A MESSAGE-BASED DISCRETE EVENT SIMULATION ARCHITECTURE

David Krahl
J. Steven Lamperti

Imagine That, Inc.
6830 Via Del Oro, Suite 230
San Jose, CA 95119, U.S.A..

ABSTRACT

This paper presents a message-based discrete event simulation architecture. It will examine each of the different types of messages used to schedule events, transfer items through the model, and resolve logic issues. Additional information is also presented on the supporting data structures.

1 INTRODUCTION

Simulation software developers are faced with the never-ending challenge of devising software which is easier to use and more powerful at the same time. Presented here is an architecture that, while more complex internally than traditional discrete event architectures, eliminates many of their counter-intuitive actions while generally requiring fewer modeling components. This has been achieved without sacrificing model fidelity or accuracy.

The architecture discussed in this paper is based on a messaging system. Instead of a centrally located simulation program executing subroutines based on simulation data, this architecture sends messages from one simulation "block" (the basic simulation modeling construct) to another. The central simulation engine performs only event scheduling and selection. The bulk of the simulation execution is performed by blocks sending messages to one another.

A glossary is included at the end of this paper to provide the reader with definitions for the terms used here.

2 SIMULATION STRUCTURE

To understand the details of the messaging system used in this discrete event modeling architecture, some understanding of both the simulation engine and the underlying data structures and event handling are necessary.

Each of the blocks in this architecture is composed of an icon, dialog, simulation data, connectors (which pass data between blocks), and code (which defines the behavior of the block). In general there are two types of discrete event blocks: blocks that hold items (residence blocks) and blocks that pass items through without holding them (passing blocks). Residence blocks are able to contain items for some duration of simulation time. Examples of residence blocks are queues or delays. Passing blocks implement modeling operations that are not time based. Examples include setting an attribute or selecting a path.

2.1 Simulation Engine

If the modeler places some blocks on a worksheet and issues the run command, the program will step through the specified time interval, sending a simulation message to each block in the model on each time step. The time duration of the simulation will be divided into even steps in this fashion. This is essentially a standard continuous simulation mechanism.

To provide a true discrete event time clock, the user adds an Executive block to the model worksheet. This block does two things: first, it maintains a data structure of information about the items in the model; second, it takes control of the time clock from the application, scheduling events and moving the clock forward to the appropriate time for the next event.

2.2 Data Structures

The data structures maintained by the Executive are essentially arrays of data. They contain all the respective information about the items that are traveling through the model. Items are indexed in these arrays in the order that they are created, such that the first item created in the model gets array index value 1. When items travel from one block to the next, the blocks pass the array index value of the item. In the code of the blocks this is usually maintained in a variable called ItemIndex.

There are several dynamic arrays of item data maintained by the Executive block including:

ItemArrayR	Real (value and priority)
ItemArrayT	Timer (cycle time)
ItemArrayI	Integer (batching and free row flag)
ItemArrayC	Costing information

The Executive block also manages a global array structure called "_AttribValues" containing attribute information for each of the items in the model.

The appendix discusses dynamic arrays and global arrays in more detail.

2.3 Messaging

The next element of the architecture is the mechanism that the blocks use to send messages to each other. A message chain is started either by the system or by the Executive sending a *simulate message* to a block. From there, the blocks propagate the messages to other blocks through their connectors. Each block calls the functions 'SendMsgToInputs' and 'SendMsgToOutputs' to send messages to other blocks. Any block that receives a message will process it in one of its message handlers, and may send on messages of its own.

As discussed later in the paper, the messaging architecture consists of basic item messaging, value connector messages, blocked and query messages, and the notify message.

2.4 Event Handling

Two event list arrays are used to store future events in the model. The first contains the event times; the second contains the number of the block that posted the event. At the start of the simulation, each resident block that posts events is allocated a position in the event list arrays. Each event-posting block generates an event time by adding the current time to the time required to complete processing of the current item. This time is placed in the block's event list position. At the start of a simulation event, the event list is searched to find the next event time. A third dynamic array (essentially a "current event list") is used to store all of the block numbers that have posted an event at the next event time. The simulation clock is then advanced to the next event time and a message is sent, one at a time, to each block within the current event list. Each block then searches its own internal list for the item with the current event time and attempts to push the item out into the next block in the process.

Some non-event posting resident blocks, such as queues and resources, also attempt to move items at event times. The purpose of this type of block is to pull in items and hold them until there is a downstream capacity. If, at any given event, no items have been pulled in or pushed out, the block will go to sleep, ignoring any simulation events. It will "wake up" when an item is either pushed in or pulled out and will attempt to process additional items.

3 BASIC ITEM MESSAGING

The actual moving of items between blocks is done through a messaging communication structure using item connectors and connections. This messaging system allows modelers to place blocks in a more intuitive sequence. Common restrictions such as requiring a queue to be immediately before an activity are removed.

