Performance Measurement of Dynamic Structure DEVS for Large Scale Cellular Space Models

Yi Sun, Xiaolin Hu
Department of Computer Science
Georgia State University, Atlanta, GA 30303
syish@hotmail.com, xhu@cs.gsu.edu

Abstract -- Dynamic Structure DEVS (DSDEVS) is an advanced modeling formalism that allows DEVS models and their couplings to be dynamically changed. The modeling power and advantages of DSDEVS have been well studied. However, the performance aspect of DSDEVS is generally overlooked. This paper provides a comprehensive performance measurement of DSDEVS for a large scale cellular space models. We consider both the modeling layer and simulation layer for performance analysis, and carry out performance measurement based on a token ring model and a fire spread model. The results shows that DS modeling can improve simulation performance for large scale cellular space models, due to the fact that it makes the simulation focus only on those active models, and thus be more efficient than when the entire cellular space is loaded. On the other hand, the DS overhead cannot be ignored and can become significant and even dominant when large number of cells are dynamically added/deleted.

Keywords: DEVS, dynamic structure DEVS (DSDEVS), Large scale cellular space model, Performance measurement, Forest fire spread simulation.

1. Introduction

Dynamic Structure (DS) modeling and simulation refers to the capability of a simulation to dynamically change its model structure as the simulation proceeds. The Dynamic Structure DEVS (DSDEVS) [1] is a specification for dynamic structure modeling based on the DEVS formalism [2]. The capability of DS modeling makes it possible to naturally model complex systems, such as living autonomous systems that change their interactions, compositions and behavior patterns to adapt to their environments, or those self-organizing, self-reconfiguring engineered systems, where the structures of the systems adapt to changed requirements [3]. Many applications have been developed using the concept of DS modeling, including adaptive computer architecture [4], wildfire spread simulation [5, 6], and dynamic team formation of robots [7], to name a few. In general, dynamic structure change can refer to dynamically changing the couplings between existing models or dynamically adding/removing models. This paper mainly concerns the later where models are dynamically added and removed.

The modeling power and advantages of DSDEVS has been well studied especially when it is applied to cellular space models. However, existing work generally overlooked the performance aspect of DSDEVS for large scale cellular space models. Intuitively, one would think dynamically adding/removing (loading/unloading) models during the runtime of a simulation introduces runtime overhead, thus will result in slower simulation speed as compared to a nonDS implementation. On the other hand, by using dynamic structure modeling, a simulation does not need to create all the models in the beginning. Instead, it maintains only a subset of the models by dynamically adding the needed models and removing unneeded models as a simulation proceeds. This allows a simulation to focus its computation power only on those “active” models and results in less memory requirement as well. Here an “active” model means the model that has scheduled next event, i.e., whose next event time is not infinity. This potentially speeds up the simulation performance for simulations with a large number, e.g., up to millions, of models but only a small portion of them are active. The above observations motivate us to measure the performance gains and loss of dynamic structure modeling and simulation.

The idea of focusing on the “active” models for improving simulation performance can be fulfilled in different ways. Dynamic structure modeling achieves this by manipulating the models, i.e., dynamically loading/unloading models so only the active models are maintained in a simulation. Alternatively, advanced simulation algorithms and data structures can be developed to achieve efficient computation focusing on the active models only. For example, a discrete event simulation engine can utilize a heap data structure to keep track of the current *imminent* models (the models with smallest next event time, also referred to as *imminents* in the remainder of the paper), and then asks only those models to go through the simulation cycles. In a discrete event simulation, one can view these two different approaches belong to two different layers of a simulation system: a modeling layer and a simulation layer. Dynamic structure modeling lies in the modeling layer by manipulating the simulation model directly. The heap based simulation engine lies in the simulation layer. It drives the execution of a model but does not modify the model’s structure and behavior. As will be discussed later, the similarities and differences between these two different approaches have important impacts on simulation performance. Each of them has its own gain and loss from the simulation performance point of view. It is the intension of this paper to take account of both the modeling layer and simulation layer in carrying out the performance measurement of dynamic structure DEVS.

Cellular space model represents an important modeling paradigm and is commonly used to model complex dynamical systems with spatial-temporal behaviors (see discussions in e.g., [8, 9]). It supports simulations of various systems, such as urban environment simulation, disease spread simulation, and ecological system simulation. An important feature of large scale cellular space models is that even though a large number of cells exist, only a relatively small portion of them may participate in the simulation at any time. This makes it attractive to apply dynamic structure modeling, i.e., dynamically adding the cells if needed and remove them if not needed, for large scale cellular space simulations. To carry out performance measurement, this paper uses two examples: the first one is a one-dimensional cellular space model of token ring simulation; the second one is a two-dimensional cellular space model of forest fire spread simulation. Both examples are developed in the DEVSJAVA [11] simulation environment. We note that even though the performance results presented in this paper are based on a specific simulation environment, the performance analysis and conclusions drawn from these results are generic and apply to DEVS-based dynamic structure models in general.

The remainder of the paper is organized as follows. In section 2, DSDEVS background and related works are introduced. Section 3 describes the modeling layer and the simulation layer that lead to four different simulation approaches considered in this paper: *nonDS-std*, *nonDS-heap*, *DS-std* and *DS-heap*. Section 4 gives a detailed performance analysis of the four approaches. Section 5 presents performance measurement results based on the token ring model. Section 6 presents performance measurement results based on five measurement metrics for the forest fire spread model. Finally, section 7 discusses and section 8 concludes this work.

2. Related Work

The DEVS (Discrete Event System Specification) [2] 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. The classic DEVS model has been extended to support dynamic structure modeling, where DEVS models and their couplings can be dynamically added/removed as a simulation proceeds. Barros presented a formalism for dynamic structure DEVS whose basic models are classic DEVS models, but the structure of a coupled model can be changed by a network executive model [1]. Uhrmacher proposed a formalism based on DEVS that emphasizes the reflective nature of variable structure models [3]. Dynamic structure modeling has been applied to different applications, such as modeling a complex adaptive computer architecture [4], simulating forest fire spread [5, 6], dynamic team formation of robots [7], and supporting simulation-based design of a DoDAF architecture [12]. An implementation of dynamic structure based on the DEVSJAVA environment was developed in [7] and supports the work in this paper. None of the above works investigated the performance aspect of dynamic structure modeling and simulation. Some preliminary performance results of dynamic structure DEVS were presented in [13].

Performance is an important consideration when simulating large scale cellular space models. The high computational cost of large scale cellular space models has motivated research from both the modeling aspect and simulation aspect for improving simulation performance. On the modeling aspect, the work [6] developed a method to predict whether a cell will possibly change state or will be left unchanged, thus helping a simulation to keep track of the actives cells as a simulation proceeds. In another work [20], the author proposed a non-modular formalism by combining multiple cells into one cell for fast simulation. This formalism uses DEVS' closure under coupling property to ensure equivalency of the models to their modular counterparts. The speedup was gained by efficient scanning of active cells and eliminating inter-cell messages as multiple cells are combined into a single atomic model. Adaptive mesh refinement (AMR) [16] can be considered as another example that dynamically adjust the grid resolution of a mathematic model to achieve computation saving. It improves the performance by assigning high resolutions for resolving developing features, while leaving less interesting parts of the domain at lower resolutions. On the simulation aspect, many different event scheduling algorithms (see, e.g., [33-40]) have been developed in order for a discrete event simulation to focus on the "active" models in an efficient manner. The efficient processing of event is especially needed for large scale simulations that involve large number of events. Lazy queue [36] is a multi-list data structure that divides the events into several parts and keep only a small portion of the near future events sorted. The far future events are unsorted. As time advances, part of the far future is sorted and transferred into the near future. Splay tree [41] uses a balancing technique to move frequently used nodes upwards, thus achieve more efficient search for self-balancing data but it can become worse for a uniform distributed data. In [43] a calendar queue was introduced to hold references to the head and tail of event list. It is suitable for approximately distributed calendar data and the performance is worse for unbalanced data. Another priority queue called twol-amalgamated priority queue [49] uses three efficient Henriksen's queue, skew heap, and splay tree, to achieve efficient operations. The large number of events can also be minimized using techniques that manipulate the statistical properties of the model to reduce the size of events [50].

Parallel discrete event simulation (PDES) [45] is another strategy for performance improvement that reduces execution time by using multiple processors. P-DEVS [46] is a parallel discrete event simulation approach based on the DEVS formalism. It provides means of handling simultaneous scheduled events, while keeping all the major properties of standard DEVS. Since parallel DEVS eliminates serialization constraints, it enables improved execution of models in parallel and distributed environments. The research [47] developed techniques to reduce the overhead of distributed DEVS and get significant improvement in performance. A risk-free optimistic simulation algorithm was presented in [25] to simulate models using shared memory multi-processor machines. In [26], the author investigated hierarchical model partition algorithms that could be used in a distributed/parallel simulation to achieve load balance when simulating complex models.

