Hierarchical Discrete Event Simulation Models

Ruzelan Khalid¹, Wolfgang Kreutzer², Tim Bell³
¹College of Arts and Sciences
Universiti Utara Malaysia, 06010 UUM Sintok, Kedah, Malaysia
²,³Department of Computer Science and Software Engineering
University of Canterbury, Private Bag 4800, Christchurch 8140, New Zealand
ruzelan@uum.edu.my, {²wolfgang.kreutzer, ³tim.bell}@canterbury.ac.nz

Abstract. Models are typically large and complex. One method to manage its complexity is through partitioning their levels of abstractions. This however still requires their behaviours to be executed in the right orders. This paper presents two mechanisms for coordinating event executions among layers in hierarchical Discrete Event Simulation (DES) models. These are the Monitor Delegation Mechanism that delegates event executions to a relevant layer and the Monitor Communication Mechanism that transfers event executions to all visited layers. The implementation of both mechanisms in a computer environment especially in the Flash environment and some potential problems are also discussed.

Keywords: abstraction, hierarchical DES, animation, component based modelling, design pattern

1. Introduction

The importance and techniques of building simple models have long been discussed [e.g., see 1, 2, 3]. However, complex and large models are sometimes preferred for many reasons (e.g., to represent real systems better, to have more confidence and understandable models, etc.), and have been supported by computing power improvement in terms of CPU speed and memory size. Consequently, their development poses challenges in all aspects of simulation modelling; e.g., how to properly build, verify, validate, present animation and visualization of the models.

Hierarchical simulation is an approach for managing the complexity of large simulation models through selectable different levels of abstractions. This complexity can either relate to the cognitive aspect (i.e., how easy the model logics can be grasped) or the representational aspect (i.e., how many elements are used and how they are arranged to represent model structures). For this, hierarchical simulation manages both aspects through its ability in arranging animation and visualization for better viewing and grasping the dynamic parts of a model (as opposed to the crowdedness of graphical objects in a flat model), and structuring the level of details for better representing the model (i.e., controlling information of the model).

This paper presents how to design an environment that partitions large and complex discrete event models (DES) [4] into levels of details through hierarchical structured concepts (i.e., by breaking up a model to sub-models). It is organized into six sections. Section 2 presents the basic concepts and some challenges of developing hierarchical DES models. Section 3 architects and traces two mechanisms of coordinating events in hierarchical DES models. Section 4 discusses their implementation in a computer environment specifically in the Flash environment [5] and lists some potential problems. Section 6 draws some conclusions.

2. Concepts and Challenges of Hierarchical DES Model Developments

Fig. 1 illustrates an example of a hierarchical DES model. The model is partitioned to four layers (Layer 1 to Layer 4) with the top layer (Layer 1) represents the model’s most basic structures. The execution of each layer is dependant on its lower layers since they comprise more detail structures and functions of their upper layers. These sets of structures are aggregated in sub models. Each sub-model has its own window used for locating its component structures, data visualization objects and animated entities. Entities flowing on this layer must be well synchronized with its lower layers; i.e., these entities must be executed appropriately based on their time delays and current simulation time.

Exploring this sub-model symbol takes users to a lower layer, thus revealing users to the layer’s structures. At any layer, there could be a sub-model that contains its own types of entities that are not transferred to any other layers. However, their flows must also be synchronized with the global model time.
Representing such concepts in a computer environment posts some challenges. Firstly, mechanisms need to be architected for connecting and synchronizing a parent model with its child model since its execution is dependent on the child model. This requires a mechanism not only to synchronize entity flows in a relevant layer but also to properly transfer these entities to its child model and back to the layer whenever they exit the last components of the child model. Secondly, facilities need to be provided to hide and display sub-model windows for properly controlling model abstraction, and these windows must truly represent the current states of their behaviours in terms of animation and visualization. Thirdly, the preservation of storing model states, visualization and component parameters for each model layer is important so that when users revisit the layer, they will get back the settings they have had before.

Our approach differs from the approach proposed by Yi and Cho [6, 7]. Our approach is based on concurrent animations where a simulation Agenda (i.e.; an event list) is used by a simulation monitor to control both aspects of simulation and animation. Thus, we only need to focus on event executions among layers. The advantage of this is that runtime interactions with the model can be supported. Their approach meanwhile is based on direct-simulation animation where a simulator and an animator are separated and have their own activity scheduling lists. Thus, besides considering event executions among layers in the simulator, they also need to find a method of communicating the simulator with animation scheduling in the animator. Although this approach offers separation of simulation-animation, animation accuracy is only guaranteed from event to event, not between them since the graphics rendering depends on computing subsystems.

