems with atomic models disappearing due to a structure change while they are still in the active set. If the active model is not a network executive, then its next state is computed immediately. Notice that this algorithm is for simulating models specified with Parallel DEVS, and so the select function of classic DEVS is absent.

```
Function atomic-model-delta-functions()
   For each d in the total-imminent-set Do
      if d is an executive model then add d to the executive-imminent-set
      else compute_delta_functions(d)
   endFor
endFunction atomicModelDeltaFunctions()
```

Algorithm 4. Simulator: Transition functions of atomic models

Algorithm 5 describes the function *compute-and-route-output()*. It begins by selecting the time $t_{n,d}$ of the first imminent model in the *scheduler*. This model is removed from the scheduler and added to the *total-imminent-set*. Next, all models having the same next event time are added to the *total-imminent-set* and removed from the *scheduler*. Then the output function of every model of the *total-imminent-set* is computed. The result is stored in the model's output bag y^b. The contents of the output bags are routed and stored in one or more destination input bags x^b. The event routing algorithm is realized by the *route()* function, which is described in the next section. Finally, models that receive input events are added to the *receiver-set* and *total-imminent-set*.

Execution of state transition functions for models in the *total-imminent-set* is presented in Algorithm 6. If the input bag x^b for a model is not empty and that model is scheduled for an internal transition, then the confluent transition function $\delta_{conf,d}()$ is computed. If the input bag x^b for a model is empty and that model is scheduled for an internal transition, then the internal transition function $\delta_{int,d}()$ is computed. Otherwise, the elapsed time e_d is found and the external transition function $\delta_{ext,d}()$ is computed. Finally, the last transition time $t_{l,d}$ is updated. If the next transition time $t_{n,d}$ is less than infinity, then the model reference and its next event time $t_{n,d}$ are added to the *scheduler*.

Since state changes for executive models can result in structural changes, they must be handled differently. Computation of the state transition functions for network executive models is showed in Algorithm 7. Prior to computing the next state of a

network executive, its current component set is recorded. After the state change, the new component set is recorded. Models that have been added are placed in the new model set. Models that have been removed are placed into the removed model set. After all of the active network executives have been processed, the new models are initialised and removed models are deleted. Every model that is in the removed model set and not in the new model set is erased from the *scheduler* and deleted. Every model that is in the new model set but not in the removed model set is initialised and scheduled via the *structure-initializations()* function.

```
Function compute-and-route-output()
   t_n ← min{ t_n,d / d ∈ A }
   While(t_n,d(scheduler) == t_n)
      d ← first(scheduler)
      add d to the total-imminent-set
      remove (d,t_n,d) from the scheduler
   endWhile
   For each d in the total-imminent-set Do
      y^b ← λ_d(q_d)
      For each output event ev_j of y^b Do
         route(parent of d, d, ev_j)
      endFor
   endFor
   total-imminent-set ← total-imminent-set ∪ receiver-set
endFunction compute-and-route-output()
```

Algorithm 5. Simulator: compute and route events

```
Function compute_delta_functions(d)
   If( x_d^b ≠ ∅ and t_n,d = t_n) Then
      q'_d ← δ_conf,d( x_d^b )
   Else If( x_d^b = ∅ and t_n,d = t_n) Then
      q'_d ← δ_int,d(q_d)
   Else
      e_d ← t_n − t_l,d
      q'_d ← δ_ext,d (q_d, e_d, x_d^b )
   endIf
   t_l,d ← t_n
   If(t_n,d < ∞) Then
      insert (d,t_n,d) in the scheduler
   endIF
endFunction compute_delta_functions()
```

Algorithm 6. Simulator: compute delta functionss

```
Function network-executive-delta-functions()
   D ← ∅
   D′ ← ∅
   For each n of the executive-imminent-set Do
      D ← D ∪ set of components for n
      compute_delta_functions(n)
      D′ ← D′ ∪ set of components for n
   endFor
   remove all d ∈ (D − D′) − (D ∩ D′) from the scheduler and atomic model set A
   structure-initializations()
endFunction network-executive-delta-functions()
```

Algorithm 7. Simulator: transition functions of network executive models

The performance of these algorithms in practice depends critically on the implementation of the supporting data structures. The essential data structures and their operations are

- set insertion, removal, containment test, and iteration, and
- inserting, removing, and rescheduling items in the scheduler.

