Proceedings of the 2008 Winter Simulation Conference
S. J. Mason, R. Hill, L. Moench, and O. Rose, eds.

PARTIAL-MODULAR DEVS FOR IMPROVING PERFORMANCE OF CELLULAR SPACE WILDFIRE SPREAD SIMULATION

Yi Sun

Dept. of Computer Science
Georgia State University
34 Peachtree Street, Suite 1450, Atlanta, GA 30303, USA

Xiaolin Hu

Dept. of Computer Science
Georgia State University
34 Peachtree Street, Suite 1450, Atlanta, GA 30303, USA

ABSTRACT

Simulation of wildfire spread remains to be a challenging task. In previous work, a cellular space fire spread simulation model has been developed based on the Discrete Event System Specification (DEVS) formalism. There is a need to improve simulation performance of this model for simulating large scale wildfires. This paper develops a partial-modular implementation of the DEVS-based cellular space model that eliminates the large number of inter-cell message exchanges for improving simulation performance. Both the modular and partial-modular approaches are presented and experiment results are provided. The results show that the partial-modular implementation can significantly improve simulation performance of the cellular space wildfire spread model.

1 INTRODUCTION

Simulation of forest fire spread remains to be a challenging task due to factors such as complex fire behavior, dynamical weather condition, and large spatial area that needs to be modeled. In previous work (Natimo, Hu, and Sun), a discrete event forest fire spread simulation model was developed. This model uses a cellular space to model a forest and each cell corresponds to a sub-area of the forest. Fire spreading is a propagation process that burning cells ignite their unburned neighbor cells. This model is based on the Discrete Event System Specification (DEVS) (Zeigler, Kim, and Praehofer 2000) formalism.

The DEVS formalism is derived from generic dynamic systems theory and provides a formal modeling and simulation (M&S) framework. One of the main features of DEVS-based modeling is that it emphasizes modular (and hierarchical) model construction, where each model is a component with input/output ports and supports well-defined concepts of coupling of components. These couplings allow models to send messages to each other through their input/output ports. The modular model construction of DEVS brings major advantages such as model interoperability and reuse, multi-formalism capability, and dynamic structure change of models. An important type of DEVS model is the cellular space model, which is commonly used to model complex dynamical systems with spatial-temporal behaviors and interactions among their subcomponents. A cellular space model includes a grid of cells where each cell's state can affect and be affected by its neighbors. Formal specifications of DEVS-based cellular space models were also developed. For example, Cell-DEVS (Wainer and Giambiasi 2002) is a specification that extended the DEVS formalism to improve the definition of cellular space models where each cell is defined as an atomic model using transport or inertial delays, and a coupled model that includes a group of these cells constitutes a cellular space model. Gabriel (Wainer 2004) introduced the main characteristics of Cell-DEVS, showing how to model complex cell spaces in an asynchronous environment. Examples of DEVS-based cellular space modeling and simulation include flow injection simulation (Troccoli et al.), traffic control simulation (Davidson and Wainer 2000), forest fire spread simulation (see e.g., Natimo, Hu, and Sun, Wainer 2006), and fire containment simulation (Hu, Muzy, and Ntaimo 2005), just to name a few.

To simulate large scale spatial systems such as forest fire, a cellular space model needs to include a large number of cells. The large number of cells poses a challenge from the simulation performance point of view. In particular, the huge number of inter-cell message exchanges that are typical for large scale cellular space models have a major effect on simulation performance. In a discrete event simulation such as DEVS, each message is an event and triggers a new simulation cycle (also called simulation iteration, see the DEVS simulation protocol in Section 3.4 for more details) for the event handling. The large number of message exchanges (referred to as *message passing* in this paper) thus results in a large number of simulation cycles. Furthermore, in DEVS-based modeling, because of the modular model construction, each cell is an atomic model and cell-to-cell communications can happen only through (indirect) message passing. This slows down the simulation further because of the overhead of message passing and message handling at the model level. Based on the above

observations, this paper exploits the pattern of cell-to-cell message passing to improve simulation performance from two aspects: 1) reduce the number of simulation cycles caused by inter-cell message passing; 2) remove the overhead of message passing between cellular DEVS models. We achieve this goal by turning the modular implementation of DEVS into a *partial-modular* DEVS. The partial-modular DEVS not only removes the overhead of message passing, but also significantly reduces the number of simulation cycles for event handling. We carry out this work based on the specific application of forest fire spread simulation (Natimo, Hu, and Sun). However, the main idea of this approach could be adapted to other DEVS-based cellular space applications.