3. Designing Mechanisms for Hierarchical DES Models

We have designed two mechanisms for coordinating event executions in hierarchical DES models. The main trick is sorting events in all hierarchies and executing the imminent event accordingly. First of all, these objects have to be introduced:

1. \((*, t)\) Messages
\((*, t)\) messages are additional messages to entity messages (i.e., dynamic entities flowing in DES models). They are also inherited from the entity class; e.g., the \textit{SimProcess} class [8, 9]. The main differences are:

   - entities flow from component to component while \((*, t)\) messages flow from layer to layer to coordinate event executions in the layers,
   - flowing entities from component to component typically consumes some delays while flowing \((*, t)\) messages does not incur delay,
- entities contain personal information (e.g., birth time, delay time, etc.) while \( (*, t) \) messages only contain the lowest simulation time of the source layers, the \( t \) value is however not used to update simulation time,
- entities are created by a Source component (i.e., a type of component that creates entity instances) while \( (*, t) \) messages are created by a Submodel object.

The insertion of \( (*, t) \) messages makes an Agenda looks clumsy. However, their existence is important to tally event executions.

2. Submodel Objects
A Submodel object encloses another layer. Entities arriving at a Submodel object could be in one of two cases: (1) the entities are from the same layer’s previous component, or (2) the entities are from a lower layer’s last component; see Fig. 2. To differentiate these entities, the entity class needs to have a property; e.g., named fromLayer that takes a value of current (the first case) or child (the second case).

![Fig. 2: Submodel Architecture and Transferring Mechanisms](image)

For the first case, the entities continue their flows to a lower layer’s first component through a child port; i.e., a port specifying the child model’s first component. For the second case, the entities flow to the same layer’s next component through an output port; i.e., a port storing its downstream component.

3. Local monitor
Each layer has its own local monitor that executes the layer’s activities stored in its Agenda in the right order.

3.1 First Mechanism: Monitor Delegation Mechanism
When a model is loaded, each Submodel inserts a \( (*, t) \) message to its local monitor to find the layer that has the lowest simulation time; e.g., in case of a Submodel object contains its own types of entities, or a Submodel object is the first component that locates a Source component under it. The model execution starts with the top layer’s monitor removes the \( (*, t) \) message and transfer it to its lower layer’s first component which then inserts the message to its local monitor. This process continues until the imminent entity is found in a relevant layer. The entity will then be executed so that it can flow to the same layer or to another layer. Their flows to another layer must be accompanied by a \( (*, t) \) message.

The imminent item after this first iteration can be of two types: \( (*, t) \) object or entity object. If it is a \( (*, t) \) object, the execution of the current local monitor is passed to either its lower or upper layer’ monitor depending on the source of the \( (*, t) \) message. Else, it is flowed to the next destination; i.e., a component or a Submodel object. For a Submodel object, the entity with a \( (*, t) \) message is transferred to a lower layer that will then be inserted into appropriate locations in the layer’s local monitor by its child’s first component. This monitor then executes and removes the imminent item from its Agenda.

Transferring the model execution to other layer’s local monitor implies that the layer contains lower next schedule time compared to the previous layer. The execution of this current layer’s local monitor continues until another \( (*, t) \) message is found in its Agenda. These processes are illustrated in Fig. 3.

Basically, the Monitor Delegation Mechanism coordinates the execution of events in a hierarchical DES model through these mechanisms:

1. Instruct Submodel objects to insert \( (*, t) \) to each local monitor. Execute the top layer’s monitor, followed by other layers.
2. Determine the imminent item type and the component that executes it.
3. (a) Flow the item to its next component in the same layer if the item is the type of entity and the component that executes it is a simulation component, or
(b) Transfer the item and a \((*, t)\) message if the item is the type of entity and the component that executes it is a *Submodel* object; see *Layer 1* Fig. 3. Insert them at appropriate locations in the layer’s *local monitor*. This process should be done by the child’s first component upon receiving the messages. Transfer the model execution to the layer’s monitor.

4. Retrieve and remove the next imminent item from the current layer’s *local monitor*. If the item is the type of \((*, t)\) message, transfer the monitor execution to the layer where the \((*, t)\) is from and then repeat this step 4. Else, repeat the step 2.

![Fig. 3: Monitor Delegation Mechanism](image)

### 3.2 Second Mechanism: Monitor Communication Mechanism

The *Monitor Communication Mechanism* differs from the *Monitor Delegation Mechanism* in two ways. First, \((*, t)\) messages are sent by a monitor, not by a *Submodel*. However, a *Submodel* object and the last simulation component in a layer still transfer entities to its lower and upper layer respectively. Second, for each iteration of simulation run monitors located above the source of a \((*, t)\) message must all be executed sequentially rather than transferring the execution of a monitor to a relevant layer. Such monitor communications through broadcasting \((*, t)\) messages demand the monitor implement the *Delegate Event Model* [see 9].