Performance evaluation and measurement play important roles in system development. It concerns the different aspects of a system performance and provides quantitative and/or qualitative results. According to [51], there are three basic techniques through which performance evaluation can be performed: 1) Analytical modeling: consists in using abstract model based on mathematical notions to describe certain aspects of the system; 2) Simulation: consists in implementing a model that reproduces the system behavior in software; 3) Measurement: consists in fitting the system with specific instruments that allow picking up the relevant values in order to measure the system's performance. The development of performance measurement metrics is system dependent and requires understanding the system and its usage well [52]. Furthermore, the measurement results are typically platform and execution environment dependent. A performance measurement methodology in the field of computer systems engineering was presented in [53]. For DEVS-based simulations, the work [48] developed DEVStone which is a synthetic benchmark devoted to automate the evaluation of DEVS-based simulations. DEVStone facilitates performance analysis for successive versions (e.g., upgrades or fixes) of the same simulation engine, and provides a common metric to compare different M&S environments. In this paper, we focus on performance measurement of dynamic structure DEVS for large scale cellular space models. Both complexity analysis and experiment measurement results are carried out.

3. Dynamic Structure (DS) Model, nonDS Model and their Simulators

As indicated in section 1, the DEVS modeling and simulation framework treats a model and its simulator (the simulation engine) as two distinct components: a model captures the structure/behavior of a system, and a simulator is the algorithm that executes the model. Within this framework, the same model can be simulated by different simulators; similarly, the same simulator can simulate different models [2]. Therefore it is necessary to consider both the model and its simulator when carrying out the performance measurement. In this work, in order to show the performance gains and loss of dynamic structure modeling, for the same cellular space model we develop a nonDS implementation and a DS implementation. They are referred to as the *nonDS* model and the *DS* model in the rest of this paper. Note that the nonDS and DS models share the same model behavior. They differ only in when/how they add or remove cells. To simulate these models, we employ the standard DEVS simulation engine (referred to as *standard coordinator*) and a binary heap-based simulation engine (referred to as the *heap-based coordinator*). Figure 1 shows these two models and two simulators in a layered structure. The nonDS and DS models belong to the modeling layer as they describe the implementations of the cellular space model. The standard and heap-based coordinators belong to the simulation layer as they are in charge of how to execute the models. As mentioned before, the heap-based coordinator is chosen because it implements the idea of focusing on the "active" models. This allows us to compare with the dynamic structure modeling, which also has the effect of focusing on the "active" models when simulation performance is concerned. We note that many advanced heap-based simulation algorithms have been developed over the years. However this paper chooses a basic binary heap-based simulation engine because it is relatively easy to analyze and straightforward for comparing with the dynamic structure modeling.

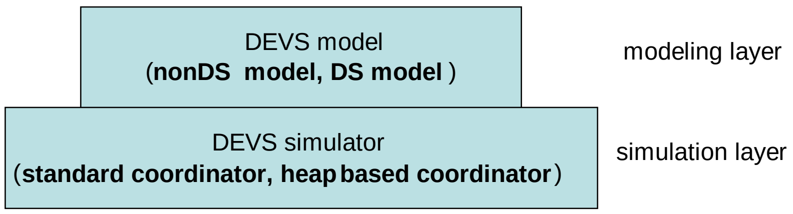

DEVS model
(**nonDS model, DS model**)
modeling layer
DEVS simulator
(**standard coordinator, heapbased coordinator**)
simulation layer

Figure 1. The modeling layer and simulation layer

From Figure 1 one can see there exist four combinations of simulation approaches: simulate a nonDS model using the standard coordinator (referred to as *nonDS-std* later); simulate a DS model using the standard coordinator (referred to as *DS-std* later); simulate a nonDS model using the heap-based coordinator (referred to as *nonDS-heap* later); and simulate a DS model using the heap-based coordinator (referred to as *DS-heap* later). This paper analyzes the performance of all these four approaches but pays more attention to the dynamic structure model when carrying out performance measurement. Next we describe the two models and the two simulators respectively and discuss their performance issues for large scale cellular space models.

3.1 NonDS Model and DS Model

The nonDS and DS models differ in when/how they add or remove cells. Implementation of the nonDS model is straightforward: all cells are added into the cell space and coupled to their neighbor cells when the cell space is constructed. These cells and their couplings do not change throughout the simulation. For example, if a cell space has 1,000,000 cells, all those cells are created and loaded before a simulation starts. This is shown by the following pseudo code.

```
// happen in the beginning of the simulation
for (all cells) {
   create cell mi;
```

```
    addModel (mi);
    addCouplings (mi, mi.outport, mj, mj.inport);
}
```

The DS implementation exploits the fact that cells not participating in the simulation need not to be loaded, thus it starts with only the “active” cells. As the simulation proceeds, other cells are dynamically created and added into the cell space when needed. Meanwhile, when a cell is not needed, that is, after transitioning from active to an inactive state, it is removed from the cell space. As the result, the dynamic structure implementation keeps only the models that are active in the system. To implement the dynamic structure features described above, an atomic model (corresponding to the network executive model in [1]) is needed to manage the dynamic structure changing. This model is responsible for adding and removing the models when needed, e.g., when receiving request inputs from some cells. This is shown by the following pseudo code.

```
// happen in the middle of the simulation
if (need to add cell) {
     create the cell mi;
     addModel(mi);
     addCoupling(mi, mi.outport, mj, mj.inport);
}
if (need to remove cell)
    removeModel(mi);
```

As can be seen in the nonDS model all cells and their couplings are created and added in the beginning before the simulation starts. This results in longer initialization time (the time it takes before a simulation can starts) than the DS model. In the DS model cells are gradually created and added at different steps of the simulation. Meanwhile, the DS model allows the inactive cells to be dynamically removed as the simulation proceeds. It is important to note that for a same cell that is added in the nonDS and DS models, the computation cost is different. Specifically, for the DS model, dynamically adding the cell in the middle of the simulation through a network executive model introduces some overhead. Dynamically removing a cell introduces overhead too because the nonDS model does not invoke the remove operation at all. On the other hand, the DS model adds the cells only needed, thus is likely to create far less cells as compared to the nonDS model because many simulations do not end up with all cells being activated. Meanwhile, by dynamically adding/removing cells, the DS model maintains only a subset of cells in the simulation. This results in some performance advantage when compared to the nonDS model because it makes it more efficient to find the global smallest *tN* due to the relatively small collection that needs to be searched.

3.2 Standard Coordinator and Heap-Based Coordinator

The simulation protocol of the DEVS standard coordinator is described in [2]. In the standard coordinator, a simulation moves forward cyclically based on the time of next event, denoted by *tN*, which is the earliest next event time among all its subcomponents. The pseudo code below describes the major steps in one simulation cycle. Specifically, in every cycle the coordinator first requests each component simulator send its next event time and then finds the minimum of the returned values to obtain the global time of next event: *tN*. After that it asks each simulator to compute the output. The simulators (called *imminents*) whose next event times equal to *tN* invoke the models' output functions to obtain the outputs. Other simulators simply return an empty message. Then the coordinator requests each simulator to send its output to its destination simulators based on models' coupling information. Finally it asks each simulator to apply its *DeltFunc* method that invokes the model's corresponding state transition function based on if external messages are received and/or if internal event time elapses. This ends one simulation cycle and the next cycle repeats.

```
simulators.AskAll("nextTN")
tN = compareAndFindSmallestTN();
simulators.tellAll("computeOutput",tN)
simulators.tellAll("sendOutput")
simulators.tellAll("ApplyDelt",tN)
```

This standard coordinator follows closely the semantic of DEVS models. Thus it is easy to understand and implement. It also serves as a benchmark to test the correctness of other simulation engines. However, the standard coordinator is not efficient when simulating models that have a large number of components. This is because in every simulation cycle, all the simulators, no matter if they are imminent or not, have to go through the simulation steps described above. For a cellular space model that has only a few active models, there exists a lot of unnecessary computation.

The heap-based coordinator overcomes the above problem by using a heap to keep track of the smallest *tNs* of its component simulators. During a simulation, each simulator updates its new *tN* in the heap whenever its *tN* changes. The global smallest *tN* and the *imminents* can be obtained from the root of the heap. Only those *imminents* are asked to go through the simulation cycle. The simulation protocol of this heap-based coordinator in one simulation cycle is given below. Specifically, the heap-based coordinator first gets the smallest *tN* and the *imminents* from the heap. With these *imminents* in hand, the coordinator then asks (only) those *imminents* to compute output and send output. The *sendOut* message will trigger *imminents* to put their output messages to their destination simulators, which are called *influences*. The *influences*, like the *imminents*, need to execute their state transition functions. Thus before *imminents.tellAll("ApplyDelt",tN)*, the coordinator adds those *influences* into *imminents* by executing *imminents* = *imminents.addAll(influencees)*. At the end of the cycle, the coordinator asks all *imminents* to update their new *tNs* in the heap to prepare for the next simulation cycle.

```
tN = Heap.getMin()
imminents = Heap.getImms()
imminents.tellAll("computeOuput",tN)
imminents.tellAll("sendOutput")
imminents = imminents.addAll(influencees)
imminents.tellAll("ApplyDelt",tN)
imminents.tellAll("updateHeap")
```

It is worthy to point out that the computation complexity of finding the smallest tN is different between the standard coordinator and the heap-based coordinator. The standard coordinator finds the smallest tN by comparing tNs of all component simulators, which takes O(N) time, where N is the total number of component simulators. The heap based coordinator uses a heap structure to find the smallest tN which takes O(log(N)) time. By finding tN in an efficient way and focusing its computation only on the imminent models, the heap-based coordinator greatly improves the simulation performance as compared with the standard coordinator. Nevertheless, the heap-based coordinator still depends on the model on memory requirement and initialization time (the time needed for a simulation to start). When simulating a nonDS model with a large scale cells, the heap-based coordinator still takes long initialization time, and even becomes unable to run the simulation due to memory constrains.

