Diagnostics Software for Concurrent Processing Computer Systems

Robert L. Jones III
Old Dominion University

A GENERIC OBJECT-ORIENTED SERVER MODEL

by

Brian Douglas Jones
B.S. December 1997, Old Dominion University
A.S. December 1995, Tidewater Community College

A Thesis Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirement for the Degree of

MASTER OF SCIENCE

COMPUTER ENGINEERING
May 2000

Approved by:

[illegible]
James Leathrum (Director)

[illegible]
John W. Stoughton (Member)

Roland Mielke (Member)

ABSTRACT

A GENERIC OBJECT-ORIENTED SERVER MODEL

Brian Douglas Jones
Old Dominion University, 1999
Director: Dr. James Leathrum

The purpose of this paper is to introduce a generic, object-oriented model for the simulation of networks of queues. Generic simulation modeling provides robust ways of laying out processes in stochastic, event driven simulations. The approach taken is to define a methodology that will be implementation independent, termed the Generic Server Simulation Model (GSSM); thus, leaving the implementers to choose the best (most suited for their use) means of implementation. GSSM uses many object-oriented concepts to provide its basic structure and has in its design the ability to take advantage of these object-oriented concepts, including reuse of (existing) code. GSSM constructs provide a natural way to easily build network (nodal) models, allowing these constructs to transform easily into a graphical representation. The key use of GSSM will be in its adaptability or mutability. By design, it handles easily and naturally the insertion and deletion of processes, as well as increases or decreases in the number of object types serviced by these processes. The model is demonstrated using a port simulation.

I dedicate this paper and work to a special friend who did not let me forget that I had better finish this paper.

ACKNOWLEDGMENTS

I thank Dr. Leathrum for his collaboration and advice. I also thank Frank Palathingal for his work on the implementation of GSSM project.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A common problem encountered in the simulation of processes is the need for the serving of queues given available resources. When developing a simulation as a network of queues, while the individual servers, queue implementations, and resource pools may be distinctly different, the basic model of the individual processes is just an instance of an abstract model. Frequently, the individual processes are developed totally independently without this concept in mind. However, when faced with the task of easily modifying the network of queues to insert a new process, one does not relish the possibility of having to generate a new model each time and grappling with the interconnection of the models. While the concept of modeling networks of queues is well understood (6,7,8,11)[1], frequently the support at the design and implementation level does not encourage the use of sound modeling constructs. Languages such as ModSim III (3) may provide support in the form of predefined queue constructs, though not in the form of a complete server model. The Generic Server Simulation Model (GSSM) attempts to provide a useful model for the easy development of complex networks of queues and to modify those networks efficiently.

The Virginia Modeling, Analysis, and Simulation Center (VMASC) originally designed GSSM as a mechanism to easily incorporate commercial conflict points into a purely military port model called PORTSIM (9). PORTSIM was originally designed to model the flow of military traffic through a commercial port; however, it ignored the ongoing commercial traffic. To inject new conflict points that resulted from commercial

---

[1]The thesis format is based on *IIE Transactions* sumittal guidelines located at http://www.ieee.org/organizations/pubs/ieeetran.zip and thesis manual guildlines.

traffic flows not apparent in military only traffic (such as intersections), an appropriate modeling construct was necessary with an appropriate template for the easy development of new server models. The resulting model is applicable to other server models as well, such as manufacturing and computer systems.

Thesis Organization

This paper is organized into five chapters: the introduction, background, theory and implementation, example, and conclusion. The background chapter describes the current technologies from which this thesis draws its theory. From the theory, a specific example and implementation are derived. The conclusion is a brief summary of what this paper is trying to accomplish and what still needs to be accomplished to make GSSM viable.

II. BACKGROUND

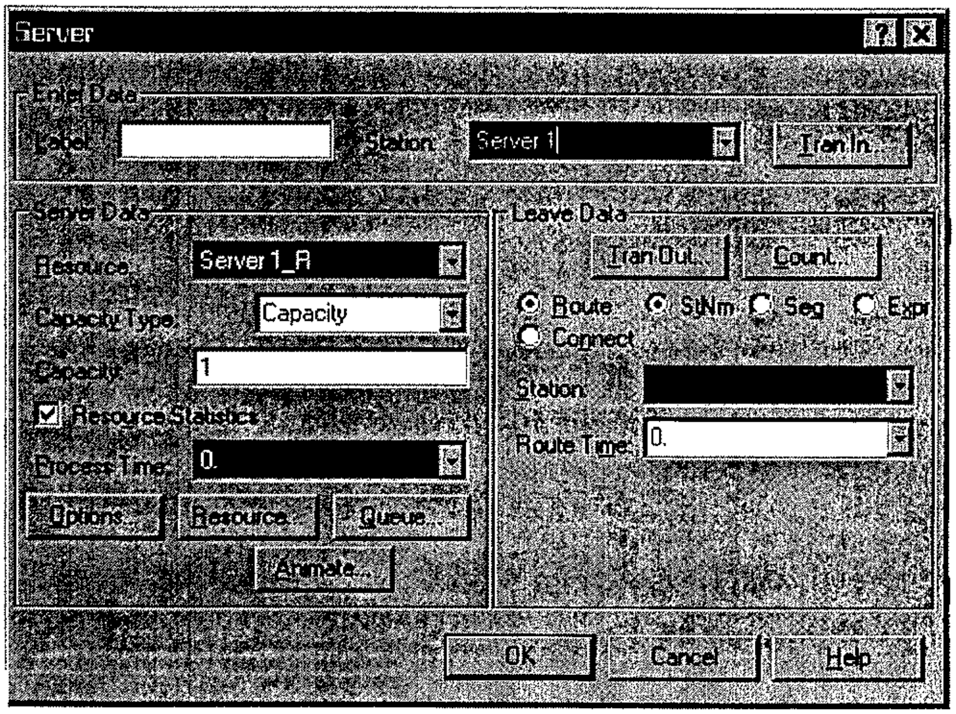
With all the emphasis today in the software engineering and simulation societies on object-oriented design, there is little in the way of a reevaluation of the basic designs intrinsic to object-oriented simulations. What is missing are the intimate relationships found within the basic foundations of simulation. While some simulation software-makers present nice, well-known object based components (e.g., resources, queues, and servers), there seems to be an insufficient, unified endeavor to present a common fundamental hierarchy between them. While independent implementations abound, with one of them described here, they seem to skip over the theory and implementation of their server/queuing models (perhaps for proprietary reasons). So, presented here are popular independent modeling constructs taken apart to find the commonalties of basic server-queuing systems. This will consist of an analysis of a popular, plug-n-play simulation tool like System Modeling's ARENA (1,5) and then applying object-oriented simulation modeling techniques (4) to develop an object-oriented server-queuing model that will allow for generic plug-n-play capabilities.

ARENA

System Modeling's ARENA 3.0 is a wonderful, complex tool that simplifies the creation of event-driven simulations. It is an incorporation of many different tools. These tools include complex data analysis, high-level and special purpose modeling components, and animation. Of particular interest to this paper are the high-level components. While ARENA's underlining language is SIMAN, a flexible and a general-purpose simulation language, it is not in essence an object-oriented language, even though it supports many high-level components and constructs (1,5).

ARENA takes these high-level components a step further by combining them to make for even a higher-level of components or modules called jointly the Common Templates. Example Common Templates include the Arrive, Depart, Server, Inspector, Advanced Server, Processor, Queue, Resource, Sets, Statistics, and Variables modules for example. These modules can be strung together by simply dragging and dropping the components, as is common in most windowing environments. By clicking on one of these components, the user can modify and customize specific features of each. Thus, it simplifies the designing of the model while still meeting the needs of the modeler. For example, the user can modify the arrival conditions of entities entering the system. The user gets a dialog box to choose the interarrival distribution from an ample supply of predefined distributions and then supplies the parameters, like the mean or standard deviation. Furthermore, the user can choose the next "station" to which the new entities will go either from a drop-down or by drawing a route from the Arrive component to perhaps another "station."

Of particular interest are the Server, Inspector, Advanced Server, and the Processor modules, all of which are just variations of each other. We concentrate on the Server module. The following is an abbreviated view of the elements of the Server module derived from ARENA 3.0.



Label:
Station: Server 1
Resource: Server 1_R
Capacity Type: Capacity
Capacity: 1
[✓] Resource Statistics
Process Time: 0.
[● Route] [● StNm] Seq Expr Connect
Station:
Route Time: 0.

Figure 1. ARENA's Server Dialog Box (1)

While this paper is not to be taken as a tutorial or guide to ARENA, it is necessary to look at the individual elements of the Server to get a complete picture of the inner workings.

Note the components of the server in Figure 1:

- a label,
- a Resource,
- a process time,
- a Queue,
- a transfer out Station,
- a Route Time,

- and possibly others (depends on which options are chosen from dialog).

Again note that ARENA allows for "sets" of Stations, Queues, and Resources, which means that the each member of the set can be associated or chosen depending upon state of the system or other aspects (e.g., the entity in question). These components are freely interchangeable and allow adequate modeling flexibility for most users.

ARENA presents a solid framework for defining a server model. The issue is to abstract this model so that a new model can be developed for supporting simulation development in simulation languages. The key lies in the effective use of object-oriented modeling (1,4).

Visual Simulation Environment

The Visual Simulation Environment (VSE) is a graphical object-oriented simulation and model-building tool. The tool provides a high-level graphical user interface (GUI) that allow the modelers to drag and drop reusable objects into and around the development workspace. By double-clicking on the objects or graphical components, the modeler can transverse through the class hierarchy. The VSE tool allows the user the ability to view the multi-model hierarchy during runtime; thus, the user can see the higher-level simulation and then hone down into the graphical display of the underlining sub-models. Being a general-purpose tool, the user can create libraries of reusable objects derived from a basic library of objects provided by VSE. VSE like ARENA provides customizable animation and statistical analysis tools and components. Unlike ARENA, it is a fully object-oriented tool (2).

Justification for an Object-Oriented Model

There are some very good reasons for an object-oriented server model. Many of these reasons explain why ARENA is not the perfect tool to accommodate a simulationist's needs.

The first benefit of using a language like C++ (or object-oriented simulation languages like ModSim III (3) or C++SIM (10)) is access to legacy code. With all the legacy code around, programmers are now just placing within object wrappers to maintain the previous investments. This task would be difficult to accomplish with ARENA.

