Proceedings of the 1993 Winter Simulation Conference
G.W. Evans, M. Mollaghasemi, E.C. Russell, W.E. Biles (eds.)

BUILDING OBJECT-ORIENTED SIMULATIONS WITH C++

Jeffrey A. Joines
Stephen D. Roberts

Department of Industrial Engineering
Box 7906
North Carolina State University
Raleigh, NC 27695-7906, U.S.A.

Kenneth A. Powell, Jr.

Simulation Solutions
6560 English Oaks Drive
Raleigh, NC 27615

ABSTRACT

This tutorial shows how to build object-oriented simulation models in C++. A set of object classes, called YANSL, have been written in C++ for network based simulation. Objects from YANSL are used to create a network queuing model of the Harbor problem. Models that use YANSL have the "look and feel" of existing network simulation languages, but possess the benefits of an object-oriented design, including the use of classes, inheritance, encapsulation, polymorphism, runtime binding, and parameterized typing. These concepts are illustrated through several seemingly difficult embellishments to the example, which actually extend the language. Object-oriented simulations provide full accessibility to the base language, faster executions, portable models and executables, a multi-vendor implementation language, and a growing variety of complementary development tools.

1 INTRODUCTION

The idea of an "object-oriented" simulation has great intuitive appeal because it is very easy to view the real world as being composed of objects. In a manufacturing cell, objects that should come to mind include the machines, the workers, the parts, the tools, and the conveyors. Also, the part routings, the schedule, the work plan, and other information items could be viewed as objects.

It is also quite easy to describe existing simulation languages using object terminology. A simulation language or simulation package provides a user with a set of pre-defined object classes from which the simulation modeler can create needed objects. For example, a network-based queuing language will typically view a system as having entity objects that travel through a network of queue objects, being served by resource objects. Using the language (object classes), the modeler would declare the network by defining the node objects and their connecting branch objects. The node objects would be described as queues and activities, with and without resources, and sinks (where entities leave the network). Pre-defined entity objects, sometimes called transactions, can be made to arrive to the network through source nodes. Most languages permit attributes that can be altered to be attached to the transactions. Resource objects and their behavior would need to be defined. Simulation support objects include the distributions, the global variables, statistical tables and histograms. The modeler creates objects and specifies their behavior through the parameters available. The integration of all the objects into a single packages provides the complete simulation model.

Some simulation packages/languages provide for special functionality, such as that needed for manufacturing simulations. Object classes may be defined for machines, conveyors, transporters, cranes, robots, and so forth. These special objects have direct usefulness in particular situations. Simulation packages centered around such objects are directed at specific vertical application areas such as AGVs, robotics, FMS, etc.

1.1 Two Critical Weaknesses of Existing Languages

Most simulation languages suffer from two important weaknesses. Because the languages offer pre-specified functionality produced in another language (assembly language, C, FORTRAN, etc.), the user cannot access the internal function of the language. Instead, a user must rely on vendor description of the algorithms, procedures, and data used to implement the concepts. Only the vendor can make modifications to the internal functionality. Second, users have limited opportunity to extend an existing language feature. Some simulation languages allow for certain programming-like expressions or statements, which are inherently limited. Most languages allow the insertion of procedural routines written in other general-purpose programming languages. However, none of this is fully satisfactory because, at best, they provide only procedural extension. For example, it might be easy to write a procedure to make a complicated computation of an activity time, but if you wanted to create a different activity, there is insufficient access to existing activity information. Any procedure written cannot use and change the behavior of a pre-existing object class and any new object classes defined by a user in general programming language do not coexist directly with vendor code. At a more fundamental level, the language structure may be inherently awkward for some purposes. For instance, consider the difficulties of modeling a tennis match using a queuing network language. The new Arena software (Pegden and Davis 1992) provides a template approach to representing blocks of Siman statements so that higher levels of models can be used directly (as can their graphical representation). However, lower levels still remain bound to the Siman language.

1.2 A Way to Overcome these Problems

Object-oriented simulation deals directly with the limitation of extensibility by permitting data abstraction as well as procedural abstraction. Data abstraction means that new data types with their own behavior can be added arbitrarily to the programming language. When the new data type is added, it can assume just as important role as implicit data types. For example, a user-defined data type that manages complex numbers can be as fundamental to a language ("first class") as the implicitly defined integer data type. In the simulation language context, a new user-defined robot class can be added to a language that contains standard resources without compromising any aspect of the existing simulation language and the robot may be used where a more complex resource was needed.