The above discussion shows that the DS model and the heap-based coordinator each has its own gains and limitations from the performance point of view. When these two work together, they compensate to each other: the heap-based coordinator can take advantage of the DS model's low memory requirement and quick initialization time; the DS model can take advantage of the heap-based coordinator's efficient simulation protocol. This combination is especially attractive for simulating large scale cellular space models that have relatively small number of active models. For this type of models, the DS model reduces the number of loaded models in a simulation and the heap-based coordinator makes it efficient to go through the simulation cycle.

4. Performance Analysis

Before carrying out performance measurement, a detailed performance analysis is presented in this section. Without losing generality, the performance analysis is based on the following simulation protocol that is implemented by both the standard coordinator and the heap-based coordinator for simulating DEVS models.

```
Models.Construction();
While (stop condition is not met){
    simulators.AskAll("nextTN")
    tN = compareAndFindTN();
    simulators.tellAll("computeOutput",tN)
    simulators.tellAll("sendOutput")
    simulators.tellAll("ApplyDelt",tN)
}
```

From the above simulation protocol, the total execution time can be calculated as the sum of model construction time and the time for the simulation cycles. The model construction time depends on how many models need to be constructed and initialized. The simulation cycle time depends on the total number of simulation cycles and the execution time of each cycle (referred to as the *cycle execution time* in this paper). Next we analyze and compare the execution time for the four approaches described in the previous section: *nonDS-std* (simulate the nonDS model using the standard coordinator); *DS-std* (simulate the DS model using the standard coordinator); *nonDS-heap* (simulate the nonDS model using the heap-based coordinator); and *DS-heap* (simulate the DS model using the heap-based coordinator).

4.1 Time Complexity Analysis

In general, the total execution time of a simulation is represented using formula:

$$T = T_{\text{construct}} + \sum_{i=1}^{m\_step} T_i + T_{\text{overhead\_add}} + T_{\text{overhead\_delete}} \qquad (1)$$

where T is the total execution time. $T_{construct}$ is the model construction time; T_i is the cycle execution time at cycle *i* (excluding the runtime overheads of dynamic structure if they exist); *m_step* is the number of simulation cycles. *m_step* is dependent on the number of external, internal or confluent transitions, which are the measurement of message exchanges among models; $T_{overhead\_add}$ refers to the runtime overhead associated with dynamically adding models; $T_{overhead\_delete}$ is the overhead time spent for removing models. These two overheads exist only for DS models. We separate them out in order to make it easier to carry out the analysis. The four elements shown in Formula (1) have different values for *nonDS-std*, *DS-std*, *nonDS-heap*, and *DS-heap* as described below.

For *nonDS-std*, all models are loaded before a simulation begins. Thus $T_{construct}$ include all models' construction time. In each simulation cycle, all simulators go through the simulation steps in the cycle (see section 3.2). However, the execution time for *imminents* is different from that of *non-imminents*. This is because an imminent model needs to execute its state transition function and/or output function, thus needs longer execution time than a non-imminent model. For *nonDS-std*, the total execution time (denoted as $T_{nonDS\text{-}std}$) can be calculated from formula (1) using the following equations:

$$T_{construct} = N * t_{construct} \tag{2}$$

$$\sum_{i=1}^{m\_step} t_i = \sum_{i=1}^{m\_step} \left(\sum_{j}^{n_i} tij + \sum_{j}^{N-n_i} tij' + t_{smallest\_tN}\right) \tag{3}$$

$$T_{overhead\_add} = 0 \tag{4}$$

$$T_{overhead\_delete} = 0 \tag{5}$$

where N is the total number of cellular models, $t_{construct}$ is the construction and initialization time for one model. In formula (3), n_i is the number of *imminents* in cycle *i*, *tij* is the cycle execution time for imminent model *j*, *tij′* is the cycle execution time for non-imminent model *j′*. As mentioned before, $tij' < tij$. For example, in the forest fire simulation that will be presented later *tij′* is about 30% of *tij*. The $t_{smallest\_tN}$ is the time to find the smallest *tN* in every simulation cycle (referred as *find-tN* time). For the nonDS model we assume this time is the same for each cycle. We explicitly separate $t_{smallest\_tN}$ out in order to compare with the other three approaches. Both $T_{overhead\_add}$ and $T_{overhead\_delete}$ are 0 in *nonDS* model. Formula (2) – (5) shows that the execution time of *nonDS-std* is heavily influenced by the total number of models *N* and the number of *imminents* $\sum n_i$.

For *DS-std*, the simulation starts with a small set of active models and then adds new activated models and removes inactive models as the simulation proceeds. Note that the new added models, even created at different simulation steps, need to go through the same construction procedure. Thus in the following formulas, the construction time of all the added models is calculated using $T_{construct}$. The total execution time (denoted as $T_{DS\text{-}std}$) for *DS-std* can be calculated using the following formulas:

$$T_{construct} = t_{construct} * n_{created} \tag{6}$$

$$\sum_{i=1}^{m\_step} t_i = \sum_{i=1}^{m\_step} \left(\sum_{j}^{n_i} tij + \sum_{j}^{n_{created\_i}-n_{deleted\_i}-n_i} tij' + t_{smallest\_tN}'\right) \tag{7}$$

$$T_{overhead\_add} = t_{adding} * n_{created} \tag{8}$$

$$T_{overhead\_delete} = t_{deleting} * n_{deleted} \tag{9}$$

where $n_{created}$ and $n_{deleted}$ are the total number of added and deleted models; $n_{created\_i}$ and $n_{deleted\_i}$ are the total number of created and deleted models up to simulation cycle *i;* t_{adding}

and $t_{deleting}$ are the overhead time for adding and deleting a model. All other parameters have the same meanings as before. In formula (7), ($n_{created\_i}$ - $n_{deleted\_i}$) represents the number of exiting models in simulation cycle *i*. In the following we use n_{active_i} to denote it. Note that since models are dynamically added and removed, n_{active_i} at different simulation cycles is different and is typically much smaller than *N*. Because of this, the *find-tN* time $t_{smallest\_tN'}$ in *DS-std* is smaller than that in *nonDS-std*. For a particular cycle *i*, the number of *imminents* n_i in *DS-std* is the same as that in *nonDS-std*. One can see that for *DS-std* the execution time is influenced by the total number of created models $n_{created}$, the number of active models in every simulation cycle n_{active_i}, and the dynamic structure overheads. Since $n_{created}$ is smaller than *N*, the overall construction time in DS is smaller than that in nonDS.

For *nonDS-heap*, the total execution time (denoted as $T_{nonDS\text{-}heap}$) can be calculated using the following formulas:

$$T_{construct} = N * t_{construct} \tag{10}$$

$$\sum_{i=1}^{m\_step} t_i = \sum_{i=1}^{m\_step} \left(\sum_{j}^{n_i} tij + t_{smallest\_tN\_heap}\right) \tag{11}$$

$$T_{overhead\_add} = 0 \tag{12}$$

$$T_{overhead\_delete} = 0 \tag{13}$$

As can be seen, $T_{construct}$ is the same as in *nonDS-std*. Formula (11) shows that the cycle execution time in *nonDS-heap* is smaller than that in *nonDS-std*. This is mainly due to the following two reasons. First, in *nonDS-heap*, only the *imminents* go through the simulation cycles. Thus, formula (11) does not have *tij'*. Second, the heap-based coordinator finds the smallest *tN* in a more efficient way than the standard coordinator. For large scale cellular space models (*N* is large), $t_{smallest\_tN\_heap}$ can be much smaller than $t_{smallest\_tN}$ in *nonDS-std*. $T_{overhead\_add}$ and $T_{overhead\_delete}$ are 0 due to the nonDS model.

Finally, for the approach of *DS-heap*, the total execution time (denoted as $T_{DS\text{-}heap}$) can be calculated using the following formulas:

$$T_{construct} = t_{construct} * n_{created} \tag{14}$$

$$\sum_{i=1}^{m\_step} t_i = \sum_{i=1}^{m\_step} \left(\sum_{j}^{n_i} tij + t_{smallest\_tN\_heap}'\right) \tag{15}$$

$$T_{overhead\_add} = t_{adding} * n_{created} \tag{16}$$

$$T_{overhead\_delete} = t_{deleting} * n_{deleted} \tag{17}$$

In this approach, the $T_{construct}$, $T_{overhead\_add}$, and $T_{overhead\_delete}$ are the same as in *DS-std*. However, because of the heap-based coordinator, the same two reasons mentioned above apply here. As a result, the cycle execution time of *DS-heap* does not have *tij'*, and also $t_{smallest\_tN\_heap'}$ is smaller than the $t_{smallest\_tN'}$ in *DS-std*. When compared to *nonDS-heap*, one can see that the construction time is different. In *nonDS-heap*, all models are constructed and initialized. But in *DS-heap*, only those created models are constructed and initialized. Furthermore, the *find-tN* time in *DS-heap* is smaller than that in *nonDS-heap*. This is because the number of active models n_{active_i} in every simulation cycle is smaller than *N*, thus resulting in a smaller heap size. On the negative side, *DS-heap* introduces overheads $T_{overhead\_add}$ and $T_{overhead\_delete}$, which do not exist in *nonDS-heap*.