Second, although ARENA is very flexible, it is not perfectly flexible. There are limited options from the dialog boxes. While this is quite appropriate for beginning modelers or particular high-level models, its constructs are limiting to the advanced modeler. An example is the difficulty in adapting ARENA to participate in a distributed environment. Object-oriented languages can provide almost the same plug-n-play features without being limiting, if the modelers design with this in mind. The use of inheritance can provide the mechanisms to facilitate the plug-n-play features of object-oriented simulations. Take for example the Server module above. If a modeler needs a special queuing arrangement that ARENA does not support, the modeler is practically out of luck, possibly requiring development of SIMAN code. Even though ARENA has some advanced features, i.e., calls to external applications, such as Visual Basic, it does not support them directly. If modeling with the proposed object-oriented server design, the programmer could derive the special queue from the base queue object and implement it anyway needed.

Third, portability is another issue that must be considered. Object-oriented modeling goes beyond a language or a program, meaning that once a system is modeled in an object-oriented manner, the implementation can be in C++, Java (although it does not support multiple inheritance), ModSim III (3), C++Sim (10), etc.

Summary

While ARENA provides an attractive environment for the rapid development of discrete-event simulations, it has limitations imposed by the interface structure. Development in a programming language removes these limitations but is more difficult due to the lack of simulation construct support. As the server model is one of the most common simulation constructs, an object-oriented server model is presented to empower the programmer with capabilities found in tools such as ARENA, even allowing him to go beyond those capabilities.

III. OBJECT-ORIENTED SERVER MODELING

In this chapter, the theory and the implementation of GSSM is discussed. The derivation, creation, and destruction of a simple GSSM model is described.

Hierachial Structure for a Server Model

The goals of developing an object-oriented model for servers are to achieve:

- well defined server components,
- reusable components, and
- flexibility available in general programming.

The object-oriented modeling features that support these goals are described below.

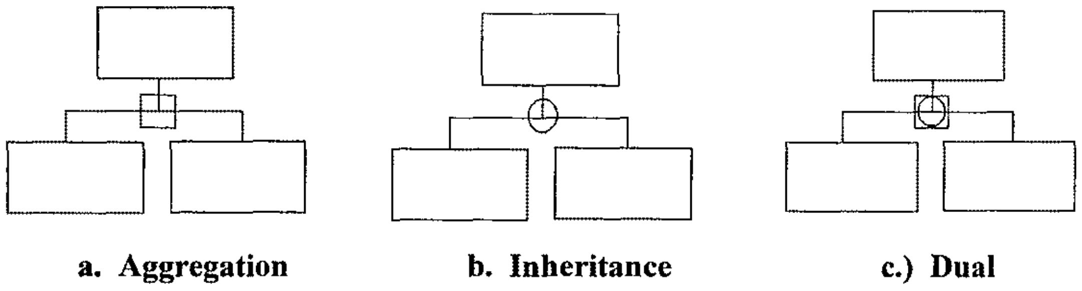


Figure 2. The Object-Hierarchy Model

Figures 2a, 2b, and 2c are the depictions of the graphical forms used to represent typical object hierarchies (4). These object hierarchies are defined by:

- aggregation - the composite of all sub-objects,
- inheritance - a form of generalization, and
- dual – a combination of the previous two.

Using the object hierarchies defined above, Figure 3 represents an object model of the server model described for ARENA.

Figure 3 illustrates some of the components that make the ARENA's Server subsystems, as well as some of the interrelations and object hierarchies. Note that there is more than one way to describe ARENA's components in an object-oriented manor, and the choice depiction found in Figure 3 is the object hierarchy of GSSM.

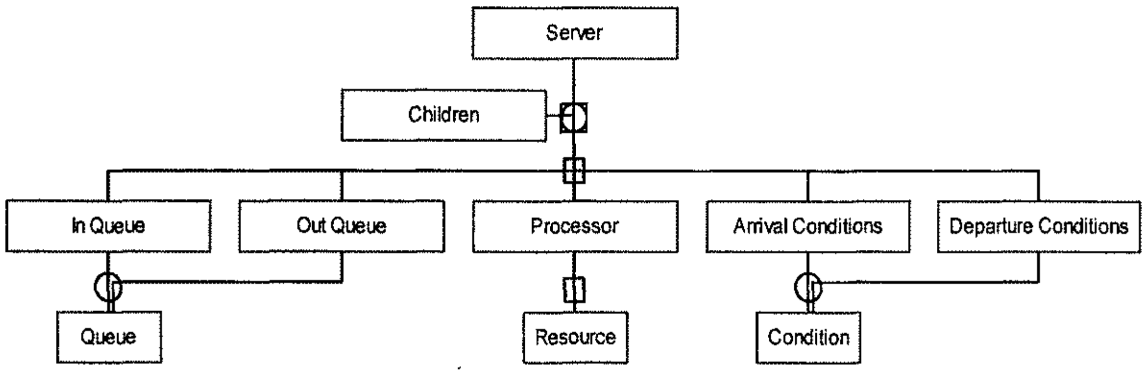


Figure 3. The Server-Queue Model

Left out of the diagram are the attributes, methods, and ARENA's "Sets" module. In dealing with Sets, there are at least two possible approaches. The first approach is to consider the Server Object to be composed of arrays of objects (queues, resources, etc.). The second approach is with linked lists of objects. Either way, the code can iterate through the set.

Certain items in Figure 3 need further discussion. The Arrival Conditions in ARENA's Server module are simply the mechanisms that choose one of the queues from the set of queues upon the arrival of the entity to the Server module. The Departure Conditions are dependent on the type of output routing that is chosen from the Server's dialog box. If a Transport is used to transport the entity from one Server to the next and the transport is not available, the entity is then queued in the Output Queue. This decision is determined by the Departure Conditions.

The Children object is another object that needs explanation. In ARENA, there are a couple of options as to how the routing between components is accomplished. Routing can be done by physically connecting components together or by specifying what station is next. The Children object represents a linked list of the next "stations," which can be Servers or one of the other components.

The Generic Server Simulation Model (GSSM)

The mechanism behind GSSM is based upon two object constructs: the node and the link. The node encapsulates the basic server model while the link provides a mechanism to route entities between nodes. Both of these constructs, along with some modeling philosophies (constraints), make up the whole of GSSM. The nodes and links connect to form a network of queues. A specific application may use some derived form of the abstract model. The model's design is robust to allow the definition of any object types. The model is presented in an abstract sense, but the basic template for the development of new server models is realized in ModSim III later.

The Basic Components of the Node

The node models the servicing of an object by available resources. A node consists of Arrival Conditions, input and output Queues, a Process Handler (Server), and Departure Conditions as illustrated in Figure 4. A discussion of each is presented.

Arrival Conditions: Arrival Conditions are the set of conditions mapping an object arriving at a node to a queue where it will be placed. The conditions are based upon the arriving object's type and state that define the set of resources required by the object.

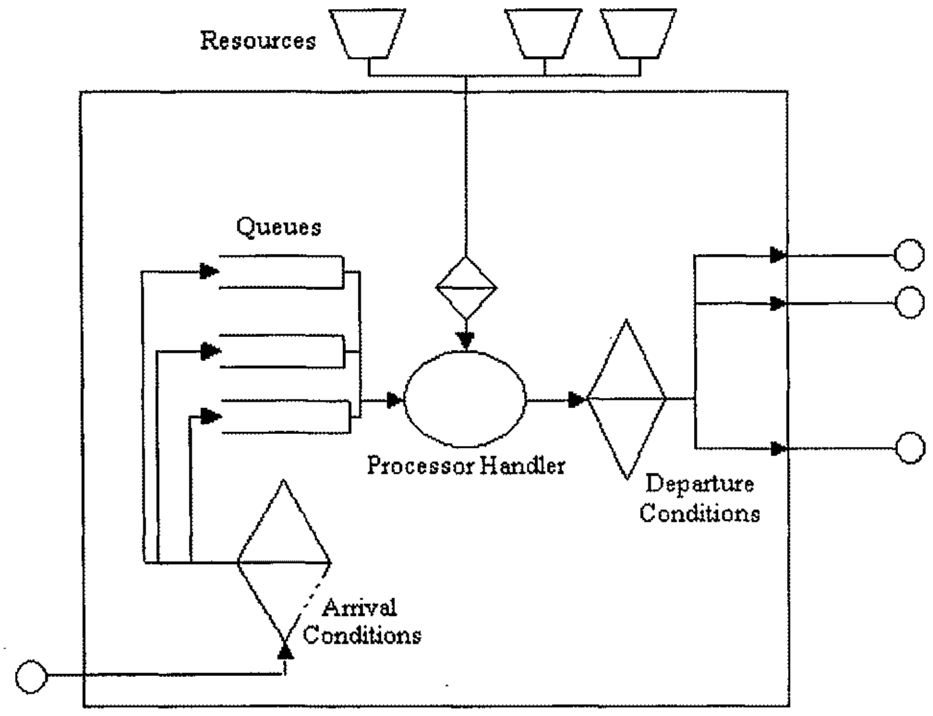


Figure 4. The Components of the Node

Input Queues: A Queue is one of the queues in the set of queues associated with a node. Anytime an arriving object enters a node, its Queue is determined by the Arrival Conditions. Once placed in the queue, the arriving object remains there until the server takes the object out. The object can leave the queue only if the server is IDLE and if the resource, or set of resources with which it associates, is FREE.

Process Handler: A Process Handler services the arriving object. This service could be the allocation/release or association/disassociation of resources to the arriving object. Typically, a time period is defined for which the simulation waits to simulate the processing of a specific operation for the arriving object during this stage. The Process Handler may serve each arriving object distinctly depending on the Arrival Conditions and the properties of the arriving object. The processing parameters could also depend on the properties of the particular resource that is allocated to that object for service.

Departure Conditions: A Departure Condition is a determination process that uses the object's type and state to determine to which exit point to send the object. Note that the processing done at the Process Handler may involve the creation of new objects, and these newly created objects are then sent to the Departure Condition to determine where to go next. Thus, the object type that comes into a node may not be the same as the one that exits the node.

Output Queues: In many cases, there might be a need for storing the object in the node after processing it and passing it through the departure conditions. Typically, this case would arise if the transfer of entities to the next node or link were constrained by some conditions. Then, the entity would have to wait in the current node until the constraint has been satisfied before it can go to the next activity in the simulation. Thus, the output Queues serves the purpose of temporary storage of entities before they are forwarded.

Resources: Every server in a node may be associated with resources or pools of resources. It would be inaccurate to say that a resource belongs to a node if other nodes could also reserve those resources. Hence, it is not a good practice to build the resources within the node and constrain it as a private attribute of a node. Hence, a resource is envisioned as a floating object that is acquired by a server if it is available and released back into the pool after the service has been carried out.