1.3 Purpose of this Paper

The purpose of this paper is to illustrate object-oriented simulation using the C++ language (this paper is a modification of the one from last year's conference, see Joines, Powell, and Roberts 1992). C++ is an object-oriented extension to the C programming language (Lippman 1991). We will use C++ to illustrate the "extensive/intensive" nature of object-oriented simulation (OOS) within the framework of the popular network-based simulation approach.

2 YET ANOTHER NETWORK SIMULATION LANGUAGE (YANSL)

In order to illustrate the importance of object-oriented simulation, we begin by describing a network queuing simulation package of roughly the power of a GPSS (Schriber 1991), SLAM (Pritsker 1986), SIMAN (Pegden, Shannon, and Sadowski 1990), or INSIGHT (Roberts 1983), but without the "bells and whistles." Users familiar with any of these languages should recognize, however, that what we present is a very powerful alternative. For convenience, we call this language YANSL (Joines, Powell, and Roberts 1992).

2.1 Basic Concepts and Objects in YANSL

When modeling with YANSL, the modeler views the model as a network of elemental queuing processes (graphical symbols could be used). Building the simulation model requires the modeler to select from the predefined set of node types and integrate these into a network. The network is constructed about a set of entities which are called transactions that flow through the network. The transaction has exactly the same interpretation it has in the other simulation languages. The transactions are routed through the network according to some logic that represents the system being modeled. Transactions may require resources to serve them at activities and thus may need to queue to await resource availability. Resources may be fixed or mobile in YANSL, and one or more resources may be required at an activity. Unlike some network languages, resources in YANSL are active entities, like transactions, and may be used to model a wide variety of real-world items (notice this feature is, in fact, more powerful than some existing languages). Although you may regard YANSL as pale in comparison with existing simulation languages, we will demonstrate how easily a knowledgeable user can extend its power and functionality.

2.2 The Harbor Problem

Ships on the ocean enter and leave a harbor (the interested reader can contrast this example with the TV Inspection problem considered in Joines, Powell, and Roberts 1992). Within the harbor, they dock at berths where they load and unload cargo. Just as in other network languages, transactions are used to represent the ships. The resource needed is the berth. There are three berths. The network is composed of a source node which describes how the ships arrive, a queue for possible wait for a berth, the unload/load activity with its requirement for the berth, and a sink where ships leave the harbor. Transactions branch from the source to the berth queue, are served at the unload/load activity, and branch to the sink. The data used in the simulation are the interarrival time of ships, which is exponentially distributed with a mean interarrival time of 5.8 minutes, and the service time (unload/load), which is exponentially distributed with a mean of 5.0 minutes.

2.3 The YANSL Model

The YANSL network has all the graphical and intuitive appeal of any network based simulation language. A graphical user interface could be built to provide "convenient" modeling with error checking and on-line help offered to the user. Whatever the modeling system used, the ultimate computer readable representation of the model would appear as follows:

```
/******************************
***   Simple Tanker Problem
******************************/

#include "simulation.h"

main()
{

// SIMULATION INFORMATION
Simulation     tankerSimulation(1);

// DISTRIBUTIONS
Exponential    interarrival( 5.8 ),
               unloadLoadTime( 5 );

// RESOURCES
Resource< PRIORITY > berth[ 3 ];

ResourceSelection< ORDER >   berthList;
      berthList.addResource( berth, 3 );

// NETWORK NODES

   /** Ships Arrive **/
Source< Transaction, DETERMINISTIC >
      tankerSource( interarrival, 0.0, 480 );
      // Begin at 0.0 and end at 480.0

   /** Wait for a Berth **/
Queue< FIFO >  berthingQueue;
   /** Load/Unload **/
Activity< DETERMINISTIC >
           unloadLoad( unloadLoadTime );
      unloadLoad.addRequirement( berthList );
      berthingQueue.addActivity( unloadLoad );

   /** Ships Leave **/
Sink finish;


// NETWORK BRANCHES
tankerSource.addBranch( berthingQueue );
unloadLoad.addBranch(finish);

// RESOURCE SERVICES
berth[ 0 ].addQueue( berthingQueue );
berth[ 1 ].addQueue( berthingQueue );
berth[ 2 ].addQueue( berthingQueue );

// RUN the simulation
tankerSimulation.run();
}
```

The previous model has the properties of any network simulation language. There is an almost one-to-one correspondence to the entities describing the problem. No more information is needed than necessary (compare this input with other simulation languages). The statements are readable and follow a relatively simple format. The pre-defined object classes grant the user wide flexibility.

While the "statements" in YANSL are very similar to those in SIMAN, SLAM, or INSIGHT, it is all legitimate C++ code -- which will be discussed in detail later. Also, this model runs in less than half the time a SIMAN model runs on the same machine! But the real advantage of YANSL is its extensibility.