The remainder of this paper is organized as follows. In section 2, the background and related work is presented. Section 3 presents the modular DEVS and partial-modular DEVS implantations of the forest fire model. Section 4 presents the experiment results and performance analysis. The conclusion and future work are given in section 5.

2 BACKGROUND AND RELATED WORK

The DEVS (Zeigler, Kim, and Praehofer 2000) formalism is derived from generic dynamic systems theory and has been applied to both continuous and discrete phenomena. It provides a formal modeling and simulation (M&S) framework with well-defined concepts of coupling of components, and hierarchical modular model construction. These features of DEVS bring advantages to modeling and simulation such as easy experimentation, easy testing, and easy maintenance. The cellular space DEVS modeling approach divides the spatial space into cells where local computations are done in each cell. A cell is implemented as an atomic DEVS model that performs the local computations internally based on its own state as well as the neighbor states that are received through the external ports. The cell space is implemented as a coupled DEVS model that contains a number of cells. The neighbor rule followed in a specific application determines the couplings between cells. Cellular space DEVS is a special case of conventional DEVS and follows the same structure of DEVS framework, e.g., external, internal and output transition functions. As discussed before, it suffers the problem of performance when a large number of inter-cell communications exist. This performance issue is significant for large cellular space models.

For DEVS-based simulation, different techniques have been researched to improve simulation performance. The Dynamic Structure DEVS (DSDEVS) (Barros 2005) is a specification for dynamic structure modeling based on the DEVS formalism (Zeigler, Kim, and Praehofer 2000). It is shown that dynamic structure modeling can potentially improve simulation performance for large scale cellular space models (Sun and Hu 2006). Dynamical structure modeling changes models' structure (e.g., adding/removing models) and their couplings as a simulation proceeds. It can improve the simulation performance because 1) it reduces the initialization time because it does not load all cells at the beginning of a simulation; 2) it speeds up the execution time of each simulation cycle because it makes a simulation focus only on the "active" cells (non-active cells are either unloaded or removed). On the negative size, dynamic structure modeling brings some overhead. More details can be found in (Sun and Hu 2006). Several efforts developed advanced simulation algorithms for improving simulation performance. Examples of such work can be found in (Muzy and Nutaro 2005; Hu and Zeigler 2004; Wainer and Giambiasi 2001) where the basic DEVS simulation engine was improved to handle messages and cell activity scanning in more efficient manner. Other related works include the quantized DEVS approach (Kofman and Junco 2001, Beltrame and Cellier. 2006), which shows that quantization helps in improving the performance of DEVS-based simulations by reducing the number of state transitions as well as the number of messages while introducing acceptable errors. Efficient implementation of DEVS-based models is also studied for the purpose of improving performance. One such work is reported in (Hall, Venkatesan, and Wood 2003), where the authors enhance the implementation by applying techniques such as pre-computing message destinations, and using a priority queue to sort models to achieve performance improvement for the Joint MEASURE simulation environment. Another technique to improve performance of DEVS models is using non-modular form that combines multiple cells into one for faster simulation. The work (Shiginah and Zeigler 2006) proposed a non-modular formalism using closure under coupling property of DEVS to ensure equivalency of the models to their modular counterparts in parallel DEVS. The speedup was gained through efficient scanning of active cells and combining multiple cells into one atomic model.

Other works use parallel and distributed approaches to improve the performance of DEVS based simulation. Various DEVS-based distributed simulation environments have been developed such as DEVS/CORBA (Zeigler and Sarjoughian 1999), DEVS/RMI (Zhang, Zeigler, and Hammonds 2006), DEVS/HLA (Zeigler, Kim, and Buckley 1999). These distributed DEVS frameworks typically involve a large computation overhead. Some recent techniques (Zacharewicz, Giambiasi, and Frydman 2005, Glinsky and Wainer 2005) are used to reduce the overhead of distributed DEVS techniques.

3 MODULAR AND PARTIAL-MODULAR APPROACHES

This section first gives an overview of the forest fire spread model, then presents the modular and partial-modular im-

plementations of the model, and finally gives an brief performance analysis of the two approaches.