4.2 Performance Comparison of Different Approaches

Based on the above formulas, further analysis is carried out to compare the performance gains and loss of these approaches. From formulas (2) – (9) we calculate the execution time difference between *nonDS-std* and *DS-std*:

$$T_{nonDS\text{-}std} - T_{DS\text{-}std} = t_{construct} * (N - n_{created}) - t_{adding} * n_{created} - t_{deleting} * n_{deleted} + \sum_{i=1}^{m\_step} \left(\sum_{j}^{N-n_{active\_i}} tij' + t_{smallest\_tN} - t_{smallest\_tN}' \right) \quad (18)$$

Formula (18) shows that the performance difference between *nonDS-std* and *DS-std* comes from three sources: 1) Different from model construction time. Since $n_{created} < N$, *nonDS-std* takes longer construction time. 2) Difference from dynamic structure overhead. This acts against the *DS-std* and depends on the total number of dynamically added and deleted models. 3) Difference from cycle execution time. This is due to the different number of active models in each simulation cycle: *N* for *nonDS-std*, and n_{active_i} for *DS-std*. This makes the *nonDS-std* run slower not only because it has more non-imminent models to take care of (the *tij'* part in formula (18)), but also it takes longer *find-tN* time in each cycle. For large scale cellular space model that have small number of active models ($n_{active\_i} << N$), in the long run, the cycle execution time plays major roles in the performance difference between *nonDS-std* and *DS-std*. However we note that the dynamic structure overhead cannot be ignored and can become significant when the number of added and deleted models are large.

A similar comparison can be done between *nonDS-heap* and *DS-heap*. Their difference is given below.

$$T_{nonDS\text{-}heap} - T_{DS\text{-}heap} = t_{construct} * (N - n_{created}) - t_{adding} * n_{created} - t_{deleting} * n_{deleted} + \sum_{i=1}^{m\_step} (t_{smallest\_tN} - t_{smallest\_tN}') \quad (19)$$

Similarly, the difference comes from three sources that are model construction time: $t_{construct} * (N - n_{created})$, DS overhead: $t_{adding} * n_{created} - t_{deleting} * n_{deleted}$, and cycle execution time. What is different is that here the cycle execution time difference is mainly the *find-tN* time difference: $t_{smallest\_tN} - t_{smallest\_tN}'$, which depends on the heap size of the two approaches. When the heap only add those node with non-infinity next event time, the heap size of nonDS and DS becomes the same and thus the difference of *find-tN* time becomes zero. So the major difference between *nonDS-heap* and *DS-heap* is the surplus construction time of the nonDS model and the dynamic structure overhead time of the DS model. When the surplus construction time is more than the DS overhead, *DS-heap* will have better performance than *nonDS-heap*, otherwise the opposite.

The next comparison shows the performance impact of the heap-based simulation engine for DS models. To do this, the *DS-std* and *DS-heap* are compared. From formulas (6)-(9) and formulas (14)-(17), one can calculate the time difference between *DS-std* and *DS-heap:*

$$T_{DS\text{-}std} - T_{DS\text{-}heap} = \sum_{i=1}^{m\_step} \left(\sum_{j}^{n_{active\_i}-n_i} tij' + t_{smallest\_tN}' - t_{smallest\_tN\_heap}' \right) \quad (20)$$

Formula (20) shows that the execution time difference between *DS-std* and *DS-heap* comes from the cycle execution time only. The performance gain of the heap-based simulation engine is clear, and is due to two reasons mentioned before: first, the heap-based coordinator asks only the *imminents* to go through the simulation cycles; second, the heap-based coordinator is faster than the standard coordinator in *find-tN* time. A similar comparison can be done between nonDS-std and nonDS-heap.

To conclude, when the same simulation engine is considered, the DS model have performance gains by having less construction and initialization time, and faster cycle execution time. On the negative side, it has runtime overhead that could become significant when the number of added and deleted models becomes large. When the same model is considered, the heap-based coordinator is more efficient than the standard coordinator because of its more efficient simulation protocol.

4.3 Runtime Memory

While the above analysis focuses on the execution time, it is useful to look at the runtime memory too to compare the DS model and nonDS model. In general, the memory required to run a simulation depends on the number of loaded models in the simulation. For nonDS, this number is N because all models are loaded. For DS, this number is $n_{exsiting\_i\_max}$, which is the maximum of $n_{exsiting\_i}$ of all simulation cycles. IN almost all situations, $n_{exsiting\_i\_max}$ is less than N because inactive models are unloaded or dynamically removed. Thus for large scale cellular space models, the DS approaches (*DS-std* or *DS-heap*) need less memory than the *nonDS* approaches (*nonDS-std* or *nonDS-heap*). In fact, as the cellular space size increases, the DS approaches become the only feasible approaches for simulations on single computers because of the memory requirement.

5. Performance Measurement on Token Ring Model

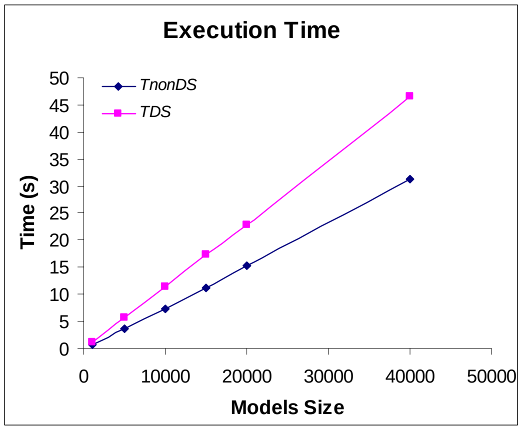
We start the performance measurement based on a simple one-dimensional cellular space model of token ring simulation. The measurement compares the simulation performance of DS model and nonDS model. Both simulations use the heap-based coordinator. The experiments (also the experiments for the forest fire spread example in the next section) were conducted on a Dell PC with Intel Celeron (M) 1.6GHZ processor, 1.0G memory, and Windows XP OS running DEVSJAVA version 3.0.

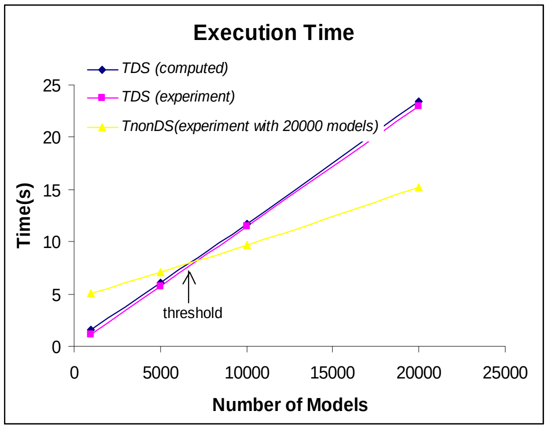
The token ring model simulates the process that a token is passed from a cell to a neighbor cell in a singly linked cell list. Only the cell that holds the token becomes active. All other cells are in inactive state. The nonDS implementation creates all cells in the beginning and passes the token from the head cell to the second cell, then from the second cell to the third cell. This process continues until the token reaches the tail cell or the simulation stops. In contrast, the DS implementation creates only the head cell and the second cell initially. When the token is passed to the second cell, the third cell is dynamically added and the first cell dynamically removed. Again, this continues until the tail cell is reached or the simulation stops. We define a *complete travel* of the token as the token is transferred from the head cell to the tail cell, a *partial travel* as the token only reaches some cell in the middle.

In both the nonDS and DS implementation, since at any time there is only one active cell (this means there is only one node in the heap), the *find-tN* time is negligible and is ignored in the following analysis. Based on the complexity analysis in section 4, from formulas (10) – (13) we have $T_{nonDS\text{-}heap} = t_{construct} * N + t_{DEVS} * n_{created}$; from formulas (14) – (17) we have $T_{DS\text{-}heap} = t_{construct} * n_{created} + t_{DEVS} * n_{created} + (n_{created} + n_{deleted}) * t_{overhead}$, where t_{DEVS} represents the DEVS function execution time for each model, and

$t_{overhead}$ represent the dynamic structure overhead for adding/removing a cell. Note that in this example for simplicity the overheads of dynamically adding cell and removing cell are treated as the same. From the above we can see that for a simulation of a *complete travel* of the token, $n_{created} = n_{deleted} = N$, so the DS implementation has worse performance than nonDS because it has extra overheads for creating and deleting models while the nonDS has not. However, for a simulation of a partial travel, the DS implementation can have better performance than the nonDS because it does not need to create all the cells.

We develop two measurement metrics to measure the performance results. The first metric measures the execution time of nonDS and DS for simulating a complete travel with different cellular space sizes. The results are depicted in Figure 2(a). Figure 2(a) shows that the execution time increases linearly with the cell space size for both the nonDS and DS model, and the DS model consumes more time than the nonDS model for the same cell space size. This is consistent to the analysis before. The second metric measures the execution time of nonDS and DS for simulating different partial travels for the cell space size of 20,000. Specifically, we simulate partial travels where the token is transferred from the head cell to the 1000th, 5000th, 10000th, and 20000th cell respectively. Figure 2(b) shows the results, where *TDS (experiment)* denotes the execution time of the DS model and *TnonDS* denotes the execution time of the nonDS model. As can be seen, when simulating partial travels that end early, the DS model gives better performance than the nonDS model. After the partial travel increases its ending position up to a certain point, the nonDS model results in better performance. The intersection point where the performance of the nonDS model bypasses that of the DS model is marked in Figure 2(b). We define this as the "threshold" point for this application, which is a dividing point to measure if the performance of the DS model is superior to the nonDS model or not. When simulating partial travels whose ending positions are smaller than the threshold point, the DS model gives better execution time. Otherwise, the nonDS model has better execution time.