The set implementation used in adevs is a dynamic array that is backed by a hashtable for retrieval of specific items. Elements are inserted into a set by adding them to the end of the array and entering them into the hashtable. Items are removed by deleting them from the hashtable and removing them from the array. Holes in the array are filled by moving the item at the end of the array to the recently vacated position. Iteration is done by looking at each array index in order. The scheduler in adevs is implemented using an array-based binary heap. Each atomic model has an extra integer that records its location in the heap array, and this integer is used to support fast rescheduling. The data structures themselves have been chosen for their time complexity [Weiss 1993]. The C++ implementations have been carefully optimized ([Abrash 1997] provides an excellent series of essays on practical code optimization).

4. Recursive event routing

Event routing is achieved without a hierarchy of coordinators by employing recursion. Every model includes a reference to its parent model, except for the model at the top of the hierarchy. Every executive model contains a routing method that describes how events are routed within a single network model.

Event routing begins when a model (atomic or network) generates an output, or when an input arrives to a network model. The immediate destinations for the event are determined using the network executive model's route method, and they are stored in a temporary-receiver-set. The routing procedure is applied recursively to each model in the temporary-receiver-set in the following way. If a destination is another atomic model within the same network, then the routing is finished. If a destination is a network model within the same network, then the routing function is repeated using the executive of that network model. If a destination is an output port of the current network model, then the routing function is repeated using the executive model of the current model's parent.

```
function route(parent,src,ev)
   compute the temporary-receiver-set for event ev using the parent model
   For each model m in the temporary-receiver-set
      if m is an atomic model
         add ev to the input bag x^b for model m
         add m to the receiver-set
      else if m = parent
         route(parent of m,m,ev)
      else
         route(m,m,ev);
      end
   endFor
endFunction route()
```

Algorithm 8. Recursive event routing

To illustrate this recursive procedure, an example of a hierarchical model is presented in Figure 2. A coupled model *CM_2* contains two coupled models *CM_1* and *CM_3*. The coupled model *CM_1* contains an atomic model *AM_1*. The coupled model *CM_3* contains an atomic model *AM_2*. The atomic model *AM_1* is coupled to the atomic model *AM_2* through the three coupled models *CM_1*, *CM_2* and *CM_3*.

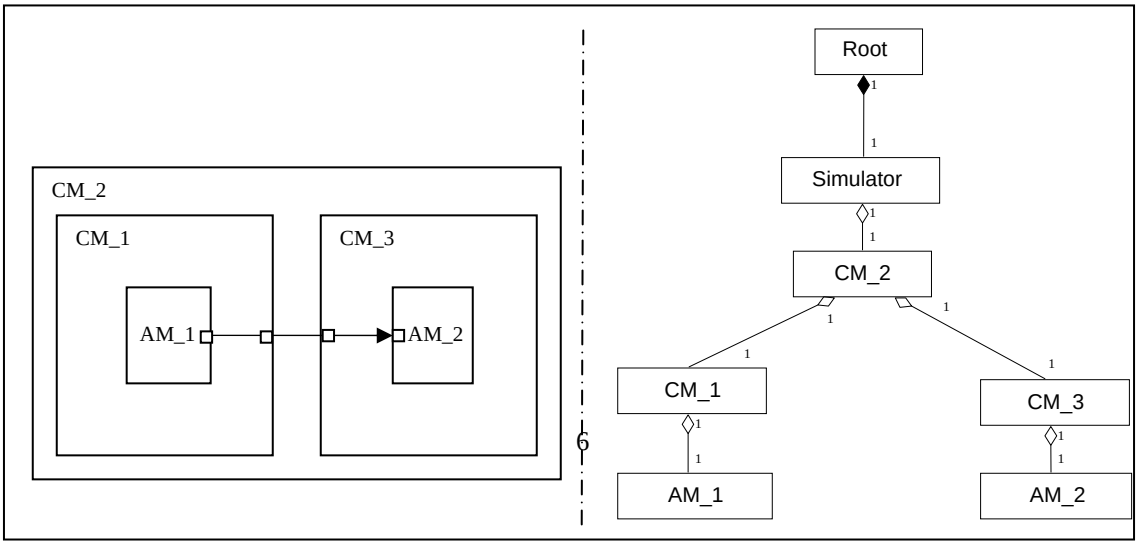

Figure 2. Models and associated simulator

Figure 3 depicts the event route that is computed using the recursive routing procedure. Here, the atomic model *AM_1* produces an output that should be consumed by model *AM_2*. Denoting the output of model *AM_1* by *ev*, the *route()* function is called recursively as follows: (1) route(*CM_1*,*AM_1*,*ev*), (2) route(*CM_2*,*CM_1*,*ev*), and (3) route(*CM_3*,*CM_3*,*ev*). The last call discovers the receiving model *AM_2*. The event *ev* is then added to *AM_2*'s input bag and *AM_2* is added to the *receiver-set*.