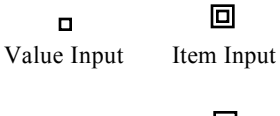

Figure 1: Connector Types

Depending on how the message sequence is initiated, items can be either pushed or pulled through the model. It is easiest to illustrate this with a series of simple examples. Figure 2 shows a first-in-first-out queue (Queue, FIFO) block connected to an activity with a single item capacity (Activity, Delay).

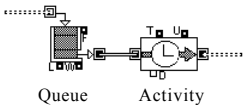

Figure 2: Queue Connected to an Activity Delay

3.1 Push Mechanism

When an item is pushed through the model, the upstream blocks (in this case, the queue) try to push their items out into any downstream blocks.

For example, assume that an item has just arrived at the queue and the queue is attempting to pass that item along into the activity. The first action is for the queue to send a *wants message* through its output connector to the activity. This indicates that the queue wants to send an item to the activity.

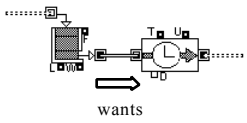

Figure 3: Wants Message Sent to Activity

If the activity is not currently processing an item (idle), it will return a *needs value* to the queue, indicating that the item can be accepted.

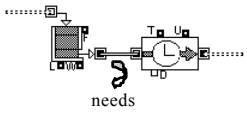

Figure 4: Needs Value Returned From Activity

If a *needs value* has been returned from the activity, the queue then sends a *needs message* to the activity. At this point the item would be committed to moving from the queue to the activity.

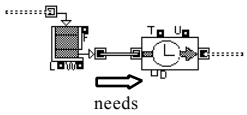

Figure 5: Needs Message Sent to Activity

However, if the activity is currently busy processing another item, it will return a *rejects value*. If the item is rejected, the message sequence will be terminated.

The item is logically moved from one block to the next by transferring its item index over the connection between the blocks. To do this, the queue sets its output connector value to the item index. When a block sets its connector value to the item index, the connector value of any connected blocks will automatically be set to that same item index value. Since the output connector of the queue is connected to the input connector of the activity, the two connectors will share the item index value. The activity then sets its input connector to a negative number and sends a *taken message* back to the queue to indicate that the item has successfully moved. In response to the *taken message*, the queue will update any internal statistics related to the departure of the item.

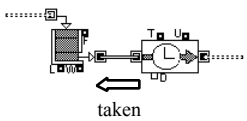

Figure 6: Taken Message Sent to Queue

3.2 Pull Mechanism

In addition to being pushed, as in the preceding example, items can also be pulled through the model. If they have remaining capacity, downstream blocks try to pull items into their inputs. For example, when the activity finishes processing the item, it will attempt to pull in another item. To do this, the activity first sends a *wants message* to the queue indicating that it is requesting an item.

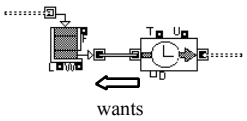

Figure 7: Wants Message Sent From Activity

If an item is available in the queue, the queue's output connector will be assigned to that item's index value. The activity will pull in the item and then send a *taken message* back to the queue. If an item is not available in the queue, a *rejects value* will be returned and the message chain will be terminated.

Both of the blocks in the above examples are residence blocks and can hold items for some period of time. Passing blocks do not have this ability and must pass the item through in 0 time.

Figure 8 shows a block reading an attribute value that will be used to set the delay of the activity. This Get Attribute block does not affect the messaging communication between the queue and the activity. Since it is a passing block, it transfers the initial *wants* and *needs messages* between the queue and activity. Thus, any number of passing blocks can be between any blocks that can hold items. Once the item moves into a passing block, it will send a *taken message* to the upstream block, as shown in Figure 6.

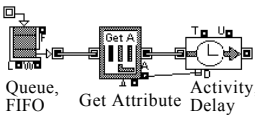

Figure 8: Passing Block Between Queue and Activity

4 VALUE CONNECTOR MESSAGES

In addition to item connectors, value connectors (see Figure 1) are used to relay model information. Value connectors pass a single number from one block to another. Examples include a value output such as length of a queue or an input such as a delay time. The use of these connectors allows the combining of blocks which

perform numerical calculations (referred to as continuous or generic blocks) to provide a control structure and logical information for the discrete event blocks. This provides additional modeling flexibility without requiring user programming or complicated interfaces.

In Figure 9, two random variables (Input Random Number blocks) are added together to specify the delay for an activity. Whenever an item arrives to the activity, the Input Random Number and Add blocks will need to be recalculated. Whenever a discrete event block detects a condition where an update to the value of a connector is needed (in this case, an item arriving to the activity), it sends a message out its value connector (in this case, value input connector "D" in Figure 9).