2.4 The Objects and their Specification

Lets take a closer look at the YANSL "statements." The model is enclosed in a recognizable C framework, namely having a `#include` statement that includes all that the simulation requires, a `main()` function header, and `{}` which enclose the block of code (YANSL statements). This framework is left only to reveal it is C++ code, as even these could be eliminated by the C preprocessor commands that would take a `Begin` and `End` and `StartSimulation` for the conventional C tokens. Also, the more clever programmer could accept other more intuitive information and convert it to the YANSL format.

The YANSL simulation consists basically of two types of statements. The first is the declaration of objects in the model and the second is function calls to structure the model. The same division of statements occurs in existing simulation languages. The only order requirement for statements is that an object must be declared before it is used. Thus, we decided to order the statements by declaring first the general information needed (like the distributions) and then we specified the network entities (resources, nodes, and branches).

2.4.1 Object Declarations

The objects in YANSL are declared in a form consistent with C and C++ . The object class is specified first, then the objects are named. Initialization of specific objects are done in parentheses. For instance,

```
Exponential    interarrival( 5.8 ),
               unloadLoadTime( 5 );
```

creates two exponential distributions whose names are `interarrival` and `unloadLoadTime` and whose initialization parameters are given in parenthesis. It is important to note that the mean unload/load time is specified as an integer 5, but in fact it is assumed to be floating point. This illustrates a simple case of "overloading." Here, initialization of the interarrival object can take either an integer or a floating point parameter. In object-oriented terminology, exponential objects are initialized by either an integer or floating point object.

Some object declarations appear more complex because the object class is also parameterized by information in <>. In object-oriented terminology, these are called "parameterized types." A parameterized type is used when the class itself needs some information. This parameterization should not be confused with initialization of objects where the object needs some information. As an example, consider

```
Activity< DETERMINISTIC >
                unloadLoad( unloadLoadTime );
```

where the `Activity` class needs some branching method class called `DETERMINISTIC`, while the object `unloadLoad` is initialized with a reference to the `unloadLoadTime` object. Notice that a class will be parameterized with another class (and possibly objects), while an object is parameterized with another object.

Because YANSL is really C++, all the "built-in" classes from C++ are directly available to the YANSL user. These include `integer`, `float`, `char`, etc. Further, in an effort to give our YANSL users a full range of "nice" basic classes, such classes as `String`, `List`, and dynamic `Array` with range checking are also available. Because an object-oriented language doesn't distinguish any differently between the C++ classes and the ones we have added, use of all these classes is very similar. In the computer literature, this property of having user objects treated like built-in objects means everything is treated as a "first class."

2.4.2 Using the Objects

The other "statements" in YANSL provide direct use of the objects. These are actual function calls in C++. In object-oriented terminology, it is called "message passing." For example,

```
berth[ 2 ].addQueue( berthingQueue );
```

the message `addQueue` with `berthingQueue` object as a parameter is sent to the `berth[ 2 ]` object. In C++ terminology the `addQueue` function in the `berth[ 2 ]` object is passed the `berthingQueue` object. The purpose of this message/function is for the berth to know that it is to service the queue of the unload/load activity when it is free to choose what to do.

Notice the "encapsulation" of functionality. The resource class obviously has the ability to accept information about what a resource is to do when it is available. All this is contained in the resource class. Suppose you want some different functionality of resource behavior. Now all the changes would be confined to the code in the resource class.

The YANSL functions are used to specify the functioning of the objects in the simulation. The `addQueue` specifies what queues the resources serve, the `addBranch` specifies how transactions branch from the departure nodes, the `addActivity` associates the activity with the queue, and the `addRequirement` specifies the resource requirements at the activities. Finally, the `tankerSimulation.run` causes the simulation execution to begin.

2.5 Running the Simulation

The prior model is compiled under a C++ compiler (a compiler should be AT&T version 3.0 compatible), linked with the YANSL simulation library, and executed. Currently, the YANSL simulation library has been compiled under Borland C++ 3.1 (Borland 1992). C++ is strongly typed, so error checking is very good.

Also, the simulation is easily linked into other C++ libraries which may be used for graphics and statistical analysis. In a sense, YANSL has the same relationship to C++ that GASP IV (Pritsker 1974) has to FORTRAN. The major difference is that whereas GASP was a set of FORTRAN functions that the model builder called, YANSL is a set of both the functions and their data organized about simulation objects (rather than simulation functions). As such, YANSL is more like SLAM, but fully compatible with the entire C++ language, rather than simply permitting general procedures to be "inserted" into a specific simulation structure like SLAM.