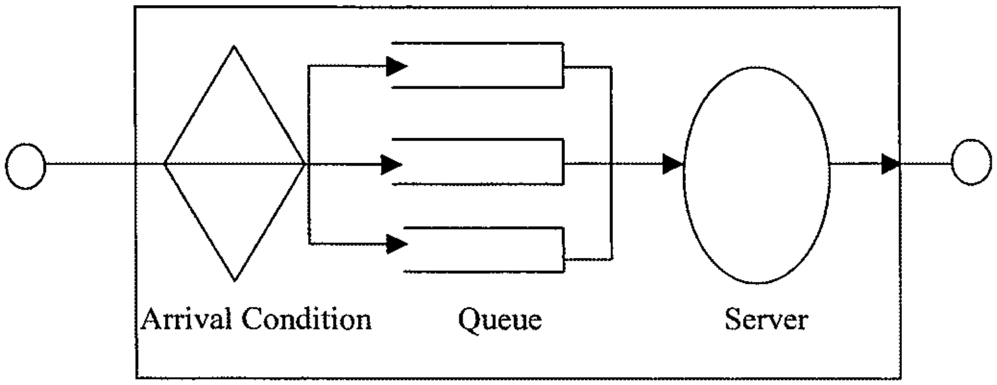


Figure 5. The Components of the Link

The Basic Components of the Link

A link models the transfer of an object from one process to the next. The link consists of Arrival Conditions, Queues, and Process Handlers as described for a node. The structure of the link is illustrated in Figure 5. The link is essentially a subset of the node. A link has exactly one entry point and one exit point, although it may have multiple internal queues. A link never has resources associated with it. Thus, a link may only have a delay operation associated with it. This wait duration could be dependent on the link properties or the entity properties. It is assumed that any extra logic that has to be associated with the delay function in the link is called separately. This reduces the complexity of the link node and keeps it as generic as possible.

Modeling Conditions (Constraints)

While using this methodology, a designer should consider some general guidelines. These guidelines are in no way meant to be an artificially imposed constraint, but are chosen for their ease of generating a nodal construct.

General Guidelines:

- Each pair of nodes has at most one link in each direction in common.
- Each node has exactly one entry point.
- Each link has exactly one entry and one exit point.
- A link has no resources associated with it besides time.
- No objects are generated in links.
- Nodes are distinguished by links.

The above allows the model to be abstracted to create a hierarchical view of the design. To this end, two or more nodes may be logically joined to create Super Nodes as shown in Figure 6.

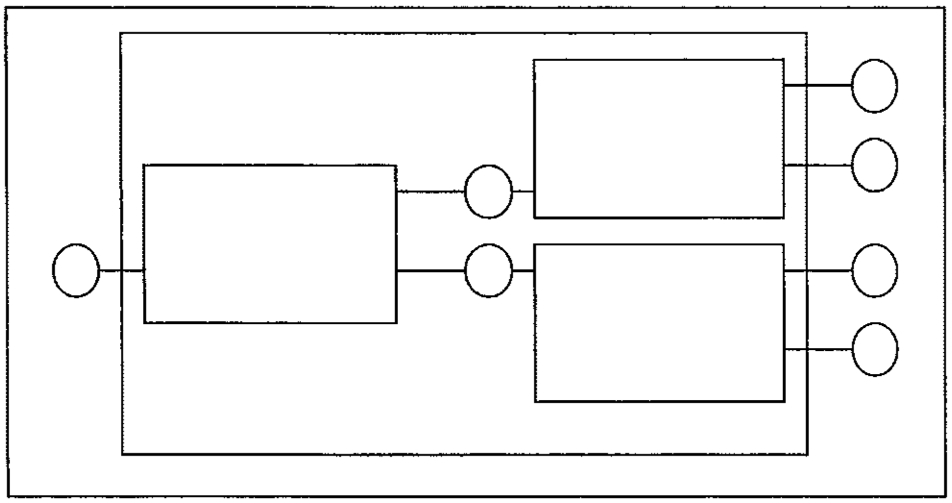


Figure 6. The Super Node

One of the uses of Super Nodes is to ease the incorporation of multiple resource type needs for a particular process. Thus, if an arriving object needs a series of resources before it can be processed, using the Super Node process allows this to be done without the use of algorithms that may be complex or lead to starvation. Note from Figure 6 that the first node distributes the objects into what could be parallel tasks.

Parameter Object

The GSSM aims at providing a clean interface between the various blocks of the simulation. The main cause for an ambiguous interface in any simulation model is due to the need for message passing throughout the model. The complexity of the simulation model increases if the task of passing information is built into the model. A cleaner manner of achieving a message passing mechanism is to tag the information with the entity itself. The entity is the only object that flows through the simulation model and could be effectively used as a vehicle to carry information from one point in the model to another. The point to be noted here, however, is that the information that is being passed along may not be information that pertains to the entity at all and should not be confused with the entity's attributes. Thus, information of this type has to be carried by the entity using a separate object that is simply a record of the data that has to be carried along with the entity. Hence, the parameter object is simply an aggregation of the entity object and the record object.

Network Model

The crucial feature in any simulation is the interfaces between objects in the simulation. The interface acts as a mechanism to transfer information through the logic of the simulation. The GSSM is designed to address this issue clearly. The rigid design constants of nodes and links along with the parameter object helps define a clear interface between any processing elements in the simulation. The added advantage of these constructs is the ease with which an unwanted object could be removed from the simulation and the interfaces between the remaining blocks rebuilt with ease. Thus, the network model would consist of a network of nodes and links with the entities flowing through it.

Implementation

The GSSM is implemented in MODSIM III (3) as a proof of concept. MODSIM provides an appropriate object-oriented language for simulation development. It provides appropriate management for queues and resources, but lacks management support for an integrated server model. The MODSIM implementation of GSSM is described below.

The node is the main building block and has to be built first in order to fully understand the functionality that is provided within the model. The node is developed using a basic template for a server model, discussed above, and includes all the basic building blocks necessary for the node. Maintaining a purely object-oriented structure, all the building blocks are developed as objects and then associated together in a node or link. First, we shall discuss the implementation of the internal blocks of the node and then the basic template that forms the node. The link is merely a subset of the node and hence can be derived from the same template. The following sections discuss the definition of these objects that build up the GSSM.

How to Build a Model Using GSSM

The code that follows in the appendix implements a basic server queuing model. The ModSim III implementation follows this general approach to building:

- Abstract and componentize your model into a network of queues.
- Build up the individual component objects from the baseline virtual objects.
- Merge all the components together.

The basic object templates are:

- entityObj
- recObj
- parameterObj
- genericQueueObj
- conditionObj
- resObj
- baseObj
- serverObj

Their interplay and dependencies combine to form the overall communication mechanism for the model. See **Appendix A** for another table view of the mod files with each object where each object shows its methods, attributes, and inherited object.

genericQueueObj

The **genericQueueObj** is the queue object in the server model, and it is derived from the **BasicGroupObj**. The rules associated with the queue are kept as generic as possible using this derivation and can be modified depending on the type of node or link the queue is being implemented in. The **genericQueueObj** is defined by:

```
genericQueueObj = OBJECT(BasicGroupObj);
        ASK METHOD insertObject(INOUT param :parameterObj);
        ASK METHOD removeObject(INOUT param :parameterObj);
END OBJECT {genericQueueObj};
```

The two methods, **insertObject** and **removeObject** have been created to allow the programmer to define the insertion and deletion rule associated with a particular GSSM. These rules could be derived or could be a separate logic. Any queue associated with a node or a link in the simulation are derived from the **genericQueueObj** and modified to the needs of the GSSM.

conditionObj

The **conditionObj** forms the base object for the condition blocks that exist in the GSSM. It is defined by:

```
conditionObj = OBJECT;
        ASK METHOD modifyParameters(INOUT param : parameterObj);
        ASK METHOD mapToQueue(INOUT param : parameterObj);
        ASK METHOD init(INOUT anyqueue : ANYARRAY);
        PRIVATE
                quarray : ARRAY INTEGER OF genericQueueObj;
END OBJECT {conditionObj};
```

The **mapToQueue** method performs the function of mapping the arriving object into the correct queue based on the logic that is appropriate for the node and the type of the entity object. In many cases, it might be necessary to modify the attributes of the arriving object based on the node. This can be accomplished using the **modifyParameters** method in the **conditionObj**. The **init** method is called to carry out any initialization of the condition object that has to be done during the setting up of the object. The arrival and departure conditions of a node are derived from the **conditionObj** because the functionality of both the objects is the same. The **conditionObj** also needs to have references to the queues. Hence, the **conditionObj** has within its attribute list an array of references to queue objects.

serverObj

The **serverObj** is the center of activity in the node or a link. The primary difference between the server in a node and a server in a link is that the server in the node has resources associated with it. The remaining functionality in the two types of servers is the same and hence could be derived from a common **serverObj** as defined by:

```
serverStateType = (IDLE, BUSY, DOWN);
serverObj = OBJECT;
        WAITFOR METHOD process;
        ASK METHOD init;
        PRIVATE
                inQ : ARRAY INTEGER OF genericQueueObj;
                outQ : ARRAY INTEGER OF genericQueueObj;
                state : serverStateType;
                localParam : parameterObj;
        ASK METHOD timeComputation(IN param : parameterObj):REAL;
        ASK METHOD getFromQ(IN qindex : INTEGER):parameterObj;
        ASK METHOD getQindex : INTEGER;
END OBJECT {serverObj};
```

As the link and node servers are derived from the common **serverObj**, there is no resource associated with this object. This would indicate that each node's server object, derived from **serverObj,** would have as an attribute an array of references to the resources. Unless the link object's server requires any additional functionality, it could be directly defined as a **serverObj**.

The **process** method in the server object is the method that carries out the actual time elapsing operation that simulates the processing time of the node or the link. The process method in a node first checks for availability of server and resource and then removes a parameter object from the queue. It then calls a **WAIT** function on the parameter object to elapse time. In the link, the resource is not a constraint that is considered, as there are no associated resources with it. The process removes the parameter object from the queue using the **getFromQ** method, which in turn calls the **getQindex** to obtain the appropriate queue that is to be used. The server to simulate the processing of the current **parameterObj** calls the **timecomputation** method to obtain the time duration value that must elapsed.

In addition to these methods, the **serverObj** has attributes associated with it. The server needs to have a state associated with it to enable it to be active or down. The **stateType** attribute is an enumerated data type and varies according to the possible states in which the server could be. The server also needs to have references to the input and output queues associated with the node so as to enable it to remove objects from the input queue and place them in the output queue after processing. The server also has a reference to the parameter object that it is currently handling.

The servers associated with nodes would have in addition to the above-mentioned nodes, an extra attribute, which is a set of references to the resources associated with it.