3.1 Overview of the forest fire spread model

In the cellular space forest fire spread model, a forest is modeled as a two-dimensional cell-space composed of individual forest cells coupled together according to their relative physical geometric locations. Each cell represents a sub-area in the forest and is implemented as a DEVS atomic model. A cell is coupled to its eight neighbors corresponding to the N, NE, E, SE, S, SW, W, and NW directions respectively. Accordingly, for each cell, eight fire spreading directions are defined. Fire spreading is modeled as a propagation process as burning cells ignite their unburned neighbor cells. Each cell can be in one of the following six states*: unburn, burning, burned, unburn-wet, burning-wet,* and *burned-wet*, where the *-wet* states model fire suppression and not used in this paper. When a cell is ignited, the maximum fire spread speed and direction of a cell is calculated using Rothermel's semi-empirical model (Rothermel 1972) that takes into account factors such as fuel model, slope, and wind speed and direction. This maximum rate of spread is then decomposed along the eight spreading directions according to an ellipse shape. Figure 1 shows a snapshot of a simulation using real GIS data with 200x200 cells. In the figure, the red cells are burning; the black cells are burned out; all other cells are unburned with the different colors representing different fuel models. More descriptions of this model and the initial conditions of simulation can be found in (Natimo, Hu and Sun).

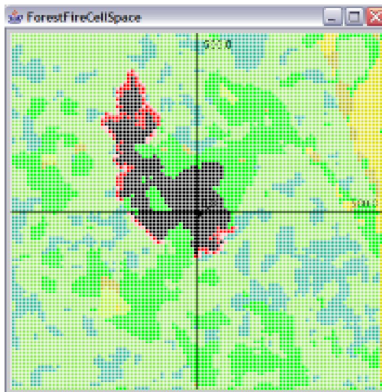


Figure 1: Fire spreading under GIS data

From the above description, one can see that the forest cell space model is composed of a large number of atomic forest cell models. Each cell executes its internal tasks and communicates with other neighboring cells by message passing through inter-connected ports. The process of fire spreading for both the modular and partial-modular implementations is that a cell (referred to as the *source cell*), once ignited, will need to schedule the time for igniting its eight neighbors (refers to as the *destination cells*). In the modular implementation, eight time points are scheduled and kept in the source cell, which sends out igniting messages at the appropriate time based on the schedule. This means for a source cell to ignite its eight neighbors, eight messages are needed (thus eight simulation cycles). The design motivation of the partial-modular implementation is that: the source cell does not keep the time to ignite its neighbors. Instead, it sets the time to its neighbors and asks them (the destination cells) to keep track of their own time-to-burning. This "time setting" happens right after a source cell is ignited and is realized through function call instead of messaging passing (each cell has its eight neighboring cells' object references). The partial-modular implementation brings two advantages from the simulation performance point of view: first, it reduces the message passing overhead between cells; second and more importantly, because a source cell sets all its neighboring cells' time-to-burn in one step (right after the source cell is ignited), it reduces the number of simulation cycles that is needed in the discrete event simulation. These result in simulation performance improvement. It is worthy to point out that the change from modular to partial-modular relies on an implementation that allows a cell to directly modify the state of its neighbor cells (instead of using message passing). Thus it breaks the modular property of DEVS models. Below we describe these two implementations in detail.

3.2 Modular Implementation

The modular implementation of forest fire spread simulation is based on the classic DEVS framework. A forest cell model in forest cell space interacts with its neighbor cells through couplings between cells' input/output ports. A cell affects its eight neighbor cells through eight output ports: *outN, outNE, outE, outSE, outS, outSW, outW*, and *outNW*, which represent eight fire spreading directions corresponding to azimuth (degrees measured clockwise from the north) of 0, 45, 90, 135, 180, 225, 270, and 315 degrees, respectively. Accordingly a cell is affected by its eight neighbor cells through eight input ports: *inN, inNE, inE, inSE, inS, inSW, inW*, and *inNW* (see Natimo, Hu and Sun for more details). Figure 2 shows the structure of message passing from a forest cell to its eight neighbor cells. The dash line in the figure means using message passing to invoke a neighbor cell's external transition functions. The message passing is handled by the DEVS simulation engine, which invokes a destination cell's external transition function at that simulation cycle. In general, messages passing to eight neighbors need up to eight simulation cycles.

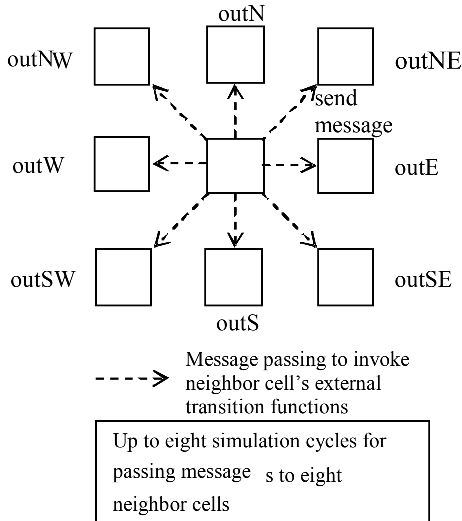


Figure 2: Modular implementation

Below is an informal description of the fire spreading scenario using the cellular space model.