3 CLASSES AND THEIR USE

The class concept is fundamental to object-oriented software. The classes provide a "pattern" for creating objects. An example is the following Exponential class:

```
#ifndef EXPON_H
#define EXPON_H

#include "random.h"
```

```
/*  expon.h contains Class Exponential. This
class describes an inverse transformation
generator for Exponential variables. */

class Exponential: public Random
{
public:
   Exponential(double, unsigned int=0, long=0);
   Exponential( int, unsigned int=0, long=0 );
   virtual double sample();
   void setMu(double initMu) {mu = initMu;}
   double getMu() { return mu; }
private:
   double mu;
};

#endif
```

The class definition determines the properties of an object.

3.1 Class Properties

Properties of classes, namely their data objects and functions, are generally grouped into "public" and "private" sections (C++ also permits another grouping called protected). The public properties can be accessed from outside the object. The private properties are information kept strictly locked within an object and are available only to object functions. For example, the double object mu is private and cannot be directly obtained. However, a public function called getMu does return the value of mu. Making a property private restricts unauthorized use and encapsulates the object.

3.2 Inheritance

The Exponential class was not defined "from scratch." For instance, it doesn't say anything about its use of random numbers or from where the random numbers come. Because the random number generator establishes the source of randomness for all random processes, it is defined in its own class. Hence, the Exponential class is derived from the Random class so the Exponential class has access to all the public properties of the Random class without having to re-code them. This use of prior classes is called "inheritance." In fact, this inheritance makes the Exponential class a "kind of" Random class. In object-oriented terminology this is a "is-a" relationship.

The other major kind of relationships between two classes is the "has-a." In the case of the Exponential, the Exponential has a double object called mu. A has-a relationship is not the result of inheritance.

3.3 Run-time Binding

The sample() function is specified as a virtual function in Exponential because we don't want to write a specific function for each class that obtains a sample from the variate generator. Therefore, the sample function will, at run-time, decide from which random variate to sample. This binding the variate to the sample at run-time is also called "delayed" or "run-time" binding. Run-time binding may extract a small run-time penalty, but makes this entire specification of sampling from variates much easier to write, maintain, and use.

3.4 Construction and Initialization of Objects

When an object from a class is needed, there needs to be a way to construct and initialize it. The function that does this is called a "constructor" and C++ will provide one if it isn't included in the class definition. In the case of the Exponential class, there are two constructors. One takes a double and the other takes an integer. Notice that some of the arguments have specified defaults, so the user doesn't have to specify all the potential features of an Exponential object (these additional arguments pertain to the control of the random number stream). Within the constructors (details not shown), space is allocated for the object and parameters are assigned.

Although, not used in Exponential, C++ permits user specified destructors. A destructor will clean-up any object responsibilities (like collecting statistics) and deallocate the space.

3.5 Polymorphism

The Exponential class has two constructors so users may specify either floating point or integer arguments for the mean interarrival time. Although it is not necessary in this case (C++ will make the right conversions), it does illustrate the use of polymorphism -- where the same property applies to different objects. Thus, the Exponential object is appropriately specified, regardless of whether an integer or double is given. This encapsulation of the data makes the addition of new types for parameters very easy and localized.

4 EMBELLISHMENTS TO HARBOR MODEL

To illustrate the broader use of an object-oriented simulation language, we present several embellishments to the Harbor problem (the reader is referred to the paper in last year's conference that considered floating resources, transaction attributes, assignment nodes, queue ranking, dependent service times, and grouped transactions within the context of TV inspect and repair problem). Although the embellishments may appear very complicated, they are handled easily and provide direct extensions to YANSL modeling capabilities.

