Proceedings of the 1994 Winter Simulation Conference
ed. J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila

DESIGN OF OBJECT-ORIENTED SIMULATIONS IN C++

Jeffrey A. Joines
Stephen D. Roberts

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

ABSTRACT

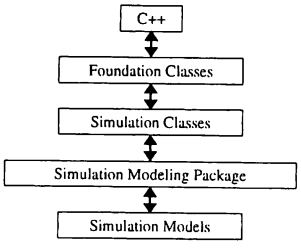
This tutorial shows how to design object-oriented simulation models using C++. A set of object classes have been written in C++ which can be used to create simulation models and simulation packages. The simulations built with these tools possess the benefits of an object-oriented design, including the use of classes, inheritance, encapsulation, polymorphism, run-time binding, and parameterized typing. These concepts are illustrated by creating a network queuing simulation language which has several notable features not available in other similar languages. 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

In previous WSC tutorial papers (see Joines, Powell, and Roberts 1992, 1993), we described the benefits of an object-oriented simulation through a network queuing simulation language. This paper will focus on the fundamental class structure (Section 2) and the design of a complete simulation system (Section 3). The network simulation language, YANSL, is then described as an illustration within the context of this design (Section 4). Everything is implemented in C++, which impacts the implementation of the object-oriented simulation package. C++ is an object-oriented extension to the C programming language (Lippman 1991).

1.1 The Object-Oriented Context

The general conceptual design of the object-oriented context for simulation, as conceived in this approach, is illustrated in the figure below:



The convention (which will be explained latter) is to refer to the top of this figure as "lower level" and to the bottem of this figure as a "higher level." Thus the lowest level of the design is the C++ general programming lanugage. Specific simulation models are at the highest level. The common notion of a simulation language falls somewhere in the middle of this design.

"Users" may "relate" to this design at any of the design levels. Persons interested only in the results may simply execute the models, while very knowledgeable persons may employ "raw" C++. The concepts at each level are "encapsulated" so that simulation model users, for example, need not be concerned about the concepts at a lower level. The more sophisticated user, however, can delve deeper into the design, eventually reaching the C++ level. Implicit in this design is a "hierarchy" of information, ranging from specific behavior of specific models to general program behavior.

In comparison, general computer users vary in their "use" of computer software. Many computer users simply execute software applications, like a spreadsheet or word processor. Some users may write computer programs that perform important needed tasks, which others may execute. More knowlegeable persons may write fundamental software like compilers that other programmers use. Sometimes those who write software for others are called "system software programmers" while those who write programs to solve specific problems are called "application programmers."

One perspective on the contribution of object-oriented simulation is that full "simulation software engineering" is now being added to "simulation applications engineering." This addition will provide simulation users with not only a full array of simulation tools, but also the means to add new tools.

1.2 OOS Appeal to Simulation Application Users

The idea of an "object-oriented" simulation has great intuitive appeal in applications, 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, as well as the work plan could be viewed as objects.

It is also quite easy to describe existing simulation languages using object terminology (see Joines, Powell, and Roberts 1992, 1993). A simulation language/package provides a user with a set of pre-defined object classes from which the simulation modeler can create needed objects. The modeler creates objects and specifies their behavior through the parameters available. The integration of all the objects into a single package provides the complete simulation model.

1.2.1 Problems with Extensibility

Because many simulation 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, only the vendor can make modifications to the internal functionality.

Also, users have only 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.

None of this is fully satisfactory because, at best, any procedure written cannot use and change the behavior of a pre-existing object class. Also, any new object classes defined by a user in general programming language do not coexist directly with vendor code. The Arena software (Collins and Watson 1993) provides some upward extensibility by a template approach to representing blocks of Siman statements (which may include the graphical representation). SLX (Henriksen 1993) will provide both higher and lower extensiblity.

1.2.2 An Object-Oriented Simulation Approach

Object-oriented simulation deals directly with the limitation of extensibility by permitting full 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 as a more complex resource.

2 CLASSES in C++

The class concept is fundamental to object-oriented software. The class provides a "pattern" for creating objects and defines the "type." An example (as it appears in C++) is the following `Exponential` class, which is used to obtain exponential random variates:

```
#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;
};
```

The class definition determines the object's properties.

2.1 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's properties.

2.2 Inheritance

The `Exponential` class was not defined "from scratch." For instance, it doesn't say anything about its use or origin of random numbers. 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.