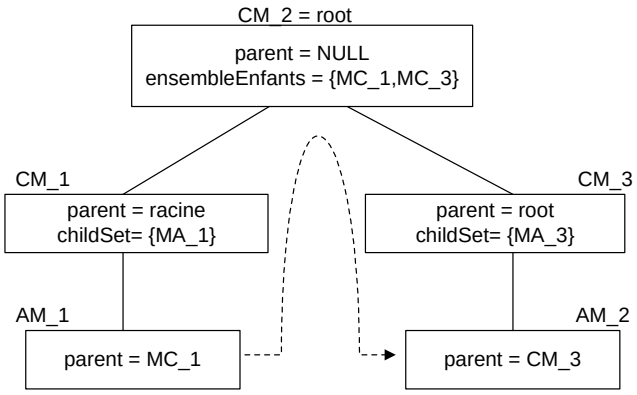

Figure 3. Implicit link tree

5. Conclusion

The simulation algorithms described here have been implemented in the adevs simulation library [Nutaro, 1999]. The performance of the simulation library has to be improved over its lifetime in three different ways. The first, and most significant, is via the application of the algorithms described here.

Secondly, the underlying data structures, and the set and scheduler implementations in particular, have been optimized for use in a simulation application. The use of dynamic arrays in particular takes advantage of the fact that most simulation applications contain a fixed set of models. While dynamic structure modeling is supported, structure changes are relatively rare.

Thirdly, the liberal use of a performance profiling tool allowed for targeted optimization of the data structure implementations. One example of this is the choice of a closed hashing scheme for the hashtable implementation (see [Weiss 1993]). This choice was dictated by performance data obtained in a comparative study of different hashing schemes to solve some benchmark simulation problems.

The optimization of discrete event simulation software requires a balance between performance and ease of use. The design and implementation of adevs has evolved to support the best performance possible without exposing optimization tricks to the end user. One example of this is the computationally expensive use of a set difference operation to detect changes in a network's component set. An alternative implementation could allow for explicit add and remove operations within the executive model's transition functions. However, this would damage the abstraction of structure change being intrinsic to an executive model's change of state.

REFERENCES

Abrash, M., *Graphics Programming Black Book Special Edition*, The Coriolis Group Inc., 1997.

Barros, F. J., "Modelling Formalisms for Dynamic Structure Systems". *ACM Transactions on Modelling and Computer Simulation*, v. 7, 1997, p. 501-515.

Hu, X., and B. P. Zeigler, "A high performance simulation engine for large-scale cellular DEVS models". *High Performance Computing Symposium (HPC'04), Advanced Simulation Technologies Conference (ASTC)*, 2004, p. 3-8.

Jammalamadaka, R., Activity characterization of spatial models: application to discrete event solution of partial differential equations: Computer science Master thesis, University of Arizona, Tucson, 2003.

Lee, W. B., and T. G. Kim, "Simulation speedup for DEVS models by composition-based compilation". *Summer Computer Simulation Conference*, 2003, p. 395-400.

Muzy, A., Elaboration of deterministic models for the simulation of complex spatial systems: Application to forest fire propagation [In french]: Computer science PhD thesis, University of Corsica, Corti, 2004.

Muzy, A., G. Wainer, E. Innocenti, A. Aïello, and J. F. Santucci, "Comparing simulation methods for fire spreading across a fuel bed". *AIS 2002 - Simulation and planning in high autonomy systems conference*, 2002, p. 219-224.

Nutaro, J., adevs (A Discrete EVent System simulator), Tucson, Arizona Center for Integrative Modeling & Simulation (ACIMS), University of Arizona, [http://www.ece.arizona.edu/~nutaro/index.php].
Nutaro, J., Parallel discrete event simulation with applications to continuous systems: Computer science PhD thesis, University of Arizona, Tucson, 2003.
Wainer, G., and N. Giambiasi, "Application of the Cell-DEVS paradigm for cell spaces modeling and simulation". *Simulation*, v. 76, 2001, p. 22-39.
Weiss, M., *Data Structures and Algorithm Analysis in C*, The Benjamin/Cummings Publishing Company Inc., 1993.
Zeigler, B. P., H. Praehofer, and T. G. Kim, *Theory of modelling and simulation*, Academic Press, 2000.