A Faster Implementation of DEVS in the Joint MEASURE Simulation Environment

Steven B. Hall, Shankar M. Venkatesan, Donald B. Wood¹, Advanced Simulation Center, Lockheed Martin Space Systems, Sunnyvale, CA 94088

Abstract: The University of Arizona definition and implementation of the DEVS framework is well known in the community of researchers that work on DEVS [1,2,6]. Not only does it provide an Object-Oriented implementation in C++ (and Java), but it also has a tight HLA connectivity (which was replaced by the HLA Interface developed at Lockheed Martin [4,5]). We present a faster and more efficient implementation of the University of Arizona DEVS here, which improves and clarifies many features of their implementation.

1. Our Results

The University of Arizona DEVS and DEVSHLA implementations have served as the simulation engine on which the discrete event simulation framework called Joint MEASURE (JM for short) was developed at Lockheed Martin. In the process of building many critical and large scale simulations based on Joint MEASURE over the past few years, we realized that a significant fraction of the time inefficiency in our runs was coming from the actual design of the DEVS implementation. This is especially so when, during the simulation, the number of models that change state at any given instant was a small fraction of the total number of models. We made it a goal to look into this DEVS implementation, and improve the situation at the first opportunity. This paper describes the successful culmination of this effort.

In this work, we present the following results that: (a) enhance the role of the external inports and outports in the models by providing the ports with real data transfer roles (b) implement a templated priority queue which serves as a repository of models keyed by various single fields (e.g. their “imminent” times or names) or by a totally ordered sequence of fields (c) precompute the mail destinations (and origins) for each port for saving time and selectively update these precomputed Destination and Origin sets when links, ports, or models are dynamically added to or deleted from the simulation (d) eliminate inefficient classes for data structures and replace them with typesafe templated classes and (e) provide data that compares the performance of the old and the new implementations.

2. DEVS and the Previous Implementation

The Discrete Event Systems (DEVS) formalism [1,2] succinctly captures time interactions between hierarchically interconnected automata or finite state machines. Therefore any time-efficient implementation of DEVS is highly suitable as the engine behind the modeling and simulation of a hierarchically specified discrete system. Many important problems in science, technology, and engineering can be modeled in such a fashion, and would benefit from an attempt to improve the efficiency and clarify the implementation.

The previous implementation of DEVS that we used as the basis of the Joint MEASURE concept worked along the following well known general lines: The simulation models are represented in a model tree, where the leaves represented Atomic Models and the interior nodes represented Coupled Models. The models
have *imports* through which input is provided, which immediately exercises the *delta-external function*. They also have *outports* that hold the aggregated output of the model, which will be drained when the *output function* of the model is exercised. A model is said to become *imminent* when a certain sleep (determined dynamically by the model) has been completed: this results in the execution of the *delta-internal function*.

The role of the model tree and the couplings between the models in the tree can be described as follows: Coupled models in turn contain other models (coupled or atomic) and this defines the natural parent-child relationship in the tree among the models. Atomic models, as their names imply, do not contain other models. An outport of a model can be connected to any subset of the inports of its siblings in the model tree, or to an outport of its parent coupled model. An inport of a model can also be connected to from an inport of its parent. This scheme allows the hierarchical transport of messages across the entire tree using simple traversal algorithms, giving rise to an overall implementation that works as follows:

(a) Do a recursive traversal of the model tree from the top, and execute the output function of all the models that are imminent (i.e. at the end of their sleep) on the way down, and collect and propagate the output messages to either siblings or parents on the way up.

(b) Do a second recursive traversal of the model tree from the top, and this time collect and propagate the messages (from step (a)) on the way down the tree, and deliver them to the leaf atomic models; when an atomic model receives a set of messages on any subset of its inports, its delta-external and delta-internal function are immediately executed (in a prechosen order) thereby draining these inports.

These algorithms are simple depth first traversals of the model tree taking time proportional to the size of the tree. This would be a very time-inefficient way to advance to next time step, if only a small fraction of the models in the tree are imminent at any given time. A second factor contributing to the inefficiency was that the model imminent times were detected by a separate full traversal of the tree (and not by keeping these imminent times in a priority queue as they should be). A third contributing factor was that the previous implementation did not use the ports for mail transport, which was done by the models by consulting their parents’ digraph coupling structure during the traversal. The problems were somewhat alleviated by keeping a certain *time-granularity*, and all models with imminent times that fall in this small time band were simultaneously executed (albeit not in exact sorted time sequence because of their depth first ordering).

When we identified the source of the inefficiencies in the previous implementation, we realized that we couldn’t fully correct the situation simply by keeping the model imminent times in a priority queue. The reason was that even a single message was delivered hierarchically by a full traversal of the tree taking time proportional to the size of the tree. Since the simulation does not advance to its next time until at least one delta-external/delta-internal function was executed, more had to be done to extract better performance.

3. **Our Current Implementation**

We started by providing the external *imports* and *outports* with real data transfer roles. This involved keeping adjacency lists for ports and parent model pointers. When an outport was connected, we first calculated all the reachable imports on atomic models (the *leafset*) and on coupled models (the *nonleafset*). When an inport was connected, we calculated similar reverse reachable sets. When an outport p (or a link
or a model) was deleted, we recomputed the reverse leafset for every port in the leafset of p. We similarly recomputed the leafset for every port in the reverse leafset when we deleted an inport p. The advantage of this obviously lies in the fact that these reachability calculations were done only when models (or ports or links) were added or deleted dynamically, and only in the part of the model tree that changed, thus adding to the efficiency of our implementation.