2.3 Construction and Initialization of Objects

When a class object is needed, the creation and initialization of it is provided by a function called a "constructor." 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.

2.4 Run-time Binding

The `sample()` function is specified as a virtual function in `Exponential` because we don't know what variate will be sampled at run-time. Because the sample function is virtual, at run-time the program will 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. Without run-time binding, the programmer must explicity specify which function to call, causing very complex code to be carefully written.

2.5 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. Under other circumstances, polymorphism allows users to produce the same behavior with different objects. For instance, the act of obtaining a material handling device need not depend on which device is sought.

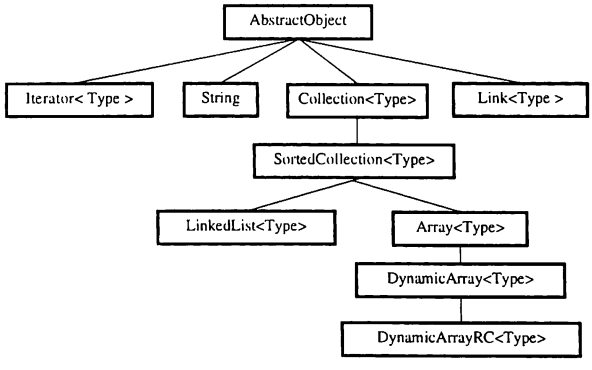
3 THE SIMULATION CLASS HIERARCHY

A key to the creation of a fully integrated simulation package is the use of a *class inheritance hierarchy* (introduced in Section 2.2). With C++ being the most abstract form (lowest level) of a simulation package, more concrete elements are added so that, at the highest level, the final product may be a specific simulation model. A specific simulation is also a kind of simulation model, which is a kind of simulation, which is a kind of programing project, which is a kind of C++ program.

An inheritance heirachy can be viewed as a tree. The base of the tree is the most abstract class and the leaves present the most specific class. Thus, the convention is that the "base" is known as the lower level, whereas the leaves are considered the higher level..

3.1 Foundation Classes

The hierarchy for the foundation classes of the simulation package are given in the following tree:



Classes in the prior figure provide a variety of general support which are useful in building simulation languages and simulation packages. While not specific to simulation, they provide a foundation framework from which more simulation-specific classes may be created.

3.1.1 AbstractObject

The `AbstractObject` forms the *fundamental base class* for the entire design and all other classes are

derived from this base class. The AbstractObject class defines and characterizes all the essential properties every class in this design should possess. No instances or objects of AbstractObject can be created since its primary purpose is to insure that all classes have the same basic form. Such a common form gives uniform character to the design and allows all classes to share common properties.

The AbstractObject provides for the following general common properties:

1. Testing and Identifying Objects: by getting the class name, class identification numer, the number objects of this class and testing if this object is a member of a particular class, a kind of class (may be a parent class of this object's class), and associated with a class based on some arbitrary association function.
2. Comparing Objects: by checking to see if the current object is the same object (object is itself), or is an exact copy of this object.
3. Printing, Tracing, and Reading Objects.
4. Copying Objects: by making shallow or deep copies.
5. Object Operators: by providing stream insertion and extraction, equality (but not from the same class) and inequality.

Any class derived from the AbstractObject will need to either inherit these properties or provide them within the class.

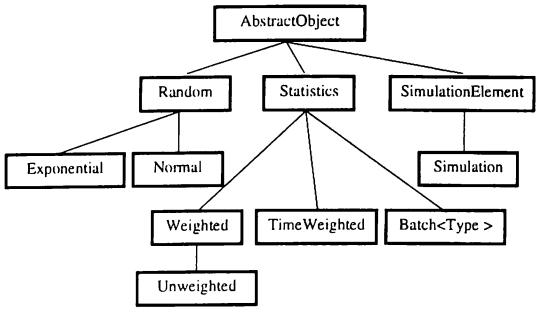
3.1.2 Foundation Support Classes

The foundation support classes provide useful classes for the general manipulation of objects found important in creating simulation languages/packages. These include the classes for string, arrays, and linked lists. Arrays may be dynamically dimensioned and may have their index range automatically checked.

These classes augment the C++ language with some widely employed object classes generally associated with containers. Similar classes are now widely available in object libraries such as those in the NIH Class Library (Gorlen, et al. 1990), the Borland object library(Borland 1993), and others. These libraries make quick work of many other elements that may be needed to build a simulation language or package.