The purpose of broadcasting \((*, t)\) messages down to a certain layer where the \((*, t)\) comes from is to find the model’s lowest simulation time in all visited layers’ *Agendas*. For this, two types of iterations are needed. The first iteration broadcasts a \((*, t)\) message from the top layer until the lowest layer to consider the cases of *Source* components are located in the lowest layer or certain layers have their own types of entities. The second iteration onward only involves broadcasting a \((*, t)\) message until a relevant layer since any lowest next scheduled time below this layer definitely has a bigger value. This can be achieved by detecting the origin of a \((*, t_n)\) message.

The \((*, t_n)\) message is actually a \((*, t)\) message containing the latest value of the lowest next scheduled time. This value is collected during its traversal to the top layer. By broadcasting the \((*, t_n)\) message up from layer to layer, a parent layer acknowledges its child layer’s lowest next scheduled time. For example, *Layer 1* stores the lowest next scheduled time for *Layer 2*; *Layer 2* stores the lowest schedule time for the *Layer 3* and so on. Thus, the execution of the child layer is controlled by its parent monitor. The details of the *Monitor Communication Mechanism* are as follows:
1. Insert a default (*, t) message in the root Agenda whenever the model is first run.
2. Broadcast the (*, t) message from monitor to monitor in a sequence order (Layer 1, Layer 2, Layer 3, …) until it reaches the lowest monitor.
3. Execute the local monitor to coordinate events in the layer each time the layer receives the (*, t) message. For example, execute the local monitor in the Layer 2, followed by the Layer 3 and so on. Consequently, send the (*, t) message to lower monitors.
4. Once the (*, t) message reaches the lowest layer’s local monitor, retrieve the imminent item in its Agenda. Take its lowest scheduled time. Update the (*, t) message with a (*, t_n), where t_n is the lowest next scheduled time for the layer. Broadcast the (*, t_n) to its parent monitor; i.e., the local monitor in its upper layer. Note that the (*, t_n) message is supposed to traverse up to the top layer.
5. Once the (*, t_n) reaches its upper layer’s local monitor, insert the message into an appropriate location its Agenda based on the t_n value. Retrieve the imminent item from the Agenda. Broadcast a new (*, t_n) message (could be the previous (*, t_n) message if it is the imminent item) to its upper local monitor. Repeat these processes until the (*, t_n) reaches the top layer. This will guarantee that each layer stores its child’s lowest next scheduled time.
6. Once the (*, t_n) reaches and has been inserted to the top layer’s Agenda (i.e., root Agenda), execute the root monitor. If the imminent item in its Agenda is the type of (*, t_n), send another (*, t) message down to the layer where the (*, t_n) message is from. During this traversal, execute all visited layers’ Agendas to remove the (*, t_n) messages. Note that only the layer that has generated the (*, t_n) message will create a new event (i.e., flowing a relevant entity); other layers only remove the message from their Agendas. Broadcast another (*, t_n) message. Repeat step 5.
7. Stop the processes if the length of simulation time has been reached.

Fig. 4 traces a sample of Agendas based on the Monitor Communication Mechanism. The figure is split up to (a), (b) and (c); each one shows the Agendas at simulation time 0, 10 and 14 respectively.

At simulation time 0 (i.e., at initial run time), broadcasting a (*, t) message down to the lowest layer (i.e., Layer 4) is compulsory to find the lowest next scheduled time for the model. This example locates a Source component in the Layer 4. However, if it were located in other layers, broadcasting the (*, t) message down to the lowest layer would ensure the lowest next scheduled time is collected among the Agendas.

When the (*, t) message reaches the lowest layer, the (*, t) is converted to a (*, t=0); we assume that 0 is the first event; i.e., the creation of first entity. The (*, t=0) is then transferred up to the top layer since it
is the lowest next scheduled time in the whole hierarchy. After this first iteration, each time a \((*, t)\) goes down toward its origin layer, all the visited layers’ monitors need to execute their Agendas by removing their imminent item; i.e., the \((*, t=\text{value})\) message. For example, executing the monitors in Layer 2 and Layer 3 at simulation time 0 removes the \((*, t=0)\) from their Agendas. Only Layer 4 that contains a default entity (which is inserted by the Source component) removes \((*, t=0)\) and schedules a new event for the entity.