These resources are from a pool of resources and are not an integral part of the node. Hence, it becomes essential to explicitly define the references.

resourceObj

The resource object is envisioned to be a stand-alone object in a pool of resources. The servers of nodes require associated resources. Hence, every server that requires a particular resource would set up a reference to that **resourceObj** from within its resource attribute. The **resourceObj** is defined by:

```
stateType = (NONE,NS,WE,UP,DOWN);
           resourceObj = OBJECT;
           state : stateType;
           ASK METHOD query(OUT qState: stateType);
END OBJECT;
```

Since the resource is a stand-alone object, there are very few methods and attributes associated with it. The resource primarily has an attribute that stores the current state of the resource. The state attribute is an enumerated data type, and the values depend on the resource. The only method associated with the resource is a method that the server would call to query the current state of the resource. Any additional functionality provided to the resource has to be done in the derived resource object.

baseObj

The **baseObj** is the foundation on which the node or link is built. The base object is simply a template that brings together the various blocks that make up the GSSM. Thus, the **baseObj** mainly includes the references to the blocks of the node or link and methods that carry out some of the operations for **parmeterObj** handling between the blocks and between the GSSMs.

```
baseObj = OBJECT;
        ASK METHOD query(INOUT param : parameterObj): BOOLEAN;
        ASK METHOD accept(INOUT param : parameterObj): BOOLEAN;
        ASK METHOD setName(IN nam : STRING);
        ASK METHOD init(INOUT arrcond : conditionObj; INOUT depcond : conditionObj; INOUT
                inarray: ANYARRAY; INOUT outarray : ANYARRAY; INOUT serv: serverObj);
        ASK METHOD setChildren(INOUT child : ANYARRAY);
        ASK METHOD requestToSend(INOUT param : parameterObj) : BOOLEAN;
        ASK METHOD send(INOUT param : parameterObj);
        ASK METHOD callProcess;
        PRIVATE
                name : STRING;
                incond : conditionObj;
                outcond : conditionObj;
                inqueue : ARRAY INTEGER OF genericQueueObj;
                outqueue : ARRAY INTEGER OF genericQueueObj;
                server : serverObj;
                children : ARRAY INTEGER OF baseObj;
END OBJECT;
```

The **name** attribute is used to identify the node or link using a unique string. The **children** attribute is an array of references to the nodes and links that immediately follow the parent node. By using these references, the node or link can establish the various options that are available as exit destinations. It is very helpful in passing the ownership of the **parameterObj** from one GSSM to another. The remaining attributes of the **baseObj** are simply references to the blocks that make up the node or link.

The **query** method of a **baseObj** is called by its parent **baseObj** to verify if the **parameterObj** of which it is looking to pass ownership to is acceptable or not. The point to note here is that the **query** method does not pass over the ownership of the parameterObj from one GSSM to another. The **accept** method, on the other hand, is called following a positive response from the **query** method and actually passes ownership from one GSSM to another.

The **requestToSend** method is called by the GSSM to access the query method of its child GSSM. Once a positive response is received on the **requestToSend** method, the send method is called to activate the **accept** method of its child GSSM. The **requestToSend** and send method are simply wrappers around the **query** and **accept** method and are used to call the **query** and **accept** method of all the children GSSM. The **setName** method and the **init** methods are used to initialize the GSSM once it has been created. The **callprocess** method is activated by the GSSM to begin the processing action of the server. The **process** method is self-maintaining and has to be simply called once before it can maintain itself for the remaining simulation.

parameterObj

The parameter object has been defined for moving an entity across the simulation model and to bring along other necessary information blocks along with it. The object is defined by:

```
parameterObj = OBJECT;
          entity : entityObj;
          rec : recObj;
END OBJECT;
```

From the above definition of the parameter object, it can be observed that the parameter object is an aggregation of the entity and record object. The entity object is a physical entity that is flowing through the simulation model. The record object, on the other hand, is basically an object that gets assigned information that it is responsible for carrying from one node to the other. Thus, if a node has to pass information to another node, it uses the **parameterObj**. The node first checks if the entity would go through the destination node, and if so, the information is assigned to its **recourceObj**. The **parameterObj** then simply carries on with its flow through the model. When the **parameterObj** arrives at the destination node, the node reads out the information from the **recourceObj.** In such a fashion, the communication interface between the nodes and links is kept as simple as possible.

Process Flow

Once an entity is created, it will enter into the network in the form of a parameter. The node in which it is entered is **ASKed** to **accept** it. If accepted, the node will then ask its **incond** object to **mapToQueue** the parameter. The node will then call its own method **startmeup**. **startmeup** calls the node's server with a **WAIT** for **process**. The server will then check its state and its required resource(s) state, and if it is available or idle, it will then process the entity. The server will determine a queue with **getFromQ** (implementation dependent) and take the parameter object from it, change its state if appropriate, calculate wait time, wait that time, modify the parameters entity or record, and then send the entity to the **outcond** queues. The server will then call the node's **requestToSend** in order to let the node know that its **outqueue** has an item in it. The server will now put itself in a self-maintaining loop. If there are entities in its queues that it could not have processed because it was busy, it will now call **getFromQ** to start the process again. If the queues are empty or it has no resources available to it, it sets its state to **IDLE** and stops processing.

Once the server calls its node's **send**, the **send** method will determine which of the possible links or nodes from the current node's **children** list to send it to by asking it to **accept** the parameter object. Any kind of scenarios is possible for acceptance and denial of the **requestToSend** and it is completely implementation dependent. Typically in basic models, it is assumed automatically accepted, but if the model needs a more robust mechanism, GSSM makes accommodations by providing for the possibility of a denial but leaves the implementation to the modeler.

Initialization

As with object-oriented coding in particular, initialization is extremely methodical and intensively tedious, but if the object model is properly designed, once initialized there is little other code to the main program, since all the objects in the model are self-maintaining. This makes for very well structured and maintainable code. In addition, note that being "extremely methodical and intensively tedious," it is straightforward and well suited to be automated (hint/hint). Thus, a well-designed graphical users interface or GUI to help eliminate syntactical and procedural mistakes or "bugs" can eliminate a lot of the overhead of this type of object-orientated code.

Below is an example of the way every node or link in the GSSM model is initialized if using an object-oriented language like MODSIM. First, create a variable of each object in the node.

For example the Gate Node:

```
gate : gateNodeObj;

gatein : gateIncondObj;

gateout : gateOutcondObj;

gateInQ :  genericQueueObj ;

gateOutQ : genericQueueObj ;

gateserver : gateServerObj;

gateRes : gateResObj;

gateQinArray: ARRAY INTEGER OF genericQueueObj;

gateQoutArray: ARRAY INTEGER OF genericQueueObj;

gateChildren : ARRAY INTEGER OF baseObj;
```

Then instigate each object and initialize where appropriate with the name and sub-objects and associated the objects in the node.

Note that a node has the sub-objects hierarchy:

```
{node object}
          {in condition object}
                    {in queue array object}
                              {in queue object}
          {out condition obj}
                    {out queue array object}
                              {out queue object}
          {server}
                    {Resource}
          {children array}
                    {node or link object}
```

For example the Gate Node:

```
{initialize}
{gate node object}
NEW(gate);
ASK gate TO setName("Gate");
{in condition obj}
                    NEW(gatein);
          {in queue object}
                    NEW(gateQinArray,1..1);
                              NEW(gateInQ);
      gateQinArray[1] := gateInQ;
```

```
  ASK gatein TO init(gateQinArray);


{out condition obj}
NEW(gateout);
          {out queue object}
          NEW(gateOutQ);
                    NEW(gateQoutArray, 1..1);
          gateQoutArray[1]:= gateOutQ;
          ASK gateout TO init(gateQoutArray);
{server}
NEW(gateserver);
          {Resource}
          NEW(gateRes);
          ASK gateRes TO init(gate);
ASK gateserver TO init( gate);
ASK gateserver TO initRes( gateRes);


ASK gate TO init(gatein, gateout,gateQinArray,gateQoutArray,gateserver);
{end of gate init}
```

The last step once all the nodes and links have been internally initialized is to build the network. This step is for the most part quite simple, and it consists essentially of instigating the **children** object array and assigning the next nodes or links in the network to it by reference.

```
{setup model structure}
NEW(gateChildren,1..1);
NEW(linkChildren,1..1);
        interChildren := NILARRAY;
        gateChildren[1] := baseObj(link);
        linkChildren[1] := baseObj(intersection);
        ASK gate TO setChildren(gateChildren);
        ASK link TO setChildren(linkChildren);
        ASK intersection TO setChildren(interChildren);
```

Note that we never instigate the **interChildren** array, and we safely assigned to **NULL** because it is the last node in the network with no children. This completes the network, and now all it needs is the entity/parameter generator to begin.

```
NEW(custGenerator);
{main program}
        TELL custGenerator TO initSimulation;
        TELL custGenerator TO GenCustomers;
StartSimulation;
{end main program}
OUTPUT("THE END");
```

Once the simulation is completed, dispose the objects to free up all allocated memory. Below is an example for the Gate Node:

```
        ASK gateInQ TO Empty;
        ASK gateOutQ TO Empty;
        {cleanup}
        DISPOSE(gatein);
        DISPOSE(gateQinArray);
```

```
DISPOSE(gateInQ);

DISPOSE(gateout);

DISPOSE(gateOutQ);

DISPOSE(gateQoutArray);

DISPOSE(gateserver);

DISPOSE(gateRes);

DISPOSE(gate);

DISPOSE(gateChildren);
```

Note that it is good practice to dispose of all sub-objects before disposing of the base object (mainly for record keeping ease). The network is disposed last (although it could have been destroyed at any time, preferably first).