1. Initially a cell is ignited by an igniter atomic model. If its fire line intensity is over the ignition threshold, it begins to burn. Otherwise, it remains unburned.
2. Once a cell is burning, the fire starts to spread to eight neighbor cells as shown in Figure 3.
3. An unburn cell will become burning if it receives a fire ignition message from a neighbor cell (if the fire line intensity is over ignition threshold as mentioned earlier). Similarly, once this cell starts to burn, it begins to spread fire to eight neighbor cells. This process repeats for all cells in the cell space during the whole simulation.
4. If a burning cell receives an input of wind speed and direction, the cell re-calculates its remaining fire spread delays and re-sends spreading messages to the neighbor cells.

Implementation of a forest cell's external, internal transition functions and output functions are listed below. As mentioned before, the modular implementation follows DEVS model's specification and executes the external, internal and output functions for handling external message, internal time event, and generate output. The advantage of the modular implementation is simple structure and easy to implement. However, in large cellular space model, all cells communicate with neighbor cells frequently by a lot of message passing. This increases the computation time from two aspects. One is increased computation for external transition functions. For each cell, it sends out eight messages to its neighbors and receives eight external messages from its neighbors as well. So totally there are 8xN (N is the number of forest cells in the cell space) external transition functions being executed in a complete simulation process. For example, if the cell space size is 100*100, the number of external transition functions being executed is 80000 if the fire spreads to the entire cell space. The other aspect is the large number of simulation cycles in the simulation engine. Since one cell sends out eight fire ignition messages to its neighbor cells at different time, the simulation cycles are large. The larger the number of simulation cycles, the more execution time is needed to run the simulation.

Pseudo code for a cell's external transition function

```
deltext(double e, message x)
  if (receive weather change && state is "burning")
      re-calculate fire spread delay {di} to the eight
      neighbors {ci} i=1,..,8;
      cself.state = burning;
            cself.spread_delay = smallest {di}
 if (receive ignition message from neighbor cells)
      if(cself.state = unburn){
           calculate fire line intensity;
           if (fire line intensity > threshold)
                calculate fire spread delay {di} to
                the eight neighbors {ci} i=1,..,8;
                cself.state = burning;
                cself.spread_delay = smallest {di}
```

Pseudo code for a cell's internal transition function

```
deltint()
   if (state is burning && ! {di} is not empty)
      remove smallest di from {di}
      cself.state = burning;
      cself.spread_delay = next smallest di
   if (state is burning and allNeighborCellBurned)
           cself.state = burned;
```

Pseudo code for devs output function

```
out()
  if (state is burning)
    send message to the corresponding neighbor
```

3.3 Partial-Modular Implementation

The partial-modular implementation updates a cell's state not through message passing. The structure of the partial-modular implementation is shown in Figure 3. In the figure, a cell updates a neighbor cell's state and sigma directly (using function call) and all eight neighbor cells' state (including sigma) updates are accomplished in a single simulation cycle. Compared with the modular imple-

mentation's eight simulation cycles, the partial-modular implementation can reduce the simulation cycles up to eight times.

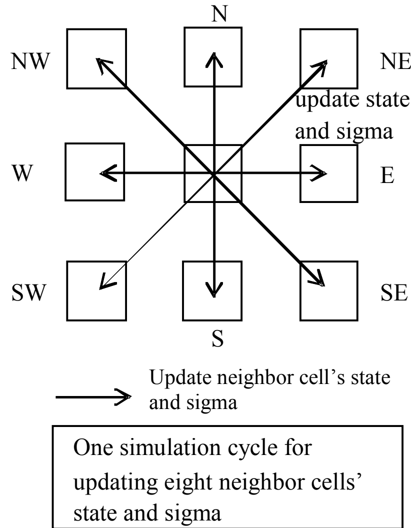


Figure 3: Partial-modular implementation

The fire spreading scenario using the partial-modular implementation is stated as follows.