4.1 More Flexible Resources

Some resources can service more than one queue and choose their next service. Further, transactions can continue use of their resources, producing activities that are required for some resources and chosen by others. Suppose now that ships entering the harbor must be pulled by a tug. The tug brings the ship from the ocean into the harbor and then to the dock where the ship berths. Also, when a ship finishes unload/load and is ready to leave, a tug must be summoned before the ship deberths and is pulled out into the ocean. Thus the ships now experience two queues which wait for the tug: wait to berth, wait to leave the berth. The tug becomes a new resource. Now this tug must serve at two new activities, the berthing activity and the deberthing activity. These two new activities also require the berth (as does the unload/load activity). The revised model is given below:

```
main()
{
// SIMULATION INFORMATION
Simulation harborSimulation(1);

// DISTRIBUTIONS
Exponential    interarrival( 4.5 ),
               deberthTime( 8.5 ),
               berthTime( 8.5 ),
               unloadLoadTime( 5 );

// RESOURCES
Resource< PRIORITY > berth[ 3 ], tugboat;

ResourceSelection< ORDER >   berthList1;
      berthList1.addResource( berth, 3 );

ResourceSelection< BUFFER >  berthList2;
      berthList2.addResource( berth, 3 );

// NETWORK NODES

   /**  Ship Arrive **/
Source< Transaction, DETERMINISTIC >
      tankerSource( interarrival, 0.0, 480 );
      // Begin at 0.0 and end at 480.0

   /**  Berthing **/
Queue< FIFO > berthingQueue;
Activity< DETERMINISTIC >
            berthing( berthTime );
      berthing.addRequirement( tugboat );
      berthing.addRequirement( berthList1 );
      berthingQueue.addActivity( berthing );

   /**  Unload/Load **/
Activity< DETERMINISTIC >
            unloadLoad(  unloadLoadTime );
      unloadLoad.addRequirement( berthList2 );

   /**  DeBerthing **/
Queue< FIFO > deBerthQueue;
CapturedActivity< DETERMINISTIC >
            deberthing( deberthTime );
      deberthing.addRequirement(
            berthList2 , CAPTURE );
      deberthing.addRequirement(  tugboat );
      deBerthQueue.addActivity( deberthing );

   /**  Ships Leave **/
Sink finish;

 // NETWORK BRANCHES

tankerSource.addBranch( berthingQueue );
berthing.addBranch( unloadLoad );
unloadLoad.addBranch( deBerthQueue );
deberthing.addBranch( finish );

// RESOURCE SERVICES
berth[ 0 ].addQueue( berthingQueue );
berth[ 1 ].addQueue( berthingQueue );
berth[ 2 ].addQueue( berthingQueue );
tugboat.addQueue( deBerthQueue );
tugboat.addQueue( berthingQueue );

// RUN the simulation
harborSimulation.run();
}
```

The revised model now declares the `tugboat` as a resource. The tug services the `deBerthQueue` before the `berthingQueue` since it uses a `PRIORITY` decision (note that the `deBerthing` is added first). Resource selection by ships for berths is given by `ORDER` at the berthing activity but once the berth is selected, that berth stays with the ship until the ship is deberthed (in the real world, it is the opposite). This association is accomplished by making sure the same berth is used in sequential activities through the `BUFFER` resource selection. Of special note is that the `deBerthing` activity is a `CapturedActivity`, which insures that the berth is kept even if the transactions must stay in the queue to wait for a tug. Note that the `addRequirement` takes both a resource and a type of requirement. This overloading of the function argument is an example of polymorphism applied to user-defined classes. Therefore, a user of YANSL now may specify a requirement involving several resource alternatives with the exact same form used to specify a single resource and new decision rules may be easily included.

4.2 Gating Resources

The previous embellishment took advantage of general YANSL features. Some embellishments will not. Consider, for instance, a complication in the harbor is that storms sometimes come up preventing the tugs from berthing or deberthing ships. The time between storms and the duration of storms can be simply declared as follows:

```
Exponential    stormInterOccurance( 0.25 );
Triangular     stormDuration( 2, 2.5, 4 );
```

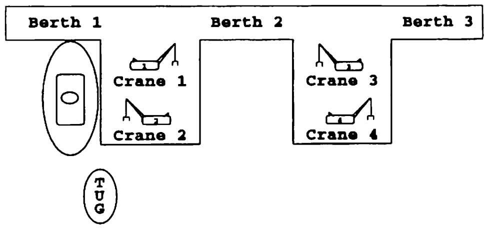
However the modeling of the storms requires more specific development. To model the actual storms, we use the special `Storm` object class developed for this problem and gate the tugboat resource by sending a `gateResource` message:

```
Storm minorStorm(
               interOccurance, stormDuration );
      minorStorm.gateResource( tugboat );
```

Obviously, a Storm class is rather narrow in application and specific to the harbor problem. When the storm comes, it prevents any new service by the tug. An idle tug is prevented from doing any new service, while a busy tug will finish its current service and then become idle. When the storm is over, the idle tug is activated.

4.3 Greater Cooperation among Entities

One of the most difficult modeling problems occurs when there is a great deal of cooperation among resources and transactions. In the case of the harbor problem, consider the following configuration of the berths:



In this example, we have added load/unload cranes, which are modeled by a new set of resources. The cranes are needed in load/unload activities. However, their operation is quite complex. Cranes #1 and #3 prefer to serve Berth #2. Crane #1 can "help" with work on Berth #1 and Crane #3 can help with work on Berth #3 if there is no work on Berth #2. Crane #2 prefers to service Berth #1, but can help with work on Berth #2 if there is no work on Berth #1. Also Crane #4 prefers to service Berth #3, but can help with Berth #2 if there is no work on Berth #3.