(a) **(b)**

Figure 2. (a) Execution Time of Token Ring Model for Different Model Sizes
(b) Execution Time of Token Ring at Different Number of Models

To further analyze the amount of dynamic structure overhead for this application, we calculate $t_{overhead}$ from the measurement results. To do this, we first calculate $t_{construct}$ and

t_{DEVS} based on the measurement result of the nonDS simulation of a complete travel with 20000 cells. Specifically, $t_{construct}$ is calculated using the measured construction time of the simulation: $t_{construct} = T_{construt\_nonDS} / N$; then t_{DEVS} is calculated based on the formula $T_{nonDS\text{-}heap} = t_{construct} * N + t_{DEVS} * n_{created}$, where $n_{created} = N$. Finally, $t_{overhead}$ is calculated using the measurement results of the DS simulation of the same model according to $T_{DS\text{-}heap} = t_{construct} * n_{created} + t_{DEVS} * n_{created} + (n_{created} + n_{deleted}) * t_{overhead}$, where $n_{created} = n_{deleted} = N$ since this is a complete travel. The calculation shows that $t_{construct}$, t_{devs} and $t_{overhead}$ are 0.205ms, 0.591ms, and 0.409ms respectively. Using these values, we also compute the "analytic" execution time *TDS (computed)* for the simulations of partial travels shown in Figure 2(b). As can be seen, the computed execution time for these simulations matches very well with the measurement results.

This example shows that the execution time of DS and nonDS follows the time complexity analysis presented in section 4. Due to the simplicity of this token ring model that has only one active cell in the simulation all the time, we were able to calculate the amount of dynamic structure overhead for adding/deleting one cell. Computation results based on this dynamic structure overhead for partial travel simulations matches well with the measured results. The results show that the DS overhead cannot be ignored and will play a significant role as the number added/deleted cells increases. Next we carry out performance measurement using a more complex model: the forest fire spread model.

6. Performance Measurement on Forest Fire Spread Model

The forest fire spread model is a two-dimensional cell-space model composed of individual forest cells coupled together according to their relative 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 when burning cells ignite their unburned neighboring cells. Each cell can be in one of the following states: *unburn*, *burning*, and *burned*. When a cell is ignited, the maximum fire spread speed and direction of a cell is calculated using Rothermel's semi-empirical model [32] that takes into account factors such as fuel model, slope, and wind speed and direction. This maximum rate of spread is then decomposed into eight spreading directions according to an ellipse shape. More descriptions of this model can be found in [10].

Compared to the token ring model in the previous section, the forest fire spread model has complex spatial-temporal behaviors and thus are more difficult to analyze the performance results. In order to carry out comprehensive performance measurement, we developed five metrics that cover both the runtime memory and execution time of the simulations as listed below.

- Memory usage
- Initialization time for different cellular space sizes
- Execution time for different cellular space sizes
- Execution time of different simulation stages
- Execution time for different model behaviors

The first metric compares the memory usage of nonDS and DS models. The second one compares the initial construction time of different cellular space sizes. Here the initialization time includes the time to construct the model and to set up the simulators before going through any simulation cycles. The third metric compares the total execution time (including the initial construction time) under various cellular space sizes. The fourth metric shows the execution time at different stages of simulations using *DS-std* and *DS-heap*. This allows us to see how the dynamically added/deleted cells in different stages of a simulation affect the simulation execution time in those stages. The fifth metric measures the execution time with different model behaviors. This measurement aims to show the significance of the dynamic structure overhead for models whose behavior resulting in large number of added/deleted cells. It provides guidelines for selecting DS or nonDS models based on the model behavior.

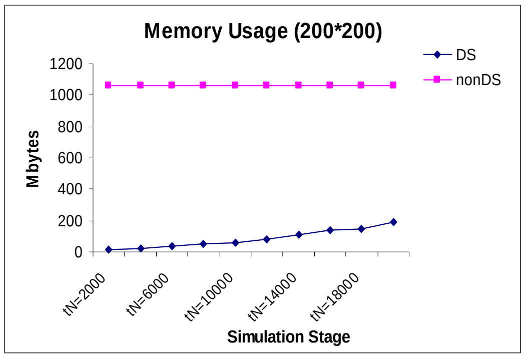
6.1 Memory Usage

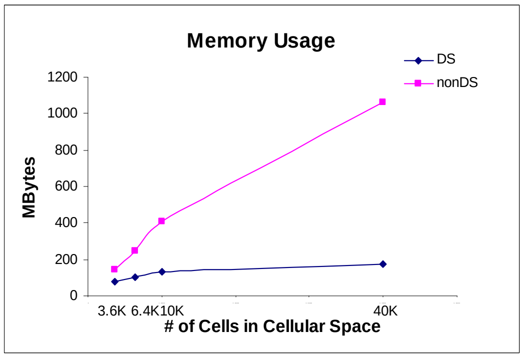
Memory usage is an important aspect to measure the simulation performance. Our previous analysis shows that the DS approach is superior to nonDS approach for memory usage. Figure 3 presents the comparison of memory usage between nonDS-heap and DS-heap. We note that for the same type of model, the difference of using a standard coordinator or using a heap-based coordinator is insignificant. Figure 3(a) shows the memory usage at different simulation stages when simulating a 200*200 cellular space model up to next event time tN=20000 second. As can be seen, in nonDS-heap the number of loaded cells always equals to the total cellular space size, so the memory usage stays on the top and does not change during the simulation. In DS-heap, the number of loaded cells (listed in Table 1) increases gradually as the simulation proceeds, so the memory usage increases accordingly. However, overall the DS-heap uses much less memory than the nonDS-heap does. This is because in the forest fire spread simulation, the number of active cells (i.e., the cells that belong to the fire front) is only a small portion of the total number of cells. Figure 3(b) shows the memory usage for different cell space sizes at the simulation time tN=20000. It shows that the memory usage for nonDS-heap "linearly" increases as the cell space size increases. But the memory usage for DS-heap does not change much because it depends on the number of loaded cells instead of the total number of cells in the cell space.

To summarize, the memory usage is closely related to the number of loaded cells in the simulation. DS model can result in much less memory usage than nonDS model because it does not load all the cells into the simulation at the same time. This feature makes the DS approach attractive for large scale cell space models. In such cases, the DS approach may become the only feasible approach because of its memory advantage.

Table 1. Number of Active Cells and Memory Use in DS-heap (200*200 Cell Space)

Simulation time (tN)	2000	4000	6000	8000	10000	12000	14000	16000	18000	20000
Number of active cells	89	210	340	459	577	711	819	958	1081	1185
Memory Use (Mbytes)	17	25	38	48	58	81	110	137	147	189





(a) **(b)**

Figure 3. Memory Usage for nonDS-heap and DS-heap
(a) 200*200 Memory Usage (b) Comparison of Different Cell Space Size

6.2 Initialization Time for Different Cellular Space Sizes

Table 2. Initialization Time (sec.) of Different Cell Space Size

$T_{initial}$ (s)	40*40	60*60	80*80	100*100	250*250	500*500	1000*1000	2000*2000
$T_{nonDS\text{-}std}$	3.4	13.2	38.5	91.2	N/A	N/A	N/A	N/A
$T_{nonDS\text{-}heap}$	3.4	13.1	38.6	91.4	N/A	N/A	N/A	N/A
$T_{DS\text{-}std}$	0.4	0.4	0.4	0.4	0.5	0.6	1.1	2.2
$T_{DS\text{-}heap}$	0.4	0.4	0.4	0.4	0.5	0.7	1.3	2.2

The initialization time of a simulation provides a direct measurement of how fast a simulation can start. In this experiment, we run simulations of forest fire spread model and measure the initialization time with different cell space sizes. Table 2 shows the results for models with sizes of 40*40, 60*60, 80*80, 100*100, 250*250, 500*500, 1000*1000 and 2000*2000. The data "N/A" means no result was collected because of the memory limitation due to the very large cellular space size. From Table 2 one can see that the initialization time of *nonDS-std* and *nonDS-heap* increases with the increase of cellular space size. This is because they need to initialize all the models at the beginning of the simulation. But for *DS-std* and *DS-heap*, the initialization time is little and increases slowly with the increase of cellular space size. This slight increase is because the DS model uses a 2D array to keep track of each cell's loading status. Therefore, with the increase of cellular space size, the setup time of this 2D array increases accordingly. Overall, the trend of initialization time matches with memory usage. Table 2 shows that the DS based approaches are much faster than the nonDS based approaches to start a simulation.

6.3 Execution Time for Different Cell Space Sizes

According to the time complexity analysis in section 4, the total execution time includes initialization time as well as the cycle execution time. In this measurement we run simulations with different cell space sizes and measure their execution time up to the simulation time tN = 12000. Table 3 records the results and Figure 4 illustrates them for *nonDS-heap*, *DS-std* and *DS-heap*. Similarly, the data "N/A" means no result was collected because of the memory limitation due to the large cell space size.