IV. GSSM EXAMPLE: PORT OBJECTS IN PORTSIM

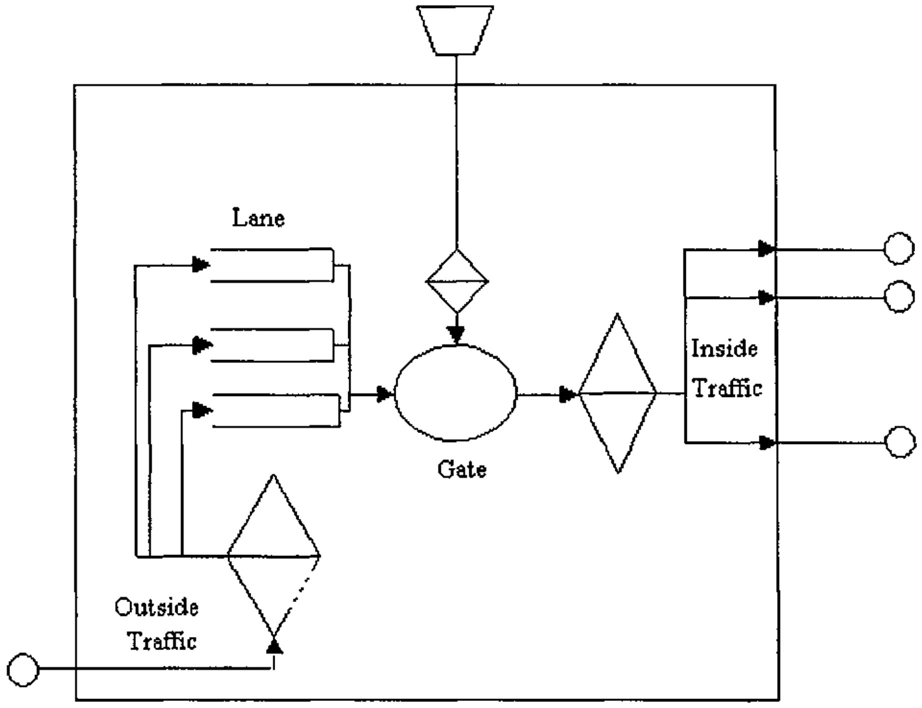


Figure 7. The Gate Node

The aim of this simulation project is to display the ease with which the nodes and links could be applied to a simulation scenario. PORTSIM simulates the military traffic through a port. The required functionality in PORTSIM is the passing of the commercial traffic along with the military traffic.

The Gate in PORTSIM

The current implementation of port gates in PORTSIM would require a separate logic for the gate processing for commercial vehicles. By employing GSSM to model the gates, the commercial traffic could be handled by the very same gate object that handles the military traffic by simply identifying the entities in the arrival condition block of the gate object. However, this is beyond the scope of this example.

In this implementation, we also assume that the gate has only one lane and hence only one vehicle can pass through the gate at a time. However, multiple lanes could be added to the model by simply adding multiple input queues and making necessary changes in the conditions in the server and the arrival condition. Thus, in the implementation of a gate, in this project the gate object is simply a base object as all the functionality necessary for the gate object exists within the base object. The model is presented in Figure 7. Figure 8 shows the gate node, the link, and the staging area network.

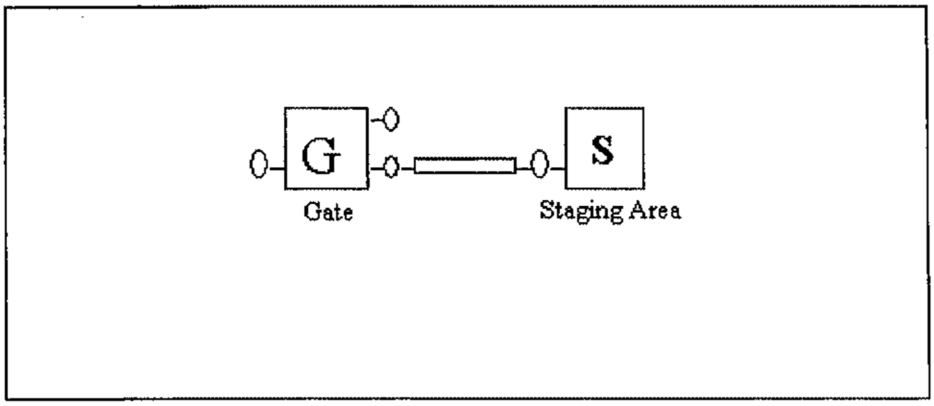


Figure 8. The Network Model

An Intersection

One of the strengths of GSSM is its ability to easily insert new nodes into an existing model. In this project, an existing port model is considered which was developed such that vehicles are transported from one node (the Gate) to another node (a staging area) as shown in Figure 8. This is done to display the ease with which new nodes can be plugged into the model without having to undergo an extensive redesigning process.

The model now has an intersection added to it between the gate and the staging area as shown in Figure 9. The designer simply creates a new node to model the rail crossing and the link that connected the gate and the staging nodes is removed. Lastly, two new links that link the gate to the crossing and the crossing to the staging are injected. The new model is demonstrated in Figure 10. What makes GSSM particularly good at accomplishing this task is that once a generic rail crossing is created, it can be reused with perhaps only slight modification.

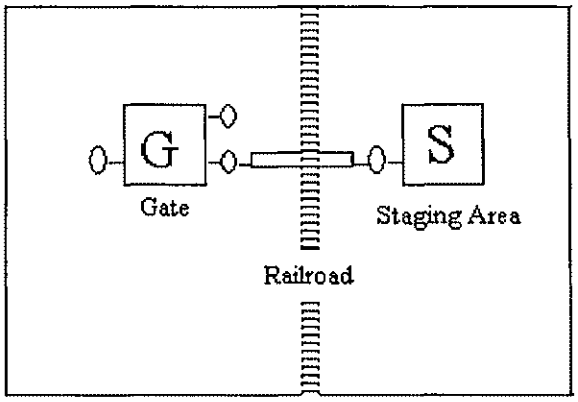


Figure 9. Conflict Point Insertion

Thus, the main advantage that can be observed in the use of the GSSM is the reusability of the templates that are created in the generic model. The intersection is developed from the basic objects that are defined as templates with minimal modifications. Then links are defined based on the needs of transfer of entities from one GSSM to another. It is also now possible to reuse the node that was created for the rail intersection anywhere in the code with minimal modifications to represent other rail intersections. All these possibilities together prove a very strong case for the application of GSSM in simulation modeling.

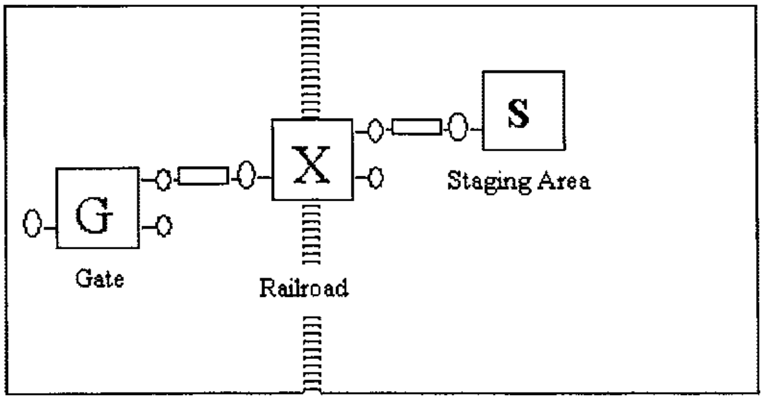


Figure 10. Conflict Point Resolution

V. CONCLUSION

With GSSM, the idea is to develop a reusable product, that is, a product that will last beyond its use in PORTSIM. The aim is to develop a vast repository of Nodal and link objects from which we will be able to draw upon for use in other models. Such repositories will allow for perhaps on the fly (pre-compile time) design modification of PORTSIM or other models based upon GSSM. Thus using GSSM, one has a mechanism that allows the quick insertion, deletion, and modifications of the network of server queues in our simulation products. GSSM is essentially a high-level abstraction to the low-level server-queuing models, and it attempts to bring into discussion the interiors of simulation engines that usually hide behind GUIs.

Some further work with GSSM will be necessary to truly make it useful to the general simulation public. This includes a refinement of the resource pool. The current state of the resource management is lacking in generic features. These guidelines should be made to handle the communications of the resources and the nodes they serve. Another portion that is currently missing is the GUI. GSSM could be the backend structure behind some easy to use tool. With those two items, GSSM should give a modeler the power to build programming constructs quickly from abstract models.

BIBLIOGRAPHY

(1) ARENA Version 3.01, Online Documentation. CD-ROM. System Modeling. 1997.

(2) Balci, O. and R.E. Nance. 1992. "*The Simulation Model Development Environment: An Overview.*" **In Proceedings of the 1992 Winter Simulation Conference** (Arlington, VA, Dec. 13-16). IEEE, Piscataway, NJ, 726-736.

(3) CACI Products Co., **ModSim Reference Manual**. La Jolla, California: CACI Products Company, 1997.

(4) Fishwick, Paul A., **Simulation Model Design and Execution, Building Digital Worlds**. Englewood Cliffs: Prentice Hall, 1995.

(5) Kelton, W. David, Randall P. Sadowski, and Deborah A. Sadowski. **Simulation with ARENA**. Boston: WCB McGraw-Hill, 1998.

(6) Kleinrock, L., **Queuing Systems, Vol. I, Theory**. New York: John Wiley & Sons, 1975.

(7) Kleinrock, L., **Queuing Systems, Vol. II, Computer Applications**. New York: John Wiley & Sons, 1975.

(8) Law, Averill M., and W. David Kelton. **Simulation Modeling and Analysis**. New York: McGraw-Hill, 1991.

(9) Leathrum, J, R. Mielke, M. Meyer, J. Joines, C. Macal, and M. Nevins. "*Strategies for Integrating Commercial and Military Port Simulation.*" **Proceedings of the 1997 National Conference of the American Society for Engineering Management**. Oct. 1997. Virginia Beach, VA. pp 37-42.

(10) Little, M. C., D. L. McCue, "*Construction and Use of a Simulation Package in C++,*" **Computing Science Technical Report**, University of Newcastle upon

Tyne, Number 437, July 1993 (also appeared in the **C User's Journal** Vol. 12 Number 3, March 1994). http://cxxsim.ncl.ac.uk/papers.html

(11) Trivedi, K., **Probability & Statistics with Reliability, Queuing, and Computer Science Applications**. Englewood Cliffs, NJ: Prentice-Hall, 1982.