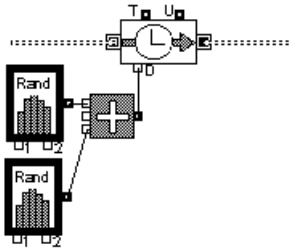

Figure 9: The Sum of Two Random Variables Specifying an Activity's Delay

4.1 Message Emulation

A default feature for generic blocks called "message emulation" will propagate the message throughout all of the generic blocks used in the calculation. Message emulation is used whenever the generic block contains no message handlers for any of its connectors. In that case, the *on simulate* message handler is used as a default message handler for all connectors.

Figure 10 illustrates an On Simulate message handler assigning the output connector (Con1Out) to the input connector (Con1In) plus one. Whenever either connector receives a message, a message will be sent out the other connector through message emulation. This will cause all connected blocks to execute their On Simulate message handlers, propagating messages where appropriate.

```
On Simulate
{
Con1Out = Con1In + 1;
}
```

Figure 10: On Simulate Message Handler

4.2 Explicit Connector Messages

Overriding message emulation gives the block programmer more flexibility in the behavior of the block. If a generic block has one or more connector message handlers, message emulation is automatically disabled and the connector message handlers are used to perform the calculation instead. In this case, the generic block must send out messages to other value input and output connectors explicitly. For example, a message must be sent out the output connector whenever a message is received on the input connector, and a message must be sent out the input connector whenever a message is received on the output connector.

Figure 11 shows the code necessary to use message handlers to make a block behave equivalently to the message emulation used in Figure 10. Both examples will perform identically in model operation. Because message handlers have been explicitly specified for the connectors in Figure 11, message emulation has been automatically disabled.

```
On Con1In
{
Con1Out = Con1In+1;
SendMsgToInputs(Con1Out);
}

On Con1Out
{
SendMsgToOutputs(Con1In);
Con1Out = Con1In+1;
}

On Simulate
{
}
```

Figure 11: Message Handlers Used for Value Connections

5 BLOCKED AND QUERY MESSAGES

One of the more complex message communication subsystems in this architecture is the communication between blocks when a block needs information about an item that has not yet arrived. This occurs when an item needs to know if it can move downstream before it starts to move, or when it needs to determine which path it will take before it gets to the block that has the paths.

Traditional discrete event architectures would require dummy resources to overcome these problems. This architecture, however, is able to predict the path of an item before it moves into the decision logic and is able

to block through decision points without any additional modeling components.

For example, in Figure 12 the model section reads an attribute on an item (Get Attribute) and selects one of two paths for that item to follow (Select DE Output) based on the value of the attribute. The Get Attribute block will read the value of the attribute on the item, but its default behavior would be to not read the attribute value until the item has entered the block. In this case, this could cause a problem, as the Select DE Output block may be blocked down the path that the item will need to travel. If the Select DE Output is blocked, and the item has to move into the Get Attribute block to present its attribute value, then the item will be stuck in the Get Attribute block. And since attribute-manipulating blocks are only meant to pass, not hold, items the Queue will understate the number of items available.

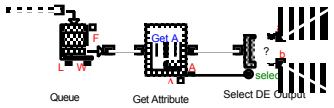

Figure 12: Select Controlled by a Get Attribute

The solution to this problem is to set up the Get Attribute block to be aware that it is in this situation. The Get Attribute block can then look upstream to see what the next item coming along will be, and not pull in the item until the downstream path is free.

5.1 Blocked Messages

The first part of this process involves sending a *blocked message* downstream from the Get Attribute to see if there are any blocks that could cause this situation. The Get Attribute does this the first time it gets an incoming message (i.e. the first time the Select DE Output block requests an attribute value from the *A* connector.) The Get Attribute sends a *blocked* message from its item output connector in its *on AttribOut* message handler. (The *on AttribOut* message handler is called when the *A* connector on the Get Attribute block gets a message.) If a TRUE value is returned in response to the message, then the Get Attribute block sets a flag which records that it is blocked.

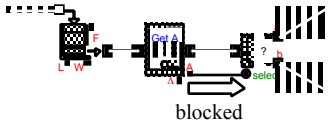

Figure 13: Blocked Message Sent to Select Output Block

5.2 Query Messages

If it has been determined that blocking can occur, each time there is a request for an attribute value, the Get Attribute block sends a *query message* upstream. This message is essentially a request for information about the next item that is available. The message will be propagated upstream by the blocks until it reaches a block that can contain items, such as a queue.