This example illustrates how cranes are associated with berths and how their association may be optional and temporary. The `ResourceSelection` provides a means for the transactions to select among alternative individual resources. However, it does not allow selection among teams of resources. In order to specify these relationships/dependencies among resources, a new `ResourceTeam` class is derived from both `ResourceBase` and `Requirement` (this is an example of multiple inheritance). The `Requirement` class which is also used in the `Activity` node holds the resource requirements for the team. Recall, these requirements can be individual resources as well as `ResourceSelection`. The `ResourceTeam` is derived from the abstract class `ResourceBase` to provide the mechanism to be a resource. The resource teams for the three berths and cranes can be simply defined as follows:

```
ResourceTeam teams[ 3 ];

/** This sets up the first team **/
teams[ 0 ].addRequirement( berth[ 0 ] );
teams[ 0 ].addRequirement( crane[ 1 ],PREEMPT);
teams[ 0 ].addRequirement(crane[ 0 ],OPTIONAL);
```

Since `ResourceTeam` is derived from `ResourceBase` and simply acts as an individual resource, we can now use `ResourceSelection` to select among teams of resources without making any more modifications. This new class works as if it was an original part of the YANSL design.

```
ResourceSelection< ORDER > berthList1;
bertList1.addResource( teams, 3 );

ResourceSelection< BUFFER > berthList2;
bertList1.addResource( teams, 3 );
```

5 IMPLEMENTATION

The embellishments in Section 4 have employed a number of features of object oriented simulation. The embellishments reveal no distinction between the base features of YANSL and its extensions, illustrating the "seamless" nature of user additions. In this section, we want to describe in more detail the actual C++ implementation of some of these developments.

5.1 Template Issues

Parameterized types are used throughout YANSL to give classes generality. Parameterized types are created by class *templates* so that the ultimate specification of a class is not known until that class is declared in the model to create the object (both the class and the object are created). Templates make it easy for a user to specify a kind of class rather than having a whole bunch of classes whose similarities are greater than their differences. For instance, consider the resource selection class:

```
template< class RSM >
class
   ResourceSelection : public RequirementBase
{
  protected:
    RSM resSelMeth;

  public:
    ResourceSelection(){}
    virtual void addResource(
                ResourceBase &rptr );
      ResourceBase *resourceAvailable();
};
```

The `template` parameterizes the `ResourceSelection` class for any `RSM` (resource selection method). Now a new resource selection method can be developed to work with any resource selection, such as `BUFFER`:

```
class
BUFFER : public virtual ResourceSelectionMethod
{
  public:
        BUFFER();
        virtual ResourceBase
                *resourceAvailable();
};
```

The virtual function permits any class derived from `BUFFER` to determine resource availability. Some network simulation languages approach the need for parameterized classes by having more general node types, like an "operation" node, but these general types cannot, in general, yield specific objects -- only their subtypes create objects (in C++, such a class would be "pure virtual class").

5.2 Inheritance and Derived Classes

Inheritance provides a fundamental means of relating object classes. In the development of the storm activity, a storm class was needed:

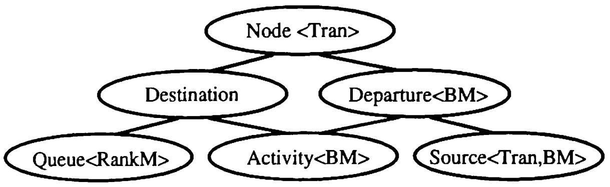
```
class Storm : public virtual Node
{
  public:
     Storm( Random& , Random& , double = 0.0);
     void executeLeaving( Transaction * );
     BOOL executeEntering( Transaction * );
     void gateResource(ResourceBase &rptr);

  protected:
     Random              *occVariate;
     Random              *durVariate;
       // ...
};
```

Here, the class `Storm` is derived from the `Node` class since storm "is-a" kind of node and thus inherits all the properties of the `Node` class. Because the `Storm` is a `Node`, `Storm` can do all the things a `Node` can do, such as be placed in a network. In contrast, storm "has" two random data objects corresponding to the occurrence and duration of the storm. Recall our discussion of is-a and has-a in Section 3.2.