APPENDIX

A. OBJECT STRUCTURES

Parameter.mod			
entityObj			
Methods		**Attributes**	
	setArraySize		type
	setAttribute		arraysize
	getAttributes		attribute
	setType		
	getType		
recObj			
Methods		**Attributes**	
	setAttribute		arraysize
	getAttributes		attribute
	getArraySize		
parameterObj			
Methods		**Attributes**	
	setName		entity
	getName		rec
	getArraySize		name

genericqueue.mod			
genericQueueObj →BasicGroupObj*			
Methods		**Attributes**	
	insertObject		N/A
	removeObject		

*ModSim III object.

condition.mod			
conditionObj			
Methods		Attributes	
	modifyParameters		quarry
	mapToQueue		
	init		

resource.mod			
stateType {Enumerated Type}			
resObj			
Methods		Attributes	
	query		state

server.mod			
serverStateType {Enumerated Type}			
serverObj			
Methods		Attributes	
	process		servRes
	init		node
	timeComputation		state
	getFromQ		localParam
	getQindex		count

base.mod			
baseObj			
Methods		Attributes	
	query		name
	accept		incond
	setName		outcond
	startmeup		inqueue
	init		outqueue
	getChildren		serv
	requestToSend		children
	send		
	callProcess		
nodeObj → baseObj			
Methods		Attributes	
	N/A		res
linkObj →baseObj			
Methods		Attributes	
	N/A		N/A

B. GSSM CODE

{dcondition.mod}

```
DEFINITION MODULE condition;

FROM genericqueue IMPORT  genericQueueObj;

FROM parameter IMPORT parameterObj;

VAR

TYPE

conditionObj = OBJECT;

        ASK METHOD modifyParameters(INOUT param : parameterObj);

        {Modifying the parameters of parameterObj based on the type of node or link}

        ASK METHOD mapToQueue(INOUT param : parameterObj);

        ASK METHOD init(INOUT anyqueue : ANYARRAY);

        PRIVATE

        quarray : ARRAY INTEGER OF genericQueueObj;

END OBJECT {conditionObj};


gateIncondObj = OBJECT(conditionObj);

END OBJECT;

gateOutcondObj = OBJECT(conditionObj)


END OBJECT;

intersectionIncondObj = OBJECT(conditionObj)

END OBJECT;

intersectionOutcondObj = OBJECT(conditionObj)

END OBJECT;

END MODULE.
```

{dbase.mod}

```
DEFINITION MODULE base;


FROM parameter IMPORT parameterObj;

FROM condition IMPORT conditionObj;

FROM server IMPORT serverObj;

FROM resource IMPORT gateResObj;

FROM resource IMPORT intersectionResObj;

FROM resource IMPORT resObj;

FROM genericqueue IMPORT genericQueueObj;

VAR

TYPE

baseObj = OBJECT;

{returns true if node can accept parameterObj at that time.

        DOES NOT TRANSFER OWNERSHIP}

ASK METHOD query(INOUT param : parameterObj): BOOLEAN;

{returns true if node can accept parameterObj at that time. TRANSFERS OWNERSHIP}

ASK METHOD accept(INOUT par : parameterObj):BOOLEAN;

TELL METHOD startmeup;

ASK METHOD setName(IN nam : STRING);

ASK METHOD init(INOUT arrcond : conditionObj; INOUT depcond : conditionObj; INOUT inarray :

        ANYARRAY; INOUT outarray : ANYARRAY; INOUT servObj: serverObj);

ASK METHOD setChildren(INOUT child : ANYARRAY);

ASK METHOD requestToSend(INOUT param : parameterObj) : BOOLEAN;

ASK METHOD send(INOUT param : parameterObj):BOOLEAN;

TELL METHOD callProcess;

{PRIVATE}

name : STRING;
```

```
incond : conditionObj;

outcond : conditionObj;

inqueue : ARRAY INTEGER OF genericQueueObj;

outqueue : ARRAY INTEGER OF genericQueueObj;

serv : serverObj;

children : ARRAY INTEGER OF baseObj;

{param: parameterObj;}

END OBJECT;


nodeObj = OBJECT(baseObj); {TEMPLATE}


PRIVATE

res : ARRAY INTEGER OF resObj;


END OBJECT;

linkObj = OBJECT(baseObj);

END OBJECT;


gateNodeObj = OBJECT(baseObj);

gateRes : gateResObj;

END OBJECT;


intersectionNodeObj = OBJECT(baseObj);

intersectionRes : intersectionResObj;

END OBJECT; {intersectionNodeObj}


END MODULE.
```

{Dgenericqueue.mod}

```
DEFINITION MODULE genericqueue;


FROM GrpMod IMPORT BasicGroupObj;

FROM parameter IMPORT parameterObj;

VAR

TYPE

genericQueueObj = OBJECT(BasicGroupObj);


ASK METHOD insertObject(INOUT param :parameterObj);

ASK METHOD removeObject(INOUT param :parameterObj);


END OBJECT {genericQueueObj};

END MODULE.
```

{Dparameter.mod}

```
DEFINITION MODULE parameter;

VAR

TYPE

entityObj = OBJECT;


ASK METHOD setArraySize(IN size : INTEGER);

ASK METHOD setAttribute(IN index : INTEGER; IN item :STRING): BOOLEAN;

ASK METHOD getAttributes(IN index : INTEGER; OUT item : STRING): BOOLEAN;


{

ASK METHOD setName(IN Name : STRING);

ASK METHOD getName : STRING;

}

ASK METHOD setType(IN Type : STRING);

ASK METHOD getType(OUT Type : STRING);


PRIVATE

{name : STRING;}

type : STRING;

arraysize : INTEGER;

attribute : ARRAY INTEGER OF STRING;

END OBJECT;


recObj = OBJECT;

ASK METHOD setAttribute(IN index : INTEGER; IN item : STRING):BOOLEAN;

ASK METHOD setArraySize(IN size : INTEGER);

ASK METHOD getAttributes(IN index : INTEGER; OUT item : STRING):BOOLEAN;
```

```
PRIVATE

attribute   : ARRAY INTEGER OF STRING;

arraysize : INTEGER;

END OBJECT;


parameterObj = OBJECT;

ASK METHOD setName(IN Name : STRING);

ASK METHOD getName : STRING;

entity : entityObj;

rec : recObj;

name : STRING;

END OBJECT;

END MODULE.
```

{Dresource.mod}

```
DEFINITION MODULE resource;


FROM base IMPORT gateNodeObj;

FROM base IMPORT intersectionNodeObj;


VAR

TYPE

stateType = (NONE,NS,WE,UP,DOWN);

resObj = OBJECT;

        state : stateType;

        ASK METHOD query(OUT qState: stateType);

END OBJECT;


gateResObj = OBJECT(resObj);

        ASK METHOD init( INOUT acqnode : gateNodeObj);

        TELL METHOD startBuss;

        {wait for 12 hours and if not DOWN call

        processor function of server}


PRIVATE

        {state : stateType;}

        nodeinst : gateNodeObj;

OVERRIDE

        ASK METHOD query(OUT qState : stateType);

END OBJECT {gateresObj};
```

```
intersectionResObj = OBJECT(resObj);

        ASK METHOD init( INOUT acqnode : intersectionNodeObj);

        TELL METHOD startBuss;  { 5 minute light switching}

PRIVATE

        {state : stateType;}

        nodeinst : intersectionNodeObj;

OVERRIDE

  ASK METHOD query(OUT qState : stateType);

END OBJECT;

END MODULE.
```

{Dserver.mod}

```
DEFINITION MODULE server;

FROM resource IMPORT resObj, stateType;

FROM genericqueue IMPORT genericQueueObj;

FROM parameter IMPORT parameterObj;

FROM base IMPORT baseObj;


VAR

TYPE

serverStateType = (IDLE, BUSY, DOWN);

serverObj = OBJECT;

WAITFOR METHOD process;

{Carrying out the preprocessing and resource capture before service}

ASK METHOD init(INOUT Node:baseObj);


PRIVATE

servRes: resObj;

node: baseObj;

state : serverStateType;

localParam : parameterObj;

        ASK METHOD timeComputation(IN param : parameterObj):REAL;

        ASK METHOD getFromQ(IN qindex : INTEGER):parameterObj;

        ASK METHOD getQindex : INTEGER;

        count: INTEGER;

END OBJECT {serverObj};


gateServerObj = OBJECT(serverObj);

  ASK METHOD initRes(INOUT resServ : resObj);
```

```
OVERRIDE
  ASK METHOD timeComputation(IN param : parameterObj):REAL;
  ASK METHOD getFromQ(IN qindex : INTEGER):parameterObj;
  ASK METHOD getQindex : INTEGER;
  WAITFOR METHOD process;
        {Carrying out the preprocessing and resource
        capture before service}
END OBJECT;


interServerObj = OBJECT(serverObj);
  ASK METHOD initRes(INOUT resServ : resObj);
OVERRIDE
  ASK METHOD timeComputation(IN param : parameterObj):REAL;
  ASK METHOD getFromQ(IN qindex : INTEGER):parameterObj;
  ASK METHOD getQindex : INTEGER;
  WAITFOR METHOD process;
        {Carrying out the preprocessing and resource
         capture before service}
        END OBJECT;
END MODULE.
```

{Ibase.mod}

```
IMPLEMENTATION MODULE base;

OBJECT baseObj;


ASK METHOD query(INOUT param : parameterObj) : BOOLEAN;

VAR

BEGIN

RETURN TRUE;

END METHOD; {query}


ASK METHOD accept(INOUT par : parameterObj):BOOLEAN;

VAR

BEGIN

  ASK incond TO mapToQueue(par);

  TELL SELF TO startmeup;

RETURN TRUE;

END METHOD;


TELL METHOD startmeup;

VAR

 temp:INTEGER;

 tbool :BOOLEAN;

BEGIN

OUTPUT(name, " Map to Queue");

{   ASK incond TO mapToQueue(param);}

  WAIT FOR serv TO process;

  END WAIT;

{   OUTPUT(name, " Sent to OutQ");}
{   ASK outcond TO mapToQueue(param);
   tbool := ASK SELF TO requestToSend(param);
   tbool := ASK SELF TO send(param);
}
END METHOD;

ASK METHOD setName(IN nam :STRING);
VAR


BEGIN
   name := nam;
END METHOD;

ASK METHOD init(INOUT arrcond : conditionObj; INOUT depcond : conditionObj; INOUT inarray :
          ANYARRAY; INOUT outarray : ANYARRAY; INOUT servObj: serverObj);
BEGIN
incond := arrcond;
outcond := depcond;
inqueue := inarray;
outqueue := outarray;
serv := servObj;
incond.init(inarray);
outcond.init(outarray);
END METHOD;{init}

ASK METHOD setChildren(INOUT child : ANYARRAY);
VAR
```

```
BEGIN
   children := child;
END METHOD;


ASK METHOD requestToSend(INOUT param :parameterObj): BOOLEAN;
VAR
temp : INTEGER;
tbool : BOOLEAN;
BEGIN


IF(children = NILARRAY)
   OUTPUT("ERROR: CHILDREN IS NilArray");
   RETURN FALSE;
END IF;
temp := LOW(children);
tbool := ASK children[temp] TO query(param);
RETURN tbool;
END METHOD;


ASK METHOD send(INOUT param: parameterObj):BOOLEAN;
VAR
temp : INTEGER;
tbool : BOOLEAN;
BEGIN
IF(children = NILARRAY)
   OUTPUT("ERROR: CHILDREN IS NilArray");
   RETURN FALSE;
END IF;
```

```
temp := LOW(children);
OUTPUT(name, " sent to ",children[temp].name);


IF param = NILOBJ
   OUTPUT("parameter = NILOBJ");
   tbool := FALSE;
ELSE
   ASK outqueue[1] TO RemoveThis(param);
   tbool := ASK children[temp] TO accept(param);
END IF;
RETURN tbool;
END METHOD;
TELL METHOD callProcess;
VAR
BEGIN
OUTPUT(name, " processing ");
   WAIT FOR serv TO process;
   END WAIT;
END METHOD
END OBJECT; {baseObj}


END MODULE.
```