1. Initially a cell is ignited by an igniter atomic model and the state becomes "schedule_to_burn".
2. When a cell is in "schedule_to_burn" state, it calculates fire line intensity. If the intensity is over threshold, the cell begins to burn. Otherwise, it remains unburned.
3. Once a cell begins to burn, its state becomes "burned". Before that it calculates the fire spread delay to eight neighbor cells and then updates their states to "schedule_to_burn" with the associated sigma being the delay time (referred to as *time_to_burn* afterwards). For each neighbor cell, it uses a variable "ignitionSet" to remember all the cells that want to ignite this cell and the corresponding fire delay for later weather update purpose. Multiple cells may try to ignite the same cell. If that happens, the cell's time_to_burn is updated only when the new calculated spread delay is smaller than the existing time_to_burn. When a cell's time_to_burn is updated, its tN is updated by the simulator correspondingly and the simulator is added to global simulation engine's imminent set. This allows the new updated time_to_burn to be treated properly by the simulation engine. Specifically, in the simulation process, the simulation engine updates its data structure based on the imminents set and gets the smallest tN from them in each simulation cycle. Step 2 and step 3 repeat for all cells in the cell space during the whole simulation process.
4. If a burned cell receives an input of wind speed and direction, and has more than one neighbor cells unburned, it re-calculates the fire spread delays. For each unburned neighbor cell, it compares the new fire delay with all those that want to ignite this neighbor cell (excluding the cell itself) and selects the smallest one as the new *time_to_burn*.

The pseudo codes of the partial-modular implementation's external and internal transition functions are listed below.

Pseudo code for a cell's external transition function

```
deltext(double e, message x)
    if (receive ignition message)
        cself.state = schedule_to_burn
    if (receive weather change)
      if(state is burned && at least one neighbor
       is not ignited)
        re-calculate fire spread delay {di} to the
        eight neighbors {ci} i=1,..,8
        for (each neighbor cell ci && ci.state!=
        burned)
            ci.time_to_burn = exclusive_min(di,
            min(ci.ignitionSet.delay)  //reschedule
            time to burn
            ci.ignitionSet.add(ci, cself)
```

Pseudo code for a cell's internal transition function

```
deltint()
    if (state is schedule_to_burn)
       calculate fire line intensity;
       if (fire line intensity < threshold)
          cself.state = unburned
       else if (fire line intensity >= threshold)
         cself.state = burned
         calculate fire spread delay {di} to the
          eight neighbors {ci} i=1,..,8;
         for (each neighbor cell ci && ci.state  !=
          burned)
            ci.ignitionSet.add(cself , di)
            if(di < ci.time_to_burn)
              ci.time_to_burn = di
              ci.state = schedule_to_burn
              update the simulator by adding ci as
                an imminent;
```

This implementation eliminates the frequent communications between neighbor cells. Compared to the modular implementation, the partial-modular implementation reduces the execution time from two aspects. On one hand, it

reduces the execution time of external transition functions. On the other hand, it reduces the number of simulation cycles in simulation engine. The second aspect is more significant (see section 3.4 for more details). From performance point of view, the major difference lies in that partial-modular implementation eliminates the frequent execution of the external transition functions triggered by inter-cell message passing and thus reduces the number of simulation cycles. A brief comparison and analysis about the execution time is provided next.

3.4 Execution Time Analysis

To understand how the partial-modular implementation reduces the simulation time, it is necessary to look at the simulation protocol of DEVS models. The modular implementation closely follows DEVS models' external, internal and output functions to simulate the forest fire spread process. The partial-modular implementation directly updates a cell's state and sigma, and the corresponding simulator's next event time tN in the simulation engine. Both these two implementations use the DEVS simulation protocol shown below. The simulation engine is a heap-based coordinator.

```
while (tN < predefined_fireSpreadTime){
        imminents.tellAll("computeOutput“,tN)
        imminents.tellAll("sendOutput")
        imminents.tellAll("ApplyDelt“,tN)
        UpdateHeap();
        tN = Heap.getMin();
}
```

In every simulation cycle, the simulation engine asks all *imminents* (whose *tN* = global *tN*) to execute the *computeOutput*, *sendOutput* and *ApplyDelt* functions. At the end of the cycle, the coordinator lets all *imminents* update their newest *tNs* in the heap and get the smallest *tN* for the next simulation cycle.

Based on the above simulation protocol, the execution time of the modular implementation is denoted by formula (1).

$$T = \sum_{i=1}^{N} t_i \quad (1)$$

Where T is the total execution time, N is the number of simulation cycles, t_i is the execution time at every simulation cycle i. t_i includes the time to execute output function, external and internal functions, as well as to find the smallest tN.