The second thing we did was to implement a general-purpose templated priority queue for storing models by various keys, which could be a name, or an event time, or an imminent time. The particular tree we chose was an AVL tree (we tested this by performing 5 million inserts and deletes that took a minute or so on an UltraSparc60), but any other well-known structure could be used instead. Our DEVS implementation uses this structure to key models by their imminent times: if two models had “nearly” the same imminent times (we had to introduce this concept to avoid errors from machine arithmetic when searching in the tree), we broke the tie using the pointer value (we could have used the model name instead; in order to make the priority queue work, the classes that are used as a template need to have “<” and “>” operators defined). We also had a reverse pointer in the model back to its location in the priority queue to make searches efficient. We also mention in passing that we now use this structure extensively in Joint MEASURE for storing models keyed by names for easy search, and storing simulation events keyed by a totally ordered key sequence (logic for determining entry and exit times for radar field of view will also be implemented using this structure).

In order to propagate messages, we precomputed the mail destinations for each port for saving time. As mentioned before, we dynamically and selectively update these precomputed destination and origin sets when links, ports, or models are added to or deleted from the simulation. Our implementation can then be summarized as follows:

1. Collect all the models that are imminent from the priority queue into a set I
2. Execute the output function of the models in this set, and propagate the output messages directly to the inports in the leafset of the outports that produced the output
3. Collect into a set M all the models that have messages waiting on their inports
4. For all the models in I∩M, execute the delta-internal and delta-external functions in a prechosen order
5. For the remaining imminent models in I, execute the delta-internal function
6. For the remaining models in M, execute the delta-external function
7. After steps 4), 5), and 6), reinsert the model in the priority queue after updating its next imminent time.
8. Repeat steps 1) through 7) until no models are imminent.

4. Performance Measurements on JM

We integrated our new DEVS implementation with the Joint MEASURE simulation environment. Before we outline our performance results, we briefly describe Joint MEASURE²: This is a discrete event simulation environment developed at the Advanced Simulation Center at Lockheed Martin, for the purpose of building and simulating detailed and accurate technical models. It has detailed environment models like terrain, clouds, and electromagnetic transmission analysis that are automatically invoked for route planning, terrain collision avoidance, visibility analysis, and electromagnetic propagation. It has sophisticated radar modeling that parallels the actual physical and logical functioning of the radar. For further details see [3]. Because of the high complexity of Joint MEASURE and the inefficiencies in its code, it may not be the ideal candidate to measure efficiency improvements derivable
from the simulation engine alone. However, it is the most realistic and most complex simulation that we could test on, and here are the results. We tested on scenarios with $n$ platforms where one-third of the models have a radar transmitter/receiver pair on them and the remaining have radar reflectors on them. The simulation time was the same across the board. The run times (in minutes) for the old DEVS Implementation (OLD) and the new DEVS Implementation (NEW) are given below.

<table>
<thead>
<tr>
<th>#models</th>
<th>NEW</th>
<th>OLD</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>2.5</td>
<td>4.5</td>
</tr>
<tr>
<td>90</td>
<td>6.2</td>
<td>12.4</td>
</tr>
<tr>
<td>120</td>
<td>12.2</td>
<td>26.2</td>
</tr>
<tr>
<td>150</td>
<td>22.1</td>
<td>49.2</td>
</tr>
<tr>
<td>180</td>
<td>37.5</td>
<td>97.18</td>
</tr>
</tbody>
</table>

5. Test of Improved DEVS on a Queuing Example

A simple DEVS demonstration program that generates customers and sends them through a queue to two servers was modified to test the new DEVS implementation. Instead of one customer generator, a user selectable number of generators are allowed. Similarly, a user selectable number of servers are allowed. The generator atomic models create a new customer at intervals selected from a normal distribution and send the customer message (note that the customer is not an Atomic model) to the queue atomic model. The queue model sends the earliest customer in the queue to the first idle server in its list of servers. As long as there are idle servers and customers in the queue, it continues sending a customer to a server. If all the servers are busy, the queue model waits for a message from a server that it is now idle, then sends the earliest customer. Each server has a random service time selected from a normal distribution. Besides notifying the queue model that it is idle, the server sends the customer message to a data collection atomic model. When the data collector receives a specified number of customers, it sends stop messages to the generators and queue. The block diagrams of the DEVS model below shows the atomic and digraph models with port couplings.

For this test the program was always run with 10,000 customers and an equal number of generators and servers. The number of generator atomic models was varied from 300 to 2000. The execution times below are from a Sun Blade.

The first chart plots execution time in seconds versus the number of atomic models. The other chart shows the ratio of execution times for the new DEVS
implementation versus the old. The results show substantial improvement.

6. Conclusions and future directions

The new implementation clearly is much faster from the above data, even given the inefficiencies in the Joint MEASURE code. When these additional inefficiencies are removed soon, we expect to see further real time savings from our implementation. As a note of caution, we mention that if a very large percentage of the models are imminent at any given instant, the timesavings would generally be much less, but this should hardly be the case in any realistic simulation.

Already, we have substantially improved our HLA interface to the old DEVS implementation so that DEVSHLA can easily talk to non-DEVS federates, and we are in the process of combining the HLA interface with the new DEVS implementation.

It will be interesting and challenging to develop a Visual Model Interface for our DEVS implementation so that a simulation can be designed with drag and drop support.

7. References