{Icondition.mod}

```
IMPLEMENTATION MODULE condition;

OBJECT conditionObj;

ASK METHOD modifyParameters (INOUT param : parameterObj);

VAR

BEGIN

{modify the parameters of parameterObj if the node or link has to make any modification}

END METHOD;

ASK METHOD mapToQueue(INOUT param : parameterObj);

VAR

temp :INTEGER;

BEGIN

temp := LOW(quarray);

IF(quarray = NILARRAY)

   OUTPUT("ERROR : quarray is NilArray");

ELSE

   ASK quarray[temp] TO insertObject(param);

END IF;

END METHOD;


ASK METHOD init(INOUT anyqueue : ANYARRAY);

BEGIN

quarray := anyqueue;

END METHOD {initcond};

END OBJECT; {conditionObj}

OBJECT gateIncondObj;


END OBJECT;
```

```
OBJECT gateOutcondObj;

END OBJECT;

OBJECT intersectionIncondObj;

END OBJECT;

OBJECT intersectionOutcondObj;

END OBJECT;

END MODULE. {Condition Module}
```

{IgenericQueue.mod}

```
IMPLEMENTATION MODULE genericqueue;


OBJECT genericQueueObj;

ASK METHOD insertObject(INOUT param : parameterObj);

VAR

BEGIN

{Add object to the Queue}

   ASK SELF Add(param);

END METHOD;

ASK METHOD removeObject(INOUT param :parameterObj);

VAR

BEGIN

{Remove the object from the Queue}

   param := ASK SELF Remove;

END METHOD;

END OBJECT;{myQueueObj}

END MODULE. {myqueue}
```

{Iparameter.mod}

```
IMPLEMENTATION MODULE parameter;

OBJECT entityObj;

ASK METHOD setArraySize(IN size : INTEGER);

VAR

BEGIN

  arraysize := size;

  NEW(attribute, 0..size-1);

END METHOD;


ASK METHOD setAttribute(IN index : INTEGER; IN item : STRING): BOOLEAN;

VAR

BEGIN

  IF arraysize > 0

    attribute[index] := item;

    RETURN TRUE;

  ELSE

    RETURN FALSE;

  END IF;

END METHOD;

ASK METHOD getAttributes(IN index : INTEGER; OUT item :STRING): BOOLEAN;

VAR

BEGIN

  IF arraysize > 0

    item:= attribute[index];

    RETURN TRUE;

  ELSE

    RETURN FALSE;
  END IF;
END METHOD;
{
ASK METHOD setName(IN Name : STRING);
VAR
BEGIN
  name := Name;
END METHOD;
ASK METHOD getName:STRING;
VAR
BEGIN
  RETURN name;
END METHOD;
}
ASK METHOD setType(IN Type : STRING);
VAR
BEGIN
  type := Type;
END METHOD;

ASK METHOD getType(OUT Type : STRING);
VAR
BEGIN
  Type := type;
END METHOD;
END OBJECT;{entityObj}
OBJECT recObj;
ASK METHOD setArraySize(IN size : INTEGER);
```

```
VAR
BEGIN
          arraysize := size;
          NEW(attribute, 0..size-1);
END METHOD;
ASK METHOD setAttribute(IN index : INTEGER; IN item : STRING): BOOLEAN;
VAR
BEGIN
   IF arraysize > 0
      attribute[index] := item;
      RETURN TRUE;
   ELSE
      RETURN FALSE;
   END IF;
END METHOD;

ASK METHOD getAttributes(IN index : INTEGER; OUT item :STRING): BOOLEAN;
VAR
BEGIN
   IF arraysize > 0
      item:= attribute[index];
      RETURN TRUE;
   ELSE
      RETURN FALSE;
   END IF;
END METHOD;
END OBJECT;{recObj}
```

```
OBJECT parameterObj;

ASK METHOD setName(IN Name : STRING);

VAR

BEGIN

   name := Name;

END METHOD;

ASK METHOD getName:STRING;

VAR

BEGIN

   RETURN name;

END METHOD;

END OBJECT;{parameterObj}

END MODULE.
```

{Iresource.mod}

```
IMPLEMENTATION MODULE resource;


OBJECT resObj;

ASK METHOD query(OUT qState: stateType);

VAR

BEGIN

END METHOD;

END OBJECT;


OBJECT gateResObj;

ASK METHOD query(OUT qState : stateType);

VAR

BEGIN

qState := state;

END METHOD;


ASK METHOD init( INOUT acqnode : gateNodeObj);

 VAR

 BEGIN

    state := UP;

   nodeinst := acqnode;

 END METHOD;


TELL METHOD startBuss;

VAR

   waitTime : REAL;

BEGIN
```

```
  waitTime := 43200.0;{12 hours of gate open time}
  WHILE (TRUE)
    WAIT DURATION waitTime
      IF state = UP
        state := DOWN;
      ELSE
        state := UP;
      END IF;
    END WAIT;
    TELL nodeinst TO callProcess;
  END WHILE;
END METHOD;
END OBJECT; {resourceObj}
OBJECT intersectionResObj;
ASK METHOD query(OUT qState : stateType);
VAR
BEGIN
qState := state;
END METHOD;
ASK METHOD init(INOUT acqnode : intersectionNodeObj);
VAR
BEGIN
       state := WE;
              nodeinst := acqnode;
END METHOD;
TELL METHOD startBuss;
VAR
  waitTime : REAL;
```

```
BEGIN
  waitTime := 3000.0;{5 minutes time between light changes}
  WHILE (TRUE)
    WAIT DURATION waitTime
      IF state = WE
        state := NS;
      ELSE
        state := WE;
      END IF;
    END WAIT;
    TELL nodeinst TO callProcess;
  END WHILE;
END METHOD;
END OBJECT; {intersectionResObj}
END MODULE. {resource}
```

{Iserver.mod}

```
IMPLEMENTATION MODULE server;


FROM SimMod IMPORT SimTime;


OBJECT serverObj;

ASK METHOD init(INOUT Node:baseObj );

VAR

BEGIN

count := 0;

state := IDLE;

node := Node;

END METHOD;


WAITFOR METHOD process;

VAR

tbool: BOOLEAN;

temp1 : INTEGER;

temp2 : REAL;

BEGIN

IF (state = IDLE)

   temp1 := ASK SELF TO getQindex;

   localParam := getFromQ(temp1);

   WHILE(localParam <> NILOBJ)

      state := BUSY;

      temp2 := timeComputation(localParam);

      WAIT DURATION temp2;

      END WAIT;
```

```
        OUTPUT("FINISH : ",node.name, " ", count," ",SimTime());
        ASK node.outcond TO mapToQueue(localParam);
        tbool := ASK node TO requestToSend(localParam);
        tbool := ASK node TO send(localParam);
        temp1 := ASK SELF TO getQindex;
        localParam := getFromQ(temp1);
    END WHILE;
    state := IDLE;
END IF;
END METHOD;


ASK METHOD getFromQ(IN qindex : INTEGER):parameterObj;
VAR
par: parameterObj;
BEGIN
IF (node.inqueue = NILARRAY)
    OUTPUT("inQ == NilArray");
    RETURN NILOBJ;
END IF;
IF node.inqueue[qindex].First = NILOBJ
   OUTPUT("node.inQ[qindex]=NILOBJ");
    RETURN NILOBJ;
ELSE
    par :=ASK node.inqueue[qindex] TO First;
   OUTPUT("REMOVED ",par.getName," from ",node.name);
            RETURN node.inqueue[qindex].Remove;
END IF;
END METHOD;
```

```
ASK METHOD getQindex : INTEGER;

VAR

BEGIN

IF (node.inqueue = NILARRAY)

   OUTPUT("inQ == NilArray");

   RETURN -1;

END IF;

RETURN LOW(node.inqueue);

END METHOD;


ASK METHOD timeComputation(IN param : parameterObj):REAL;

VAR

BEGIN

   RETURN 2.0;

END METHOD;

END OBJECT;


OBJECT gateServerObj;

ASK METHOD initRes(INOUT resServ : resObj);

VAR

BEGIN

   servRes := resServ;

   state := IDLE;

END METHOD;

WAITFOR METHOD process;

VAR

tbool: BOOLEAN;
```

```
temp : stateType;
temp1 : INTEGER;
BEGIN
IF (state = IDLE)
   temp1 := ASK SELF TO getQindex;
   localParam := getFromQ(temp1);
   WHILE(localParam <> NILOBJ)
      ASK servRes TO query(temp);
      IF(temp <> DOWN)
         state := BUSY;
         WAIT DURATION timeComputation(localParam);
         END WAIT;
         count := count + 1;
         OUTPUT("FINISH : ",node.name, " ", count," ",SimTime());
         ASK node.outcond TO mapToQueue(localParam);
         tbool := ASK node TO requestToSend(localParam);
         tbool := ASK node TO send(localParam);
         temp1 := ASK SELF TO getQindex;
         localParam := getFromQ(temp1);
      ELSE
         localParam := NILOBJ;
      END IF;
   END WHILE;
   state := IDLE;
END IF;
END METHOD;
ASK METHOD getFromQ(IN qindex : INTEGER):parameterObj;
VAR
```

```
par:parameterObj;
BEGIN
IF (node.inqueue = NILARRAY)
   OUTPUT("inQ == NilArray");
   RETURN NILOBJ;
END IF;
IF node.inqueue[qindex].First = NILOBJ
  OUTPUT("node.inQ[qindex]=NILOBJ");
   RETURN NILOBJ;
ELSE
   par :=ASK node.inqueue[qindex] TO First;
  OUTPUT("REMOVED ",par.getName," from ",node.name);
   RETURN node.inqueue[qindex].Remove;
END IF;
END METHOD;


ASK METHOD  getQindex : INTEGER;
VAR
BEGIN
IF (node.inqueue = NILARRAY)
   OUTPUT("inQ == NilArray");
   RETURN -1;
END IF;
RETURN LOW(node.inqueue);
END METHOD;
ASK METHOD timeComputation(IN param : parameterObj):REAL;
VAR
BEGIN
```

```
  RETURN 2.0;
END METHOD;
END OBJECT;
OBJECT interServerObj;
ASK METHOD initRes(INOUT resServ : resObj );
VAR
BEGIN
  servRes := resServ;
  state := IDLE;
END METHOD;


WAITFOR METHOD process;
VAR
tbool:BOOLEAN;
str: STRING;
temp : stateType;
temp1 : INTEGER;
BEGIN
IF (state = IDLE)
  temp1 := ASK SELF TO getQindex;
  localParam := getFromQ(temp1);
  WHILE(localParam <> NILOBJ)
    ASK servRes TO query(temp);
    IF(temp <> NS)
      state := BUSY;
      WAIT DURATION timeComputation(localParam);
      END WAIT;
      count := count + 1;
```

```
        OUTPUT("******************************");
        OUTPUT("FINISH : ",node.name, " ", count," ",SimTime());
        str := ASK localParam TO getName;
        OUTPUT("DISPOSED ",count," ",str);
        OUTPUT("******************************");
        OUTPUT("");
        DISPOSE(localParam);
        temp1 := ASK SELF TO getQindex;
        localParam := getFromQ(temp1);
      ELSE
        localParam := NILOBJ;
      END IF;
    END WHILE;
    state := IDLE;
  END IF;


  END METHOD;
  ASK METHOD getFromQ(IN qindex : INTEGER):parameterObj;
  VAR
  par:parameterObj;
  BEGIN
  IF (node.inqueue = NILARRAY)
    OUTPUT("inQ == NilArray");
    RETURN NILOBJ;
  END IF;
  IF node.inqueue[qindex].First = NILOBJ
   OUTPUT("node.inQ[qindex]=NILOBJ");
    RETURN NILOBJ;
```

```
ELSE
   par :=ASK node.inqueue[qindex] TO First;
  OUTPUT("REMOVED ",par.getName," from ",node.name);
  RETURN ASK node.inqueue[qindex] TO Remove;
END IF;
END METHOD;
ASK METHOD  getQindex : INTEGER;
VAR
BEGIN
IF (node.inqueue = NILARRAY)
   OUTPUT("inQ == NilArray");
   RETURN -1;
END IF;
RETURN LOW(node.inqueue);
END METHOD;
ASK METHOD timeComputation(IN param : parameterObj):REAL;
VAR
BEGIN
   RETURN 1.0;
END METHOD;
END OBJECT;
END MODULE.
```