3.2 Simulation Classes

The simulation classes provide basic simulation functionalty including random number and random variate generation, statistics collection, and base simulation elements. The simulation class hierarchy is shown below:



As can be seen, all the clsses are derived from AbstractObject in order to maintain the commonality.

3.2.1 Random Numbers and Random Variates

Random number generation is obtained from the Random class. Random variate generators are derived from the Random class so that each source of variate generation has its own random number generator(or generators). This design has two benefits. One, it facilitates the use of inverse transform method of random variate generation, and two, by associating each variate generator with its own random number stream, variance reduction through correlated sampling is possible. Random number and random variate generation properties include: (1) setting and getting generator parameters, (2) obtaining random numbers/variates, and (3) creating antitthetic sampling.

3.2.2 Statistics Collection

Basic statistics can be collected on Weighted, Unweighted, and TimeWeighted variables. Also, statistics may be "batched" from any of the basic statistic types. Tables, plots, and histograms may be displayed for basic or batched statistics.

Statistics collection properties include: (1) stopping and starting statistics collection, (2) clearing the statistics, and (3) reporting statistics. Basic statistics are collected during the simulation and provide: (1) observation base of (weighted) observations or time, (2) mean and standard deviation, and (3) minimum and maximum observations. Batched statistics are also collected during the simulation and provide both over-batch and current batch results. Batchs can be based on time intervals or numbers of observations.

3.2.3 Simulation Component Classes

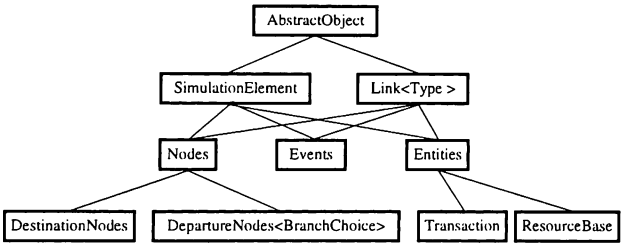
SimulationElement contains the simulation time and manages the event calendar. It provides for event and time management by being capable of: (1) scheduling events, (2) getting the next event, and (3)

getting and setting the current time. This class provides an important base class from which modeling classes are derived.

The `Simulation` class has the run control properties which manage the complete simulation and include: (1) getting the current replication number, (2) setting the number of replications, (3) setting the length of the run, (4) stopping the simulation or current replication, and (5) printing summary and individual output reports.

3.3 Simulation Modeling Classes

To aid in the construction of network queueing languages and packages, several simulation modeling classes have been implemented. The components are events, nodes, transactions, and resources. These components are derived from both the `SimulationElement` and the `Link` classes:



`Events` contain the properties related to event management and provide: (1) the means for setting and getting the event time, (2) setting and getting other event information, and (3) processing the event. `Nodes` contain the properties which include: (1) getting and setting the node count, (2) getting node identification number, (3) obtaining the node type, (4) accessing a list of all nodes in the network, and (4) finding the entities at the current node. Derived from nodes are the `Destination` and `Departure` nodes. A destination node can be entered while a departure node may be exited. Departure nodes have branches connected to them and therefore need a "BranchingChoice." The properties of the destination node include the entering process while the departure nodes provides the exiting process, the branching method, and related branching functions.

`Entities` contain general properties of active elements of the network simulation. These properities include: (1) getting and setting the entities creation time, (2) obtaining its status, (3) getting its current location, and (4) obtaining the status entry time. `Transaction` and `ResourceBase` classes are derived from `Entities`. The `Transaction` class provides entities that may demand service and have properties for: (1) acessing the list of captured resources and (2) getting and setting the node entry time. The `ResourceBase` class provides entities that can provide service and have properties for: (1) getting and setting the resource name, (2) getting resource states, and (3) defining the resource states.

4 CREATING A SIMULATION PACKAGE

Special simulation languages and packages may be created from these object classes. In this section we present the YANSL network queuing simulation language that has been presented in the prior WSC papers (Joines, Powell, and Roberts 1992, 1993). YANSL is an acronym for "Yet Another Network Simulation Language."

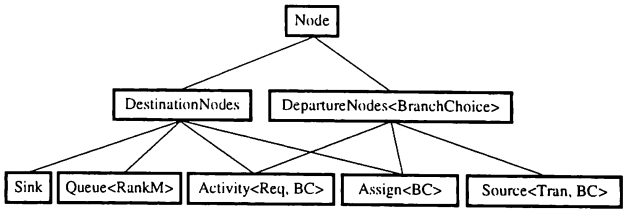
4.1 Basic Concepts and Objects in YANSL

YANSL was developed to illustrate the importance of object-oriented simulation. YANSL is 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 it is a very powerful alternative.