The execution time T of the partial-modular implementation is denoted by formula (2).

$$T = \sum_{i=1}^{N'} (t_i') \quad (2)$$

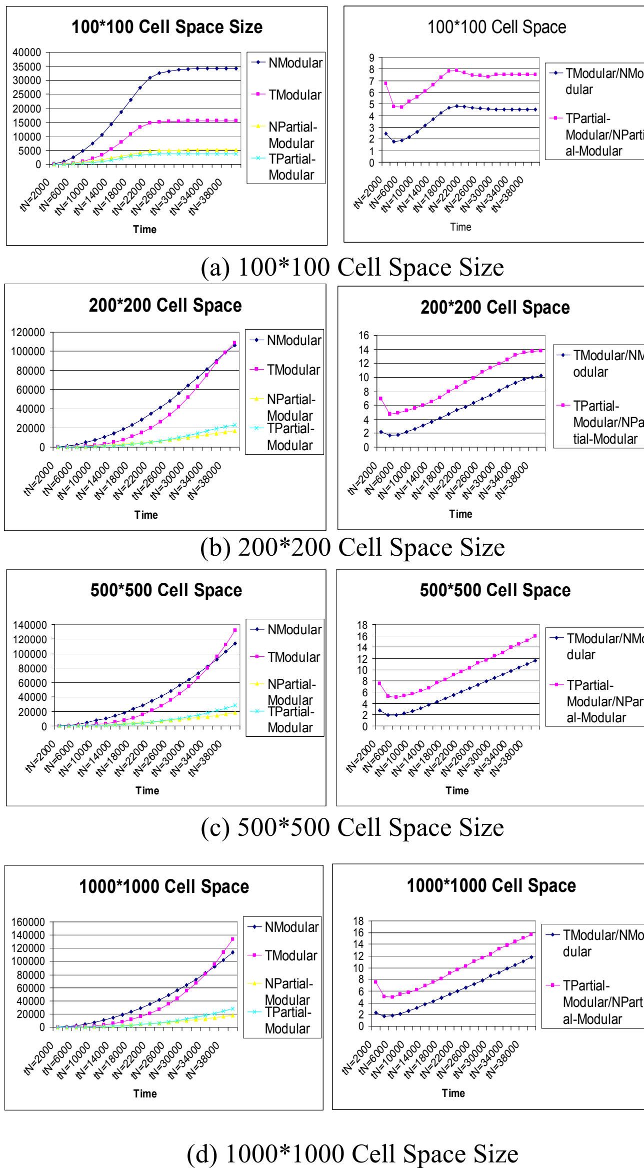
Where N' is the number of simulation cycles in the partial-modular implementation, t_i' is the execution time at each simulation cycle i. Based on the previous analysis, N' is less than N and the ratio of N/N' can reaches 8 (the exact ratio will depend on the specific model behavior). On average, t_i' is larger than t_i. This is because the partial-modular implementation accomplishes fire spread operation in one cycle, which means every simulation cycle involves computation of Rothermel's fire behavior model. However, in the modular implementation, some of the cycles does not need to compute Rothermel's fire behavior model (e.g., when an already ignited cell receives an ignition message). Therefore the partial-modular implementation's execution time in one cycle is larger. But overall the execution time of the partial-modular method is less than the modular method as shown by the experiment results next.

4 EXPERIMENT RESULTS AND ANALYSIS

To compare the simulation performance between the modular and partial-modular approaches, two experiments on forest fire spread model are conducted using different measurements. The simulations were conducted on a Toshiba laptop with Intel Celeron (M) 1.6GHZ processor, 1.2G memory, and Windows XP OS running DEVSJAVA version 3.0. The experiments are based on the forest fire model that uses a dynamically structure implementation (see Natimo, Hu and Sun for more details) and a *Heap* based simulation engine. The same model parameters are used in both experiments. The first experiment is conducted to compare the execution time for different cell space size. The second one compares the execution time for multiple ignitions behavior.

4.1 Execution Time for Different Cell Space Size

Figure 4 shows the total execution time (in milliseconds) for different cellular space models on a 40000 simulation time. Performance results were collected based on every 2000 interval simulation time. The experimental results were measured on different cellular space sizes from 100*100, 200*200, 500*500 to 1000*1000, which are displayed in the figure below. For each cell space size the left diagram displays the execution time T and the number of simulation cycles N for the both approaches. The right diagram displays the ratio of T over N of the two approaches.



(a) 100*100 Cell Space Size

(b) 200*200 Cell Space Size

(c) 500*500 Cell Space Size

(d) 1000*1000 Cell Space Size

Figure 4: Execution time for different cell space size

The results displayed by the left diagrams in Figure 4 show one principle that the execution time has a positive relationship with the number of simulation cycles, i.e., the execution time increases with the increase of simulation cycles. Note in Figure 4(a) 100*100 cell space, the execution time remains almost the same after around 22000 simulation time. This is because all cells in the cell space are in *burned* state after that. The results displayed by the right diagrams in Figure 4 show another principle that the execution time in one simulation cycle (T/N) has the same trend for the two approaches. When the simulation proceeds, the number of fire front cells increases, so the execution time in one simulation cycle increases as well (because there are more cells are involved in the simulation). However the partial-modular approach uses more time per cycle than the modular approach. This is consistent with our discussion before.