Table 3. Execution Time (sec.) (tN = 12000) for Different Cellular Space Sizes

	40*40	60*60	80*80	100*100	250*250	500*500	1000*1000	2000*2000
$T_{nonDS\text{-}std}$	21.4	67.4	149.6	278.8	N/A	N/A	N/A	N/A
$T_{nonDS\text{-}heap}$	7.1	20.8	52.2	118.4	N/A	N/A	N/A	N/A
$T_{DS\text{-}std}$	11.1	20.8	29.5	34.6	35.5	34.9	34.7	34.5
$T_{DS\text{-}heap}$	9.2	16.0	22.7	25.5	26.3	25.9	25.9	26.1

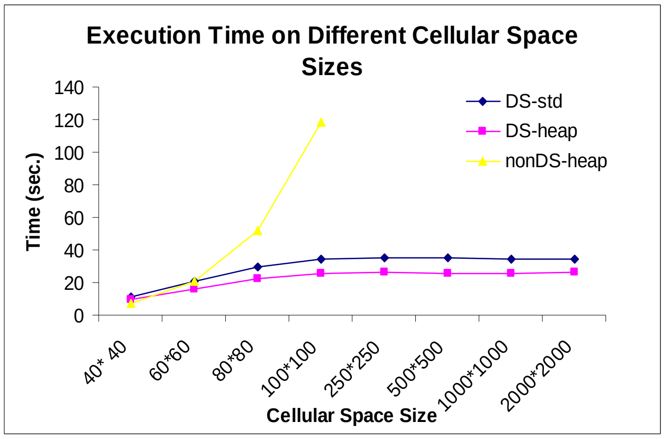


Figure 4. Execution Time of Different Cell Space Sizes

Table 3 shows that for the same cell space size, *DS-heap* is the most efficient and *nonDS-std* is the least efficient. This can be explained from formulas (18) - (20). In particular, formula (19) computes the difference between DS-heap and nonDS-heap. It shows that DS-heap has better performance than nonDS-heap when the DS overhead of the DS-heap is less than the surplus construction time of nonDS-heap. This is what we see in Table 3 for all cases except for 40*40, where nonDS-heap gives better performance result than DS-heap. For this case, the number of added cells $n_{created}$ almost reaches N when tN=12000. So the surplus construction time of nonDS-heap is small. Overall in this measurement because the simulations end early (tN = 12000), only relatively small number of cells have been dynamically added/removed, thus the DS overhead is relatively small. This makes the DS approaches have better performance than the nonDS approaches as shown in Table 3 and Figure 4. We note that because of the model construction time, both $T_{nonDS\text{-}heap}$ and $T_{nonDS\text{-}std}$ increases when the cellular space size increases. However this is not true for $T_{DS\text{-}heap}$ and $T_{DS\text{-}std}$. After a certain point, further increase of the cell space size will have no effect on the number of added/removed cells and thus the execution time of *DS-std* and *DS-heap* will remain unchanged. This is because in this particular forest fire application, the spread speed of fire front keeps constant after the simulation proceeds to a certain point. So the increase of the cellular space size does not affect the execution time.

6.4 Execution Time at Different Simulation Stages

This section provides a performance measurement on DS-std and DS-heap to show how the dynamically added/deleted cells in different stages of a simulation affect the simulation execution time. Figure 5 displays the relationship between the execution time and $n_{created}/n_{deleted}/n_{active}/n_{imminent}$ for *DS-std* and *DS-heap* on a 200*200 cell space. In this experiment, we divide the entire simulation into multiple stages. Each stage consists of 2000 simulation steps (referred to as *m_step* in formulas (2) – (17)). For example, stage 6000-8000 represents the simulation stage from simulation step 6000 to 8000. In the figure, $n_{created}$ and $n_{deleted}$ represents the number of created and deleted models respectively during a stage; n_{active} represents the number of existing models at the end point of a stage, which is calculated by adding the change of number of cells in this stage ($n_{created}$ - $n_{deleted}$) to the n_{active} in the previous stage; and $n_{imminent}$ is the total number of cells that are imminent in a stage. We note here that $n_{imminent}$ is used as the measurement of event transitions or activity during a stage. The left y-axis denotes the execution time for $T_{DS\text{-}std}$ and $T_{DS\text{-}heap}$, while the right y-axis denotes the number of cells for $n_{created}/n_{deleted}/n_{active}/n_{imminent}$.

Figure 5 shows that $n_{created}$ at different stages maintains about the same and decreases very slowly, whereas $n_{deleted}$ increases slowly. The $n_{created}$ is larger than $n_{deleted}$ before t_{step} = 52000, and becomes smaller after that. As a result, n_{active} increases in the beginning and then decreases after t_{step} = 52000. One can see that $T_{DS\text{-}std}$ follows the same trend as the number of active cells n_{active}. This can be explained as follows. As shown in formulas (6)-(9), the execution time of *DS-std* includes three parts: 1) $t_{construct}$ * $n_{created}$, 2) active models' cycle execution time, which is dependent on n_{active} and m_step, and 3) DS overheads t_{adding} * $n_{created}$ and $t_{deleting}$ * $n_{deleted}$. In this experiment, each stage has 2000 cycles and the number of active cells is large. So the stage execution time is mainly determined by the active cells' execution time, thus shows the same trend of the number of active cells. There lie differences between these two lines as shown in Figure 5. This is due to the overheads of adding/deleting cells, which cause, for example, the execution time to increase faster than the number of active cells does.

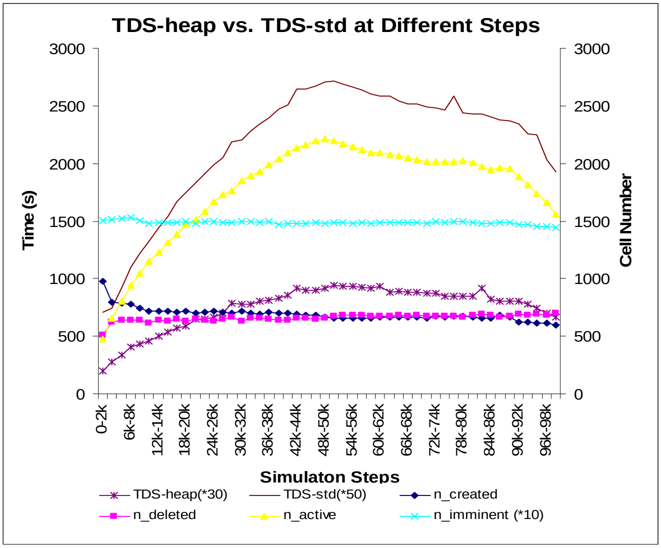


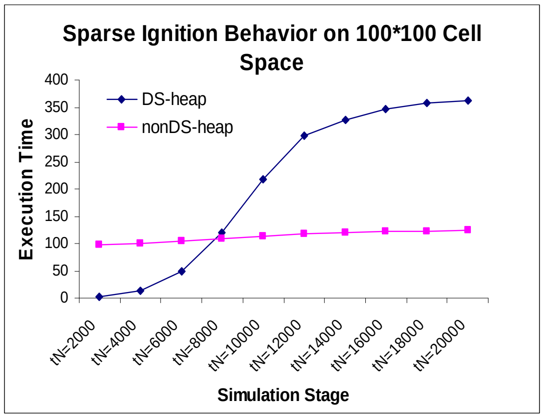
Figure 5. Execution Time at different stages for *DS-std* and *DS-heap*

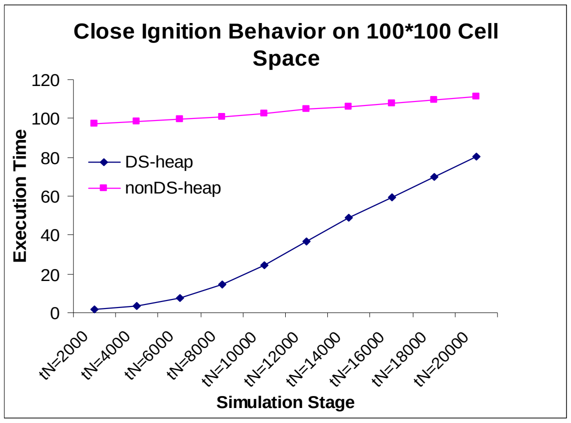
From Figure 5 one can also see the number of imminent cells $n_{imminent}$ keeps almost the same for different stages. Note that the heap-based coordinator asks only *imminents* to go through the simulation cycles. This explains why $T_{DS\text{-}heap}$ does not follow the trend of n_{active}. Instead, it "smoothes" out by following the trend of $n_{imminent}$ that is almost constant along all stages. On the other hand, n_{active} still plays a role there because it affects the *find-tN* time in the heap-based coordinator (see formula (15)). The results of DS-heap and nonDS-heap shown in Figure 5 are consistent with the analysis in Section 4.

6.5 Execution Time for Different Model Behaviors (Multiple Ignition Points)

In the above metrics, the overheads of dynamic structure are not very obvious because $n_{created}$ and $n_{deleted}$ are relatively small. In this experiment, we intend to show that the DS overheads can become significant. To do this we set up simulations that use multiple ignition points to ignite multiple fires at the same time.