{Mproj.mod}

```
MAIN MODULE proj;

FROM base IMPORT gateNodeObj;

FROM base IMPORT intersectionNodeObj;

FROM base IMPORT linkObj,baseObj;

FROM condition IMPORT gateIncondObj, gateOutcondObj;

FROM condition IMPORT intersectionIncondObj, intersectionOutcondObj;

FROM condition IMPORT conditionObj;

FROM genericqueue IMPORT genericQueueObj;

FROM server IMPORT gateServerObj, interServerObj,serverObj;

FROM resource  IMPORT gateResObj, intersectionResObj;

FROM parameter IMPORT parameterObj,entityObj,recObj;

FROM SimMod IMPORT StartSimulation,SimTime,StopSimulation;


TYPE

 GeneratorObj = OBJECT

   TELL METHOD initSimulation;

   TELL METHOD GenCustomers;

  END OBJECT;


VAR


i,j,k: INTEGER;

ch :CHAR;

gate : gateNodeObj;

intersection : intersectionNodeObj;

gatein : gateIncondObj;

gateout : gateOutcondObj;
```

```
interin : intersectionIncondObj;

interout : intersectionOutcondObj;

gateInQ :  genericQueueObj ;

gateOutQ : genericQueueObj ;

interInQ :  genericQueueObj;

interOutQ : genericQueueObj;

gateserver : gateServerObj;

interserver : interServerObj;

gateRes : gateResObj;

gateQinArray: ARRAY INTEGER OF genericQueueObj;

gateQoutArray: ARRAY INTEGER OF genericQueueObj;

interQinArray: ARRAY INTEGER OF genericQueueObj;

interQoutArray: ARRAY INTEGER OF genericQueueObj;

interRes : intersectionResObj;

link : linkObj;

linkInQ :  genericQueueObj;

linkOutQ :  genericQueueObj;

linkserver : serverObj;

linkin : conditionObj;

linkout : conditionObj;

linkQinArray: ARRAY INTEGER OF genericQueueObj;

linkQoutArray: ARRAY INTEGER OF genericQueueObj;

gateChildren : ARRAY INTEGER OF baseObj;

linkChildren : ARRAY INTEGER OF baseObj;

interChildren: ARRAY INTEGER OF baseObj;

result: BOOLEAN;
```

```
{parameter obj}

transporter: parameterObj;

transArray: ARRAY INTEGER OF parameterObj;




OBJECT GeneratorObj;

 TELL METHOD initSimulation

 VAR

 endtime : REAL;

BEGIN

  endtime := 300.0;

  WAIT DURATION  endtime;

  END WAIT;

  StopSimulation;

END METHOD;


 TELL METHOD GenCustomers

 VAR

 ch1 : CHAR;

 tbool: BOOLEAN;

 BEGIN

  tbool := FALSE;

 {  NEW(transArray, 1..100);}

      OUTPUT("TIME", SimTime());

      IF tbool <> TRUE

         tbool := TRUE;

         FOR i := 1 TO 100

            NEW(transporter);
            ASK transporter TO setName(INTTOSTR(i));

            { transArray[i] := transporter;}

            result :=   ASK gate TO accept(transporter);

            WAIT DURATION 0.50;

            END WAIT;

         END FOR;

      END IF;

  END METHOD;  { GenCustomers }

END OBJECT;


  custGenerator       : GeneratorObj;

BEGIN


{initialize}

{gate node object}

NEW(gate);

ASK gate TO setName("Gate");


   {in condition obj}

   NEW(gatein);

      {in queue object}

      NEW(gateQinArray,1..1);

      NEW(gateInQ);

      gateQinArray[1] := gateInQ;

   ASK gatein TO init(gateQinArray);


   {out condition obj}

   NEW(gateout);
```

```
    {out queue object}
    NEW(gateOutQ);
    NEW(gateQoutArray, 1..1);
    gateQoutArray[1]:= gateOutQ;
  ASK gateout TO init(gateQoutArray);

  {server}
  NEW(gateserver);
    {Resource}
    NEW(gateRes);
    ASK gateRes TO init(gate);

  ASK gateserver TO init( gate);
  ASK gateserver TO initRes( gateRes);

ASK gate TO init(gatein, gateout,gateQinArray,gateQoutArray,gateserver);
{end of gate init}




{intersection node object}
NEW(intersection);
ASK intersection TO setName("Intersection");

  {in condition obj}
  NEW(interin);
    {in queue object}
    NEW(interQinArray,1..1);
```

```
      NEW(interInQ);

      interQinArray[1] := interInQ;

   ASK interin TO init(interQinArray);


   {out condition obj}

   NEW(interout);

      {out queue object}

      NEW(interOutQ);

      NEW(interQoutArray, 1..1);

      interQoutArray[1]:= interOutQ;

   ASK interout TO init(interQoutArray);


   {server}

   NEW(interserver);

      {Resource}

      NEW(interRes);

      ASK interRes TO init(intersection);


   ASK interserver TO init( intersection);

   ASK interserver TO initRes(interRes);


ASK intersection TO init(interin, interout,interQinArray, interQoutArray, interserver);

{end of intersection init}


{link object}

NEW(link);

ASK link TO setName("Link");
```

```
 {in condition obj}
 NEW(linkin);
   {in queue object}
   NEW(linkQinArray,1..1);
   NEW(linkInQ);
   linkQinArray[1] := linkInQ;
 ASK linkin TO init(linkQinArray);

 {out condition obj}
 NEW(linkout);
   {out queue object}
   NEW(linkOutQ);
   NEW(linkQoutArray, 1..1);
   linkQoutArray[1]:= linkOutQ;
 ASK linkout TO init(linkQoutArray);

 {server}
 NEW(linkserver);

 ASK linkserver TO init(link);

ASK link TO init(linkin, linkout,linkQinArray, linkQoutArray, linkserver);

{end of link init}
{set up model structure}

NEW(gateChildren,1..1);
NEW(linkChildren,1..1);
```

```
interChildren := NILARRAY;

gateChildren[1] := baseObj(link);

linkChildren[1] := baseObj(intersection);


ASK gate TO setChildren(gateChildren);

ASK link TO setChildren(linkChildren);

ASK intersection TO setChildren(interChildren);


NEW(custGenerator);


{main program}

TELL custGenerator TO initSimulation;

TELL custGenerator TO GenCustomers;

StartSimulation;


{end main program}

OUTPUT("THE END");


ASK gateInQ TO Empty;

ASK gateOutQ TO Empty;

ASK interInQ TO Empty;

ASK interOutQ TO Empty;

ASK linkInQ TO Empty;

ASK linkOutQ TO Empty;


{FOR i := 1 TO 100

   transporter:=transArray[i];

   DISPOSE(transporter);
END FOR;


DISPOSE(transArray);
}
{cleanup}
DISPOSE(gatein);
DISPOSE(gateQinArray);
DISPOSE(gateInQ);
DISPOSE(gateout);
DISPOSE(gateOutQ);
DISPOSE(gateQoutArray);
DISPOSE(gateserver);
DISPOSE(gateRes);
DISPOSE(gate);
DISPOSE(intersection);
DISPOSE(interin);
DISPOSE(interQinArray);
DISPOSE(interInQ);
DISPOSE(interout);
DISPOSE(interOutQ);
DISPOSE(interQoutArray);
DISPOSE(interserver);
DISPOSE(interRes);
DISPOSE(link);
DISPOSE(linkin);
DISPOSE(linkQinArray);
DISPOSE(linkInQ);
DISPOSE(linkout);
DISPOSE(linkOutQ);

DISPOSE(linkQoutArray);

DISPOSE(linkserver);

DISPOSE(gateChildren);

DISPOSE(linkChildren);


 OUTPUT;

  OUTPUT("Enter a character to quit.");

  INPUT(ch);


END MODULE.
```

CURRICULUM VITA
for
BRIAN DOUGLAS JONES

NAME: Brian Douglas Jones

DEGRESS:

Master of Science (Computer Engineering), Old Dominion University, Norfolk, Virginia, August 2000.

Bachelor of Science (Computer Engineering), Old Dominion University, Norfolk, Virginia, December 1997.

Associates of Science (Computer Engineering), Tidewater Community College, Portsmouth, Virginia, December 1995.

PROFESSIONAL CHRONOLOGY:

Space and Naval Warfare System Center, San Diego, California.
Deputy Chief of Information Systems for the Joint Warfighting Center, August 1998 – Present.

Old Dominion Research Foundation, Norfolk Virginia.
Research Assistant for the Virginia Modeling and Simulation Center, Suffolk, Virginia. May 1998 – May 1999.