4.1.1 Classes Specific to YANSL

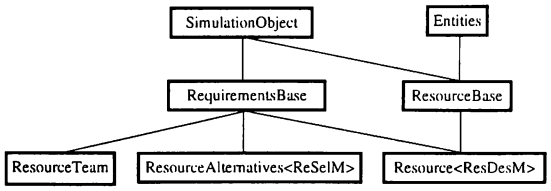
Several classes have been dervied from the general simulation classes to create the YANSL modeling package. The general simulation support classes, such as the variate generation, statistics collection, and time managment, are used directly. The node and resourse classes are further enhanced and a "choices" class is introduced.

The node hierarcy for the YANSL nodes is:



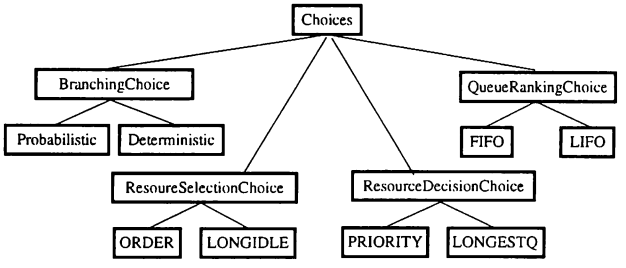
Sink and queue nodes can have transactions branched to them and are therefore destination nodes. An activity node is both a departure and a destination node, so it inherits from both the departure and destination node classes. This inheritance from multiple parents is called "multiple inheritance." The properties of the YANSL nodes allow transactions to be created at source nodes, wait at queue nodes, receive attribute assignment at assign nodes, be delayed at activity nodes, and exit the network at sink nodes. Resources may service transactions at activity nodes.

The resource heirarchy enhanced for YANSL is:



The resource classes allow resources to be identifed as individuals, as member of alternative groupings at an activity, or as members of teams. When there is a choice of resource service at an activity, then a resource selection method is employed. The ability to request a resource service at run-time without specifying it explicitly is another example of polymorphism.

The `Choices` available in YANSL are shown in the following class heirarchy:



The choices available add wide flexiblity to the decision-making functions in the simulation without having different classes for each different function. Instead, classes are parameterized with these choice classes. The choices consist of the following methods. They allow for the selection of alternative branches from a departure node, selection among alternative requirments at an `Activity`, as well as provide the decision making ability for resources to choose what to do next, and ranking choices among transactions at an `Queue`. The choices are used to represent the time-dependent and changing decisions that need to be modeled.

4.1.2 Modeling with 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. Transactions flow through the network. and have exactly the same interpretation it has in the other simulation languages. 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).

4.2 The TV Inspection and Repair Problem

As a portion of their production process, TV sets are sent to a final inspection station (refer to Joines, Powell, and Roberts, 1992 and to the harbor problem in Joines, Powell, and Roberts, 1993). Some TVs fail inspection and are sent for repair. After repair, the TVs are returned for re-inspection. Transactions are used to represent the TVs. The resources needed are the inspector and the repairman. The network is composed of a source node which describes how the TVs arrive, a queue for possible wait at the inspect activity, the inspect activity and its requirement for the inspector, a sink where good TVs leave, a queue for possible wait at the repair activity, and the repair activity. Transactions branch from the source to the inspect queue, are served at the inspect activity, branch to either the sink or to the repair queue, are served at the repair activity and return to the inspect queue. The data used in the simulation is that the interarrival time of TVs is exponentially distributed with a mean interarrival time of 5.0 minutes, the service time is exponentially distributed with a mean of 3.5 minutes, the probability a TV is good after being inspected is .85, and a repair time that is exponentially distributed with a mean of 8.0 minutes.