At simulation time 10, a \((*, t)\) message is broadcasted to Layer 4 from which the \((*, t=10)\) has come. During this \((*, t)\) broadcasting, all visited Agendas’ imminent items are removed (denoted by italic words). However, only Layer 4 schedules a new event for its imminent entity (denoted the bold words). Its new lowest scheduled time, i.e., \((*, t=16)\) is then transferred to Layer 3 and inserted to the layer’s Agenda (denoted by underlined words). This value is then compared with its lowest next scheduled time; i.e., \(t=14\). Since \(t=14\) is smaller than \(t=16\), the \((*, t=14)\) is transferred up to Layer 2. The processes of broadcasting a \((*, t)\) message, inserting it to an Agenda, comparing the value with the lowest value of the Agenda and re-broadcasting the smallest value are repeated until the top layer to ensure that all parent layers know their child layers’ next scheduled time.

At simulation time 14, traversing down until Layer 4 is not needed since its lowest next scheduled time is bigger than the lowest next scheduled time in Layer 3. Layer 3 then transfers a \((*, t=16)\) message to Layer 2 since \(t=16\) is smaller than \(t=22\). Layer 2 transfers a \((*, t=16)\) message to Layer 1 after comparing the value of \(t=16\) with \(t=18\). However, at simulation time 16, a \((*, t)\) will again need to traverse down to the Layer 4.

4. Implementation and Potential Problems

To broadcast \((*, t)\) messages between layers, each Submodel and Monitor class must be designed based on the Delegation Event Model pattern as implemented in simulation component classes [see 9]. Thus, the classes have to implement two methods; e.g., handleMsg(SimProcess, time) for inserting the messages to an Agenda and executeMsg(SimProcess) for executing the messages. Listing 1 shows samples of code for the Submodel and simulation classes.

```
function handleMsg (entityInstance:SimProcess, time:Number) {
    /* schedule the entity to its Agenda */
    entityInstance.delay(this, time)
}

function executeMsg (entityInstance:SimProcess) {
    /* if the entity is from the current layer*/
    if (entityInstance.fromLayer() == "current") {
        /* send the entity to its lower layer*/
        /child.handleMsg(entityInstance, 0)
    } else {
        /* send an extMsg if use the Delegation Monitor Mechanism*/
        extMsg = externalMsg.createNew( );
        /child.handleMsg(extMsg, 0)
        extMsg = externalMsg.createNew( );
        /* send the monitor execution to the Source of the extMsg monitor*/
        entityInstance.getSource( ).handleMsg(entityInstance, 0);
    } else {
        /* transfer the message with some delay to the next component */
        outport.handleMsg(entityInstance, delay);
    }
}
```

(a) Submodel Class Definition

```
function handleMsg (entityInstance:SimProcess, time:Number) {
    /* schedule the entity to its Agenda */
    entityInstance.delay(this, time)
}

function executeMsg (entityInstance:SimProcess) {
    if (entityInstance.typeOf ExternalMsg) {
        /* transfer the monitor execution to the Source of the extMsg monitor*/
        entityInstance.getSource( ).handleMsg(entityInstance, 0);
    } else {
        /* transfer the message with some delay to the next component */
        outport.handleMsg(entityInstance, delay);
    }
}
```

(b) Simulation Class Definition

Listing 1: Class Definition

We are currently implementing the mechanisms in our DES tools [8, 9] to support hierarchical DES models. The implementation is based on one keyframe with a number of main movie clips. Each clip represents a layer that contains many other movie clips; e.g., simulation and visualization components, etc. The use of only one keyframe ensures that we can access simulation components from other movie clips and preserve the lifecycles of entities and \((*, t)\) messages. To prevent the clumsiness of many main movie clips on a stage, the main movie clips can be hidden or displayed depends on learners’ wishes.

Our target is to use Flash’s keyframes to form layers. Unfortunately, Flash treats each keyframe as a totally new program without allowing communications between objects in other keyframes. It only allows a basic transition between keyframes; i.e., moving an execution point from keyframe to keyframe. This hinders us from passing entities and messages to other keyframes. It is good if it follows other programming
language approaches, e.g., Microsoft Visual Basic that allows the use of `FormName.ObjectName.Property` to access objects in other forms and allows objects to freely be passed to other forms. We are finding a method to solve this problem.

5. Conclusions

In this paper, we have proposed two mechanisms for coordinating event executions in hierarchical DES models; i.e., the Monitor Delegation Mechanism and the Monitor Communication Mechanism. Both mechanisms are easy to implement if we design simulation components, SubModel objects and simulation monitors based on the Delegation Event Model. Implementing these mechanisms in the Flash environment will produce attractive and interactive DES hierarchical models that are valuable for representing different levels of model structures and displaying model behaviours.

We are currently extending our work to provide attractive and interactive tools that support the construction of flat and hierarchical DES models. Using this platform, model designers can create a sub-model by only dragging and dropping a window to represent a layer where all its structures and visualization are constructed. The results of this will be reported soon.

References