5.3 Multiple Inheritance and the Class Hierarchy

In YANSL, node classes are formed in a class "hierarchy." This hierarchy starts with a broad division of nodes and specifies more specific nodes lower in the hierarchy. Nodes lower in the hierarchy inherit the properties of the nodes above them. A portion of that hierarchy is given below:



In the hierarchy, nodes are broadly defined as departure and destination nodes. Departure nodes have branches connected to them and therefore need a "BranchingMethod (BM)." Sink and queue nodes can have transactions branched to them and are therefore destination nodes. An activity node is both a departure and a destination node, so it inherits from both the departure and destination node classes. This inheritance from multiple parents is called "multiple inheritance." Not all object-oriented languages permit multiple inheritance like C++. Portions of the `Activity` class are:

```
template< class BM>
class Activity :
       public virtual Departure< BM >,
       public virtual Destination
{
  public:
       Activity( Random& );
       int resourcesAvailable();
       void addRequirement(
              RequirementBase &rsPtr );
       virtual void executeLeaving(
              Transaction * );
       virtual BOOL executeEntering(
              Transaction * );
       ResourceBase *getResBuffer();
  protected:
       Random                 *actVariate;
       Requirements            requirements;
       ResourceBase           *bufferStorage;
       //...
};
```

Multiple inheritance is specified in the header of the class definition.

5.4 Virtual Functions and Run Time Binding

The `executeEntering` and `executeLeaving` are virtual functions in Departure and Destination classes that act as place holders, permitting derived nodes special functionality as transactions enter and leave. These functions were especially useful in the `CapturedActivity` class (derived from Activity), where the `executeEntering` was used to capture resources:

```
template< class BM >
class CapturedActivity : public Activity< BM >
{
  public:
        CapturedActivity( Random& );
        void addRequirement( RequirementBase
                &rsPtr, int flag = 0 );
        virtual BOOL executeEntering(
                Transaction * );
 protected:
        CapturedRequirments    caprequirements;
};
```

Therefore, a transaction enters a node using its branching:

```
branch.nextNode()->executeEntering( tptr );
```

Now, whatever the node, the appropriate executeEntering function is executed, even though it had not been explicitly identified.

5.5 Static and Dynamic Objects

The objects in a YANSL simulation can be either static or dynamic. The static objects from the Harbor example are declared within the main() function and are created at compile time. In contrast to the static objects, dynamic objects are created during run time whenever the simulation logic requires their existence. For example, the exact number of Transactions traveling through the network is unknown at compile time, so it is convenient to create these objects when they enter the network and destroy them as they exit. In C++, this is accomplished through the new and delete operators. Transactions, or any of their derived types, are created in Source nodes.

```
template< class T, class BM >
void Source<T,BM>::
        executeLeaving(Transaction *t )
{
// ...
  Transaction *currentTransaction = new T;
  scheduleEvent( currentTransaction,
                interArrivalTime->sample() );
// ...
}
```

The type of Transaction actually created is declared at compile time as a template parameter to the Source node. This allows the modeler ultimate flexibility in the types of Transactions in the network. Transactions as complex as a passenger ship may exist in the same network as a simple cargo ship. This method is preferred over existing simulation languages in which all Transactions are required to have the same number of attributes regardless of their complexity, or lack of complexity. Transactions leave the network though Sink nodes where their existence is terminated with the delete operator and the space allocated is reclaimed for future use. This dynamic construction/destruction process allows as many, or as few, Transactions as needed to exist, up to the limits of memory.

5.6 Related Embellishments

Many more embellishments are simply parallel application of the approaches used in the prior sections. For example, the embellishments shown in the earlier paper (Joines, Powell, and Roberts 1992) could be applied here to the harbor problem including new branching methods, transaction attributes to distinguish ships, queue ranking, transaction grouping, etc. These embellishments can be added for a single use or they can be made a permanent part of YANSL, say YANSL II. In fact a different kind of simulation language, say for modeling and simulating logistics systems, might be created and called LOG-YANSL for those special users. Perhaps the logistics users would get together and share extensions and create a more general LOG-YANSL II. And so it goes! For those of you familiar with some existing network simulation language, consider the difficulty of doing the same.

6 CONCLUSIONS

Modeling and simulation in an object-oriented language possesses many advantages. We have shown how internal functionality of a language now becomes available to a user (at the discretion of the class designer). Such access means that existing behavior can be altered and new objects with new behavior introduced. The object-oriented approach provides a consistent means of handling these problems (other general object oriented languages include Smalltalk (Goldberg and Robson 1989) and Eiffel (Meyer 1992)). Object-oriented simulation systems include Smalltalk, Modsim II (Belanger and Mullarney 1990), and Sim++ (Lomow and Baezner 1991). C++ based simulation packages include Sim++ and Simpack (Fishwick 1992).

The user of a simulation in C++ is granted lots of speed in compilation and execution. The C language has been a language of choice by many computer users and now C++ is beginning to supplant it. With the new C++ standard (Ellis and Stroustrup 1991), all C++ compilers are expected to accept the same C++ language. We can build an executable simulation on one machine and run it on another, only as long as the operating systems are compatible -- you don't need a C++

compiler on both machines. Most commercial simulation languages require some proprietary executive.