4.3 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 help offered to the user. Whatever the modeling system used, the ultimate computer readable representation of the model would appear as follows:

```
#include "simulation.h"

main()
{
// SIMULATION INFORMATION
Simulation      tvSimulation( 1 );
                // One replication

// DISTRIBUTIONS
Exponential     interarrival( 5 ),
                inspectTime( 3.5 ),
                repairTime( 8.0 );

// RESOURCES
Resource< PRIORITY >  inspector, repairman;
```

```
// NETWORK NODES

   /** Transactions Arrive **/
Source< Transaction, DETERMINISTIC >
        tvSource( interarrival, 0.0, 480 );
     // Begin at 0.0 and quit at 480.0

   /** Inspection **/
Queue< FIFO >  inspectQueue;
     inspector.addQueue( inspectQueue );
Activity<PROBABILTIY>   inspect( inspectTime );
     inspect.addRequirement( inspector );
     inspectQueue.addActivity( inspection );

   /** Repair **/
Queue< FIFO >          repairQueue;
     repairman.addQueue( repairQueue );
Activity<DETERMINISTIC>   repair( repairTime );
     repair.addRequirement( repair );
     repairQueue.addActivity( repair );

   /** Transactions Leave **/
Sink finish;

//NETWORK BRANCHES
tvSource.addBranch( inspectQueue );
inspect.addBranch( finish, .85 );
                // 85% are good and leave
inspect.addBranch( repairQueue, .15 );
                // 15% need repair
repair.addBranch( inspectQueue );


//RUN the Simulation
tvSimulation.run();
}
```

The previous model has all 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. The statements are highly readable and follow a 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. 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.

4.4 The Objects and their Specification

Lets take a closer look at the YANSL "statements." The model is enclosed in a recognizable C/C++ framework, namely having a `#include` statement that includes all 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.

There are 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).

4.4.1 Object Declarations

The objects in YANSL are declared in a form consistent with 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 ),
                inspectTime( 3.5 ),
                repairTime( 8.0 );
```

creates three exponential distributions whose names are `interarrival`, `inspectTime`, and `repairTime` and whose initialization parameters are given in parenthesis. It is important to note that the mean interarrival time is specified as an integer 5, but in fact it is assumed to be a floating point 5.0. 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." 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. A parameterized type is used when the object class needs some information. This should not be confused with initialization of objects where the object needs some information. As an example, consider

```
Activity< PROBABILITY > inspect( inspectTime );
```

where the `Activity` class needs some branching choice class called `PROBABILITY`, while the object `inspect` is initialized with a reference to the `inspectTime` object. Notice that a class will be parameterized with another class, 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. 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" object.

4.4.2 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,

```
inspector.addQueue( inspectQueue );
```

the message `addQueue` with `inspectQueue` object as a parameter is sent to the `inspector` object. In C++ terminology the `addQueue` function in the `inspector` object is passed the `inspectQueue` object. The purpose of this message/function is for the inspector to know that it is to service the queue of the inspection 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 `tvSimulation.run` causes the simulation execution to begin.

4.5 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++ 4.0 (Borland 1993). 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.

4.6 Embellishments

There is no distinction between the base features of YANSL and its extensions, illustrating the "seamless" nature of user additions. Many more embellishments are simply parallel application of the approaches used in the prior sections. For example, the embellishments shown in the earlier papers (Joines, Powell, and Roberts 1992, 1993) could be applied here. 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 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 1990), 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. 1993. *Borland C++ version 4.0.* Borland International, Inc. 100 Borland Way, Scotts Valley, CA 95066-3249.

Collins, N. and C. M. Watson. 1993. Introduction to Arena. In *Proceedings of the 1993 Winter Simulation Conference*, ed., G. W. Evans, M. Mollagasemi, E. C. Russell, and W. E. Biles, 205-212. Institute of Electrical and Electronics Engineers, Los Angeles, CA..

Ellis, M, and B Stroustrup. 1990. *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.

Gorlen, K. E., Orlow, S. M., and P. S. Plexico. 1990. *Data Abstraction and Object-Oriented Programming in C++.* New York: John Wiley & Sons.

Henriksen, J. O. 1993. SLX, the successor to GPSS/H. In *Proceedings of the 1993 Winter Simulation Conference*, ed., G. W. Evans, M. Mollagasemi, E. C. Russell, and W. E. Biles, 263-268. Institute of Electrical and Electronics Engineers, Los Angeles, CA..

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.

Joines, J. A., K. A. Powell, Jr., and S. D. Roberts. 1993 Building object-oriented simulations with C++. In *Proceedings of the 1993 Winter Simulation Conference*, ed., G. W. Evans, M. Mollagasemi, E. C. Russell, and W. E. Biles, 79-88. Institute of Electrical and Electronics Engineers, Los Angeles, CA..

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

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.

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.

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, IIE, and IEEE. His interests include object-oriented simulation, artificial neural networks, and genetic algorithms as applied to manufacturing.

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 has been 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 past Chair of the Board of Directors of WSC.