Figure 6 and Table 4 show the performance results when 5 sparse ignitions and 5 close ignitions are used separately in a 100*100 cellular space for *nonDS-heap* and *DS-heap*. Here we differentiate two situations: a sparse ignition situation where the five ignition points are far away from each other; and a close ignition situation where the five ignition points are close to each other. In the sparse ignition situation, five fires are started and spread independently for a long time, whereas in the close ignition situation the five fires quickly merge into a single fire. We measure the execution time (the time from when the simulation starts) every 2000 simulation time point along the entire process of the simulation (up to tN = 20000). The measurement results are recorded in Table 4 and depicted in Figure 6 for both the sparse and close ignition situations. From the results, one can see that because the sparse ignition situation has more $n_{created}$, $n_{deleted}$ and n_{active} ($n_{created}$ - $n_{deleted}$), its execution time is more than that for the close ignition situation. This is true for both DS-heap and nonDS-heap. In the case of nonDS-heap, the execution time difference between the sparse and close ignition situations is small. But in the case of DS-heap, their execution time difference is much larger. This large difference is partially due to the large number of active cells in the sparse ignition situation. More importantly, it is due to the DS overheads. Because a lot more cells are dynamically added and deleted in the sparse ignition situation, thus a lot more DS overheads exist, which leads to larger execution time.





(a) (b)

Figure 6. (a) Execution Time of Sparse Multiple Ignitions
(b) Execution Time of Close Multiple Ignitions

Table 4. (a) Sparse Ignitions (100*100 Cell Space)

tN	2000	4000	6000	8000	10000	12000	14000	16000	18000	20000
$n_{created}$	455	1360	2737	4322	5858	6718	7392	7922	8280	8484
$n_{deleted}$	10	310	1095	2325	3859	5501	6427	7141	7697	8116
$T_{DS\text{-}heap}$	2.4	12.6	48.6	120.7	218.8	296.8	325.9	345.7	357	362.8
$T_{nonDS\text{-}heap}$	97.8	100.3	103.8	108.2	113.3	117	119.5	121.6	123.2	124.1

Table 4. (b) Close Ignitions (100*100 Cell Space)

tN	2000	4000	6000	8000	10000	12000	14000	16000	18000	20000
$n_{created}$	195	449	812	1255	1794	2318	2801	3279	3752	4187
$n_{deleted}$	10	150	380	708	1138	1655	2168	2657	3118	3582
$T_{DS\text{-}heap}$	1.6	3.7	7.7	14.4	24.3	36.9	48.8	59.7	69.8	80.1
$T_{nonDS\text{-}heap}$	97.5	98.3	99.5	100.9	102.7	104.6	106.3	108	109.7	111.3

The significance of DS overheads can also be illustrated when comparing *DS-heap* and *nonDS-heap* for the same model behavior, for example, the sparse ignition behavior shown in Figure 6(a). As can be seen, at the beginning of the simulation *DS-heap* uses less time than *nonDS-heap* because the nonDS model needs to loads all the cells before the simulation starts. However as the simulation proceeds $T_{DS\text{-}heap}$ bypasses $T_{nonDS\text{-}heap}$ (at tN = 8000). According to formulas (10)-(17), the reason that can cause this bypass is the dynamic structure overheads, which depend on $n_{created}$ and $n_{deleted}$. As $n_{created}$ and $n_{deleted}$ increases, the DS overheads increase too. Eventually the execution time of *DS-heap* becomes larger than that of *nonDS-heap* as shown in Figure 6(a). For the close ignition situation shown in Figure 6(b), until tN = 20000 DS-heap is still better than nonDS-heap. This is because $n_{created}$ and $n_{deleted}$ are not large enough to make the DS overheads become dominant. However based on the trend of $T_{nonDS\text{-}heap}$ and $T_{DS\text{-}heap}$, if the simulation continues further, these two lines will intersect and $T_{DS\text{-}heap}$ will bypass $T_{nonDS\text{-}heap}$ eventually.

The above analysis shows that the number of added and deleted cells plays an important role to decide if a DS model outperforms or underperforms a nonDS model. Similar to what has been discussed in the token ring model, there exists a "DS-threshold" for the forest fire spread model. This "DS-threshold" indicates when the DS model will become underperformed than the nonDS model. For a forest fire simulation, we can roughly compute the ratio of the number of added and deleted cells ($n_{created}$+$n_{deleted}$) over the total number of cells *N* and then check if it reaches the "DS-threshold". A simulation whose ($n_{created}$+$n_{deleted}$)/N is smaller than the "DS-threshold" will give better performance result when using the DS model. Otherwise the nonDS model will give better performance result.

The concept of the "DS-threshold" is further illustrated by another experiment that takes 100 randomly generated ignition points in a 100*100 cell space up to tN=20000. Same as before, we compare the nonDS-heap and DS-heap approaches. Figure 7 presents the performance results. It can be seen that in this experiment, the DS-threshold is easily reached because there are 100 ignition points, which quickly result in a large number of added and deleted cells (and thus reaches the “DS-threashold”). Therefore, the DS model quickly underperforms the nonDS model from the execution time point of view.

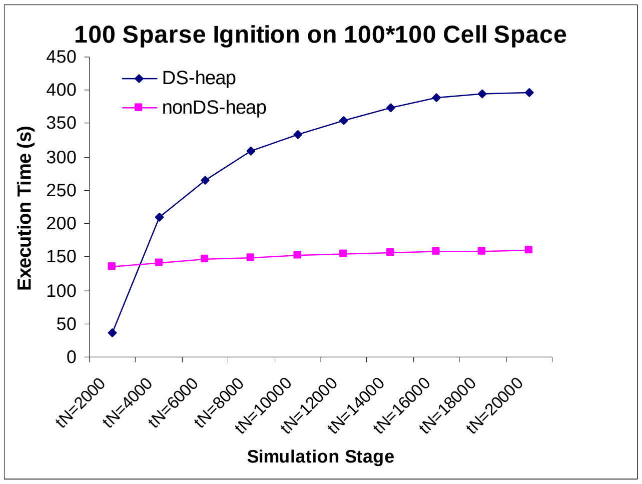


Figure 7. 100 Sparse Ignition for *DS-heap* and *nonDS-heap*

7. Discussions

The measurement results from both the token ring example and the forest fire spread example show that dynamic structure modeling has important impact on the simulation performance for large scale cellular space models. By dynamically adding and removing cells as needed, a DS model maintains only the active cells instead of loading the entire cell space. This has positive influences to the simulation performance from three aspects. First, it reduces the memory requirement as compared to a nonDS model. Second, it results in fast initialization time for models that have small number of active cells in the beginning. Third, maintaining only the subset of active cells reduces the search space for the simulation engine and allows the simulation engine to focus its computation power on the active cells in a more effective manner. However, on the negative size, the overhead of dynamically adding/removing cells at runtime cannot be ignored and can become significant (even dominant) for simulations that activate most of the cells in the cell space. In these cases, the advantage of not constructing all the cells in the cell space diminishes and the disadvantage of DS overhead becomes dominant. It is important to note that these performance gains and loss of DS modeling become significant only for large scale cellular space models, which is what this paper focuses on.

The performance gains and loss of DS modeling results in an effect of “DS-threshold” that was demonstrated in both the token ring simulation example and the fire spread simulation example. Specifically, for a given cellular space size, the DS model gives better performance if a simulation ends with small number of cells being activated. This could happen either the simulation terminates early (for example, simulating only 10 minutes of fire spread) or the number of cells involved in the simulation is small (for example, a fire spread simulation where most of the cells are unburnable). As the number of activated cells increases, the overhead of dynamically adding/deleting cells increases and eventually the DS model underperforms the nonDS model. This happens either because the simulation lasts for enough long time or the model behavior activates large number of cells in a short time like in the multiple ignitions experiment presented before. In the extreme case where all cells are activated, the DS model will definitely results in slower execution time than the nonDS model. Because this effect of “DS-threshold” is due to the inherent performance gains and loss of DS modeling, it is general for all DS models, and can be used as a guideline for deciding if a DS model gives better or worse performance than a nonDS model. However, we note that the specific value of the “DS-threshold” is application dependent, and may be difficult to calculate for models with complex behavior such as the fire spread model. This is because the runtime overhead of dynamically adding/deleting models is closely related to the specific implementation of the model and simulator, as well as the programming language (such as object creation and garbage collection in JAVA).

Based on the above discussions, we suggest DS modeling be used for the following situations: 1) the system to be modeled is dynamic structure in nature. In this case, the motivation is not on the performance aspect but on the modeling power of DS model; 2) large scale models that require excessive amount of memories if loading all the models all at once. In this case, the DS modeling (if it can be applied) can help to load only a subset of the models in a dynamically manner; 3) simulations have massive number of models but only a relatively “small” portion of them are active. In this case, DS modeling significantly saves model construction time by introducing relatively small DS overhead. As mentioned before, for larges scale cellular space models, this could be either the simulation terminates early or the number of cells involved in the simulation is small in nature.

8. Conclusions

Dynamic Structure DEVS (DSDEVS) is an advanced modeling technique that allows DEVS models and their couplings to be dynamically changed for modeling complex systems. In this paper, we focus on the performance aspect of dynamic structure DEVS, and carry out comprehensive performance analysis and performance measurement based on a token ring model and a large scale forest fire spread model. Approaches of simulating DS model and nonDS model using a standard coordinator and a heap-based coordinator are considered. The analysis and measurement results show that dynamic structure modeling has both positive and negative impacts on the simulation performance for large scale cellular space models. Discussions about DS modeling’s applicability in different situations are provided

Acknowledgements

This research was supported by grants CNS-0540000 and CNS-0720675 from the National Science Foundation.