Because C++ has many vendors, the price of compilers is low, while the environments are excellent. For example, the Borland package includes a optimizing compiler, a fully interactive debugger, an object browser, a profiler, and an integrated environment that allows you to navigate between a code editor and the other facilities. Also numerous class libraries for windowing, graphics, and so forth are appearing that are fully compatible with C++. Graphical user interfaces for simulation modeling, animation of simulation, and statistical analysis of simulation results could be offered by individual vendors. Their interoperability would be insured by their use of a common means for defining and using objects.

To take full advantage of object-oriented simulation will require more skill from the user. However, that same skill would be required of any powerful simulation modeling package, but with greater limitations.

REFERENCES

Belanger, R., and A. Mullarney. 1990. *Modsim II tutorial.* CACI Products Company, La Jolla, CA.

Borland. 1992. *Borland C++ Version 3.1.* Borland International, Inc. 1800 Green Hills Road, Scotts Valley, CA 95067-001.

Ellis, M, and B Stroustrup. 1991. *The annotated C++ reference manual.* Reading, Massachusetts: Addison-Wesley.

Fishwick, P. A. 1992. Simpack: getting started with simulation programming in C and C++. In *Proceedings of the 1992 Winter Simulation Conference*, ed., J. J. Swain, D. Goldsman, R. C. Crain, and J.R. Wilson, 154-162. Institute of Electrical and Electronics Engineers, Washington, D.C.

Goldberg, A., and D. Robson. 1989. *Smalltalk-80: the language.* Reading, Massachusetts: Addison-Wesley.

Joines, J. A., K. A. Powell, Jr., and S. D. Roberts. 1992. Object-oriented modeling and simulation with C++. In *Proceedings of the 1992 Winter Simulation Conference*, ed., J. J. Swain, D. Goldsman, R. C. Crain, and J.R. Wilson, 145-153. Institute of Electrical and Electronics Engineers, Washington, D.C.

Lomow, G., and D. Baezner. 1991. A tutorial introduction to object-oriented simulation and Sim++. In *Proceeding of the 1991 Winter Simulation Conference,* ed., B. L. Nelson, W. D. Kelton, and G. M. Clark, 157-163. Institute of Electrical and Electronics Engineers, Phoenix, AZ.

Lippman, S.B. 1991. *C++ primer,* Second Edition. Reading, Massachusetts: Addison-Wesley.

Meyer, B. 1992. *Eiffel: the language.* New York:Prentice Hall.

Pegden, C.D., R.E. Shannon, and R.P. Sadowski. 1990. *Introduction to simulation using SIMAN.* New York: McGraw-Hill.

Pedgen, C.D., and D. A. Davis. 1992. Arena: a SIMAN/Cinema-based hierarchical modleing system. In *Proceedings of the 1992 Winter Simulation Conference,* ed., J. J. Swain, D. Goldsman, R. C. Crain, and J.R. Wilson, 390-399. Institute of Electrical and Electronics Engineers, Washington, D.C.

Pritsker, A.A.B. 1974. *The GASP IV simulation language.* New York: John Wiley and Sons.

Pritsker, A.A.B. 1986. *Introduction to simulation and SLAM II,* Third Edition. New York: Halsted Press.

Roberts, S. D. 1983. *Modeling and simulation with INSIGHT.* Indianapolis, Indiana: Regenstrief Institute.

Schriber, T.J. 1991. *An introduction to simulation using GPSS/H.* New York: John Wiley and Sons.

AUTHOR BIOGRAPHIES

JEFFERY A. JOINES is currently pursuing a Ph.D in the Department of Industrial Engineering at North Carolina State University. He received his B.S.I.E, B.S.E.E, and M.S.I.E from NCSU. He is a member of Phi Kappa Phi, Alpha Pi Mu, Tau Beta Pi, Eta Kappa Nu, and IEEE. His interests include object-oriented simulation, artificial neural networks, and genetic algorithms as applied to manufacturing.

KENNETH A. POWELL, JR. is owner and operator of Simulation Solutions, a company specializing in custom simulation software. He received his B.S.I.E and M.S.I.E. from North Carolina State University. His interests include simulation, object-oriented programming, and Microsoft Windows GUI design.

STEPHEN D. ROBERTS is Professor and Head of the Department of Industrial Engineering at North Carolina State University. He received his B.S.I.E., M.S.I.E., and Ph.D. from Purdue University. He is the Modeling Area Editor for *TOMACS.* He has served as Proceedings Editor and Program Chair for the Winter Simulation Conference. He is the TIMS/CS representative to and current Chair of the Board of Directors of WSC.