The speedup of the partial-modular implementation over the modular implementation is given in Table 1 for the four cellular space sizes respectively. Overall the speedups are significant and are around 4.7 times.

Table 1: Comparison of execution time of modular and partial-modular implementations

T(s)	100*100	200*200	500*500	1000*1000
Modular	154.6	1088.2	1320.5	1337.6
Partial-Modular	38.2	232.2	288.2	282.8
Speedup	4.0	4.7	4.6	4.7

4.2 Execution Time for Fire Spread Simulation With Multiple Ignition Points

The first experiment shows that the number of iterations (simulation cycles) affects the execution time. As the number of iterations increases, the execution time increases too. Another important observation is that the number of ignited cells (fire front cells) affects the average execution time in each iteration. From Figure 4, one can see that for both approaches, initially the number of ignited cells per iteration (T/N) decreases. For example, the initially ignited cell ignites 8 neighbor cells, while later on each cell only ignites about 2 to 3 neighbor cells. At the beginning of the simulation the number of burning cells is small, so the number of ignited cells in each iteration decreases. But as the simulation continues, the number of ignited cells increases again. This is mainly because as the fire front increases, the number of burning cells increases, so the total ignited neighbor cells increases.

Based on these two observations, another experiment with multiple ignitions is conducted to further support the results in the first experiment. In this experiment, instead of using one ignition point to start the fire spread simulation. Forty randomly generated ignition points were used and the experiment results are displayed in Figure 5. The left diagram shows that the execution time increases with the increase of iterations, but the increase rates in the early stage were much larger than those in the late stage. This is because in the early stage, a lot of cells are ignited at the same time, but in the late stage most of the cells are already burned. In the right diagram, from tN=2000 to 4000, each ignition cell ignites neighbor cells from initial 8 to later average 2 to 3. From tN = 4000 the burning area of each ignition cell overlaps to each other, so the average ignited neighbor cells of each cell is less than 2. Therefore the execution time of each iteration in right diagraph increases initially and decreases later on.

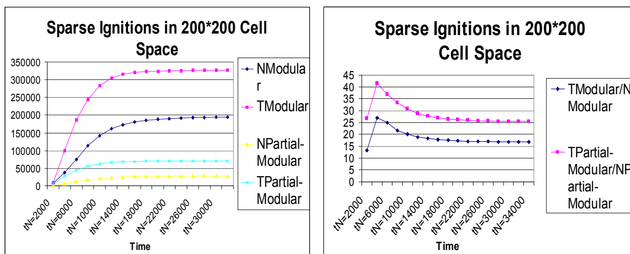


Figure 5: Execution time and iterations for sparse ignitions model

The results in Figure 5 point out the fact again that the number of iterations (simulation cycles) affects the simulation performance. By reducing the number of iterations and then reducing the message passing, even though the execution time in each iteration is increased, the total execution time is reduced.

5 CONCLUSIONS AND FUTURE WORK

The classic modular implementation of DEVS model does not perform well in simulation speed for simulating large cellular space application. In this paper, a partial-modular approach is developed to improve simulation performance by reducing message passing between cells. The reduction of message passing decreases the execution time from two aspects. One is decreasing the overhead time of message passing and message handling in the model. The other is reducing the number of simulation cycles. From the analysis and the experiment results, the number of simulation cycles affects the execution time significantly. For the experiments carried out in this paper, the partial-modular implementation reduces the simulation cycles up to 6-7 times, correspondingly the speed up of execution time gets up to 4-5 times.

The partial-modular approach, although based on the forest fire spread model in this paper, provides an example for improving simulation performance for other types of DEVS-based cellular space models. For the cases that inter-cell messaging passing increases the simulation iterations and thus reduces simulation performance, the approach of partial-modular implementation provides a way for improving performance.

REFERENCES

Barros. F.J. “Modeling Formalisms for Dynamic Structure Systems”. ACM Transactions on Modeling and Computer Simulation, Vol. 7, No. 4, 501-515

Beltrame, T., and F. E. Cellier, "Quantised state system simulation in Dymola/Modelica using the DEVS formalism," in *Proceedings 5th International Modelica Conference,* 2006, pp. 73-82.