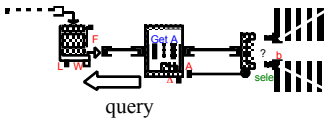

Figure 14: Query Message Sent to Queue

The block responding to the query message will check to see what the item index of the next item to be released will be and will return that value to the querying block. The Get Attribute block will then access the global array *_AttribValues* to get the attribute value for that item. From this point on, each time that an attribute value is requested from the Get Attribute block, it will send out the *query message* and check the attribute value of the next item. It will not need to resend the *blocked message* again.

The system architecture will notify users if there are any logical ambiguities that will not allow the model to operate properly. When this occurs, an error message is issued that recommends a course of action that will resolve the ambiguity.

6 NOTIFY MESSAGE

The final message used by this system is the *notify message*. This message is used to notify other blocks that an item has just passed by a specified point in the model. A special *sensor* connector receives this message. Sensor connectors do not pass items, they monitor the message stream, processing only the *notify message*. Only a few blocks have sensor connectors, although all of the blocks that process items will send the *notify message* through their item output connector. The following situation requires a notify message.

In Figure 15, a block (Gate) uses its sensor connector to limit the number of items in the section of the model between its output connector and the activity's output connector.

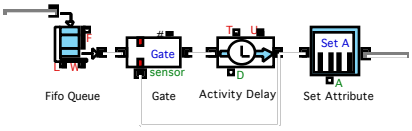

Figure 15: Situation That Requires a Notify Message.

As an item travels from the activity to the Set Attribute, a *taken message* will be sent to the activity. In response to this message, the activity will send out the *notify message*. The normal item input connectors in the blocks will ignore the *notify message*, but sensor connectors will respond to it and start processing information about the item. In Figure 15, when a *notify message* is received by the sensor connector, the Gate block knows that another item has passed by and can allow an additional item into the model section.

7 CONCLUSIONS

This system has been found to ease the model building process by allowing the modeler to put the model together in an intuitive manner. It has been implemented in the Extend™ family of simulation software products. Feedback from end users indicates that the architecture is successful in shortening the simulation learning curve while providing a powerful modeling environment. In addition, source code is provided to the user should they want to modify or study the system's behavior. Finally, this architecture is in use in over 10,000 locations and has proved itself as a powerful, easy to use simulation system.

APPENDIX:

Block: A modeling component that is selected from a library and placed into a simulation being constructed. A block is composed of an icon, dialog, simulation data, connectors (which pass data between blocks), and code (which defines the behavior of the block). Some blocks hold items for a period of time (residence blocks) while others merely pass items through (passing blocks). Typical blocks would include FIFO Queue, Activity Delay, and Executive.

Connector: A point on a block where a connection can be made. Connectors are either inputs or outputs and are used to transfer values or items between blocks.

Connection: The link between two blocks that transmits data. Connections will be drawn on the model/worksheet as single lines for values and as double lines for item connections.

Dynamic array: An array which has a variable size. The size of the first dimension can be changed at any time with a function call. Dynamic arrays can be passed from block to block. These structures are used to store the majority of the information about the items. Dynamic arrays can have up to five dimensions (row, column, depth …).

Global Array: An array which is global to the entire model. The number of columns is defined when the array is created. The number of rows can be resized at any time with a function call. Global arrays are always two-dimensional. For example, the global array "_AttributeValues" is used to store the values of the item attributes in the simulation. When the simulation initializes, the number of columns is specified as the number of attribute values in the model. The number of columns is resized as the number of items in the model increases.

Item: A conceptual entity or object moving through the model. Items will have various information associated with them, such as priorities, attributes, or values.

Library: A collection of blocks (modeling components). Some standard libraries are: the Discrete Event, Generic, and Plotter libraries.

Message: Signals sent to blocks in the model. Each block will have a message handler associated with a given message and will execute the code in the handler when the given message is received. In general there are two types of messages that blocks will receive. The first is system messages that are sent out by the program, such as InitSim, Simulate, and EndSim. The second type is connector messages, such as ItemIn and ItemOut. These are sent from one block to another through the block's connectors, and are received in message handlers with the same names as the connectors.

Message Handler: A section of code within a block that is executed when the given message is sent to the block. A message handler has the form 'on XXX', where XXX is the name of the message. An example of an 'On Simulate' message handler is:

```
On Simulate
{
// code that is to be executed on each
simulate step
}
```

Model: A collection of blocks connected together to form the logical basis of the system being simulated.

Worksheet: The screen or window containing the model.

AUTHOR BIOGRAPHIES:

DAVID KRAHL is the Director of Technical Services at Imagine That, Inc. Mr. Krahl is responsible for block development and technical support. He received a MS in Project and Systems Management in 1996 from Golden Gate University and a BS from the Rochester Institute of Technology in 1986.

J. STEVEN LAMPERTI is the Director of Research and Development at Imagine That, Inc. Mr. Lamperti is responsible for Extend application development and is the designer of this discrete event architecture. He received his BS in Computer Science from the University of Vermont in 1985.