References:

[1] Barros, F.J. 1997. Modeling Formalisms for Dynamic Structure Systems. *ACM Transactions on Modeling and Computer Simulation* 7(4), 501-515.

[2] Zeigler, B.P., T.G. Kim, and H. Praehofer. 2000. *Theory of Modeling and Simulation,* 2nd edition. Academic Press, New York, USA.

[3] Uhrmacher, A.M. 2001. Dynamic Structures in Modeling and Simulation - A Reflective Approach. *ACM Transactions on Modeling and Simulation* 11(2), 206-232.

[4] Zeigler, B.P., T.G. Kim and C. Lee. 1991. Variable structure modelling methodology: An adaptive computer architecture example. *Trans. Sot. Comput. Simulation* 7(4), 291-319.

[5] Barros, F. J. and M. T. Mendes. 1997. Forest fire modelling and simulation in the DELTA environment. *Simul. Pr. Theory* 5(3), 185-197.

[6] Muzy, A., E. Innocenti, D.R.C. Hill, A. Aïello, J.F. Santucci, and P.A. Santoni. 2004. Dynamic structure cellular automata in a fire spreading application. *Proceedings of the First International Conference on Informatics in Control, Automation and Robotics, IEEE/CSS/IFAC/ACM/AAAI/APPIA, Setubal, Portugal*. 143–151.

[7] Hu, X., B. P. Zeigler, and S. Mittal. 2005. Variable Structure in DEVS Component-Based Modeling and Simulation. *Simulation: Transactions of The Society for Modeling and Simulation International* 81(2), 91-102.

[8] Wainer, G. and N. Giambiasi. 2002. N-Dimensional Cell-DEVS. *Discrete Events Systems: Theory and Applications* 12(1), 135–157.

[9] Wainer, G. A. Modeling and simulation of complex systems with Cell-DEVS. 2004. *Proceedings of the 36th conference on winter simulation* 1, 45-56.

[10] Ntaimo, L., B. Khargharia, B. P. Zeigler and M. J. Vasconcelos. 2004. Forest fire spread and suppression in DEVS, *SIMULATION*, 80 (10), 479-500.

[11] DEVS-Java Reference Guide, www.acims.arizona.edu.

[12] Mittal, S., E. Mak, and J.J. Nutaro. 2006. DEVS-Based Dynamic Model Reconfiguration and Simulation Control in the Enhanced DoDAF Design Process. *Journal of Defense Modeling and Simulation (JDMS)* 3(4), 239-267.

[13] Sun, Y. and X. Hu. 2007. Performance measurement of DEVS dynamic structure on forest fire spreading simulation. *Proc.14th AI, Simulation and Planning in High Autonomy Systems (AIS 2007)* 12.

[14] Finney, M.A. 1998. FARSITE: Fire area simulator – Development and Evaluation. *Research Paper RMRS-RP-4.* US Dept. of Agriculture, Forest Service, 52.

[15] Filippi, J-B. and P. Bisgambiglia. 2002. Enabling large scale and high definition simulation of natural systems with vector models and JDEVS. *Proceedings of the 2002 Winter Simulation Conference*. 1964-1970.

[16] Berger, M. J. and P Colella. 1989. Local Adaptive Mesh Refinement for Shock Hydrodynamics. *J. Comput. Phys*. 82, 64-84.

[17] Hu, X. and L. Ntaimo. 2006. Dynamic Multi-resolution Cellular Space Modeling for Forest Fire Simulation. *Proc. DEVS Integrative M&S Symposium (DEVS'06),* Spring Simulation Multiconference, April 2-5, 95-102.

[18] Hu, X. and B. P. Zeigler. 2004. A High Performance Simulation Engine for Large-Scale Cellular DEVS Models. *High Performance Computing Symposium (HPC'04), Advanced Simulation Technologies Conference*, April, 3-8.

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

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

[21] 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. 1333-1340.

[22] Zeigler, B., and H. S. Sarjoughian. 1999. Support for hierarchical modular component-based model construction in DEVS/HLA. *Simulation Interoperability Workshop*, March 14-19, Orlando, FL.

[23] 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.

[24] Mittal, S., J.L. Risco-Martín, and B.P. Zeigler. 2007. DEVSML: Automating DEVS Execution Over SOA Towards Transparent Simulators. DEVS Symposium, *Spring Simulation Multiconference*, March, 287-295, Norfork, Virginia.

[25] Nutaro, J. 2004. Risk-free optimistic simulation of DEVS models. *Advanced Simulation Technologies Conference – Military, Government, and Aerospace Simulation Symposium*, 113-118, Arlington, VA.

[26] Sunwoo, P. 2003. Hierarchical Model Partitioning for Distributed Simulation of Hierarchical and Modular DEVS Models. Ph.D. Dissertation, Univ. of Arizona, May.

[27] Foster, I., C. Kesselman, and S. Tuecke. 2001. The Anatony of the Grid: Enabling Scalable Virtual Organizations. *Int'l J. High-Performance Computing Applications* 15(3), 200-222.

[28] Foster, I. 1998. Computational Grids. Reprinted by permission of Morgan Kaufmann Publishers from The Grid: The Blueprint for a Future Computing Infrastructure.

[29] Song, H. J., X. Liu, D. Jakobsen, R. Bhagwan, X. Zhang, K. Taura, and A. Chien. 2000. The MicroGrid: a Scienti_c Tool for Modeling Computational Grids. *Scientific Programming* 8(3), 127-141.

[30] http://www.Globus.org

[31] Seo, C., S. Park, B. Kim, S. Cheon, and B. P. Zeigler. 2004. Implementation of Distributed high-performance DEVS Simulation Framework in the Grid Computing Environment. In *Proceedings of the 2004 Advanced Simulation Technologies Conference (ASTC '04)* — High Performance Computing Symposium 2004 (HPCS 2004), 9-15, Arlington, VA.

[32] Rothermel, R. 1972. 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.

[33] Wyman, F. P. 1975. Improved event-scanning mechanisms for discrete event simulation. *Commun. ACM*, 18, 350-353.

[34] Vaucher, J. G. and P. Duval. 1975. A comparison of simulation event list algorithms. *Commun. ACM*, 18, 223-230.

[35] Steinman, J. S. 1996. Discrete-event simulation and the event horizon part 2: Event list management. In *PADS '96: Proceedings of the Tenth Workshop on Parallel and Distributed Simulation*, 170-178.

[36] Rönngren, R., J. Riboe and R. Ayani. 1991. Lazy queue: An efficient implementation of the pending-event set. In *ANSS '91: Proceedings of the 24th Annual Symposium on Simulation*, 194-204.

[37] McCormack, W. M. and R. G. Sargent. 1981. Analysis of future event set algorithms for discrete event simulation. *Commun. ACM*, 24, 801-812.

[38] Jones, D. W. 1986. An empirical comparison of priority-queue and event-set implementations. *Commun. ACM*, 29, 300-311.

[39] Chung, K., J. Sang and V. Rego. 1993. A performance comparison of event calendar algorithms: an empirical approach. *Softw. Pract. Exper.*, 23, 1107-1138.

[40] Bahr, H. and R. DeMara. 2004. Smart priority queue algorithms for self-optimizing event storage. *Simulation Modeling Practice and Theory*, 12 (April), 15-40.

[41] SLEATOR, D. D. AND TARJAN, R. E. 1985. Self-adjusting binary search trees. *J. ACM 32*, 3 (July), 652–686.

[42] Vaucher, J. and G. P. Duval. 1975. A comparison of simulation event list algorithms, Communications of the *ACM* 18(4), 223-230.

[43] Brown, R. and Calendar queue. 1988. A fast O(1) priority queue implementation for the simulation event set problem. C*ommunication of the ACM* 31(10), 1220-1227.

[44] Zeigler B. P. and H. Sarjoughian. 2002. *Introduction to DEVS Modeling and Simulation with JAVA: A Simplified Approach to HLA-Compliant Distributed Simulations*, The University of Arizona, Tucson, Arizona, USA, http://www.acims.arizona.edu.

[45] Fujimoto, R. M. *Parallel and Distribution Simulation Systems*. Wiley. 1999.

[46] Glinsky, El. and G. Wainer. 2005. Abstract simulation algorithms for Parallel CD++. Technical Report SCE-05-11. Carleton University. 2005.

[47] 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*. 273-282, Montreal, Canada 2005.

[48] Glinsky, E. and G. A. Wainer. 2005. DEVStone: a Benchmarking Technique for Studying Performance of DEVS Modeling and Simulation Environments, 9th *IEEE International Symposium on Distributed Simulation and Real Time Applications*. 273-282, Montreal, Canada 2005.

[49] RICK SIOW MONG GOH and IAN LI-JIN THNG. 2005. Twol-Amalgamated Priority Queues. *ACM Journal of Experimental Algorithmics*, 9(1.6), 1–45.

[50] L'Ecuyer, P. and Y. Champoux. 2001. Estimating small cell-loss ratios in ATM switches via importance sampling. *ACM Trans. Model Computer Simulation*. 11(1) (Jan.), 76-105.

[51] Willig, A. 2005. Performance Evaluation Techniques, Fundamentals and Big Picture. Telecommunication Networks Group, Technical University Berlin.

[52] Le Boudec, J. Y. 2007. Performance Evaluation of Computer and Communication Systems. EPFL.

[53] Jain, R. 1991. The Art of Computer Systems Performance Analysis. John Wiley & Sons, England.

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.*