Davidson, A., and G. Wainer. “Specifying truck movement in traffic models using Cell-DEVS”. In Proceedings of the 33rd Annual Symposium on Computer Simulation. Washington, D.C. U.S.A. 2000

Glinsky, E. and G.A. Wainer, DEVStone: a Benchmarking Technique for Studying Performance of DEVS Modeling and Simulation Environ-ments, 9-th IEEE International Symposium on Distributed Simulation and Real Time Applications, (Montreal, Canada, 2005).

Hall, S. B., S. M. Venkatesan, and D. B. Wood, “A Faster Implementation of DEVS in the Joint MEASURE Simulation Environment”, in Proc. of Summer Computer Simulation Conference, Montreal, July 2003

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

Hu, X., A. Muzy and L. Ntaimo, A Hybrid Agent-Cellular Space Modeling Approach for Fire Spread and Suppression Simulation, Proceedings of 2005 Winter Simulation Conference, December, 2005

Kofman, E., and S. Junco, "Quantized-state systems: a DEVS Approach for continuous system simulation," *Trans. Soc. Comput. Simul. Int.,* vol. 18, pp. 123-132, 2001.

Muzy A., and J. J. Nutaro, "Algorithms for efficient implementations of the DEVS & DSDEVS abstract simulators," in *1st Open International Conference on Modeling & Simulation (OICMS),* 2005.

Natimo, L., X. Hu, and Y. Sun, DEVS-FIRE: Towards an Integrated Simulation Environment for Surface Wildfire Spread and Containment, submitted to SIMULATION: Transactions of The Society for Modeling and Simulation International.

Rothermel, R., “A mathematical model for predicting fire spread in wildland fuels”. Research Paper INT-115. Ogden, UT: U.S. Department of Agriculture, Forest Service, Intermountain Forest and Range Experiment Station, 1972

Shiginah, F. A., and B. P. Zeigler, "Transforming DEVS to non-modular form for faster cellular space simulation," in *Proceedings of 2006 DEVS Symposium,* 2006, pp. 86-91.

Sun, Y., and X. Hu, Performance Measurement of DEVS Dynamic Structure on Forest Fire Spread Simulation, Proc.14th AI, Simulation and Planning in High Autonomy Systems (AIS 2006), 2006 - 12

Troccoli, A., J. Ameghino, F. Inon, and G. Wainer. “A flow injection model using Cell-DEVS”. In Proceedings of the 35th IEEE/SCS Annual Simulation Symposium. San Diego, CA. U.S.A.

Wainer G., and N. Giambiasi, "Application of the Cell-DEVS Paradigm for Cell Spaces Modeling and Simulation," *Simulation,* vol. 76, pp. 22-39, 2001.

Wainer, G., and N. Giambiasi. “N-Dimensional Cell-

DEVS." *Discrete Events Systems: Theory and Applications* 12, no. 1 (January 2002): 135–157 (Kluwer).

Wainer, G. A.. Modeling and simulation of complex systems with Cell-DEVS, Proceedings of the 36th conference on Winter simulation, 2004.

Wainer, G.. Applying Cell-DEVS Methodology for Modeling the Environment, SIMULATION, Vol. 82, No. 10, 635-660 (2006)

Zacharewicz, G., , N. Giambiasi, and C. Frydman. 2005. Improving the Lookahead Computation in G-DEVS/HLA Environment. Proceedings of the 9th IEEE International Symposium on Distributed Simulation and Real-Time Applications.

Zeigler, B., D. Kim, and S. Buckley. 1999. Distributed supply chain simulation in a DEVS/CORBA execution environment. In *Pro-ceedings of the 1999 Winter Simulation Conference*, Phoenix, AZ.

Zeigler, B., and H. S. Sarjoughian. 1999. Support for hierarchical modular component-based model construction in DEVS/HLA. *Simulation Interoperability Workshop*, Orlando, FL.

Zeigler, B.P., T.G. Kim, and H. Praehofer.: *Theory of Modeling and Simulation.* 2 ed. 2000, New York, NY: Academic Press

Zhang, M., B. Zeigler, and P. Hammonds. 2006. DEVS/RMI – An auto-adaptive and reconfigurable distributed simulation environment for engineering studies. *Spring Simulation Multiconference – DEVS Integrative M&S Symposium*, Huntsville, AL.

AUTHOR BIOGRAPHIES

YI SUN is a Ph.D. candidate in the Computer Science Department at Georgia State University. Her research interests include performance improvement of discrete event systems.

XIAOLIN HU is an assistant professor in the Computer Science Department at Georgia State University. His research interests include modeling and simulation, agents, and simulation-based design.