Proceedings of the 1996 Winter Simulation Conference
ed. J. M. Charnes, D. J. Morrice, D. T. Brunner, and J. J. Swain

DESIGN OF OBJECT-ORIENTED SIMULATIONS IN C++

Jeffrey A. Joines
Stephen D. Roberts

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

ABSTRACT

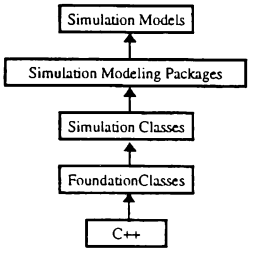
An object-oriented simulation (OOS) consisting of a set of object classes written in C++ can be used to create simulation models and packages. The simulations built with these tools possess the benefits of an object-oriented design, including the use of encapsulation, inheritance, polymorphism, run-time binding, and parameterized typing. These concepts are illustrated by creating a set of object frames which encapsulate simulation requirements. Simulation modeling is contained within a set of modeling frameworks. A network queuing simulation language is developed which has several notable features not available in other non-OOS languages. OOS provides full accessibility to the base language, faster executions, portable models and executables, a multi-vendor programming language, and a growing variety of complementary development tools.

1 INTRODUCTION

Previous WSC tutorial papers (see Joines, Powell, and Roberts 1992, 1993, 1994, 1995) described the benefits of object-oriented simulation (OOS). This paper will focus on the fundamental class structure (Section 2) and the design of the simulation system (Section 3). The network simulation language, YANSL, is presented as an illustration within the context of this design (Section 4). Everything is implemented in C++, which impacts the implementation. C++ is an object-oriented extension to the C programming language (Ellis and Stroustrup 1990). The source code in this paper is available from anonymous ftp: `ftp.eos.ncsu.edu/pub/simul`.

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 following figure.



The lowest level construct is the C++ general programming language. 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 a simulation model user, for example, needs not be concerned about the concepts at a lower level. The more interested user, however, can delve deeper into the design, ultimately reaching the C++ level. Implicit in this design is a "hierarchy" of information, ranging from the behavior of specific models to general program behavior.

One perspective on the contribution of object-oriented simulation is that "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 or modify existing 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 easy to describe existing simulation languages using object terminology (see Joines, Powell, and Roberts 1992). A simulation language provides a user with a set of pre-defined object classes from which the simulation modeler can create needed objects. The modeler declares objects and specifies their behavior through the parameters available. The integration of all the objects into a single bundle provides the 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 modify 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 co-exist directly with vendor code.

The Arena software (Hammann and Markovitch 1995) provides some upward extensibility by a template approach to representing blocks of Siman statements (which may include the graphical representation). SLX (Henriksen 1995) will provide both higher and lower extensibility. Full object-oriented simulation systems include Smalltalk (Goldberg and Robson 1989), Modsim III (Belanger and Mullarney 1990), and Simple++ (Geuder 1995). C++ based simulation packages include C++Sim (Little and McCue 1994), C++/CSIM17 (Schwetman 1995) and Simpack (Fishwick 1995).

1.2.2 An Object-Oriented Simulation Approach

OOS 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 a new data type is added, it can assume just as important a role as any implicit data types. For example, 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"
/*Class Exponential describes inverse transfor-
mation generator for exponential variates. */

class Exponential: public Random {
 private:
   double mu;
 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; }
};
```

2.1 Class Properties

The class definition specifies the object's properties, namely the data and functions, and are generally grouped into "public" and "private" sections (C++ also permits another grouping called "protected"). When the object is created, the public properties can be accessed from outside the object. The private properties are information kept strictly locked within the object and are available only to object's functions. For example, the object `mu` (exponential mean) is declared as a private data member of type double and cannot be directly accessed. However, a public function called `getMu()` does return the value of `mu`, while `setMu()` allows the user to change the value of `mu`. Making a property private restricts use outside the class and encapsulates the object's properties.

2.2 Inheritance

The `Exponential` class was not defined "from scratch." It doesn't say anything explicitly about how it obtains 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 it 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, inheritance makes the `Exponential` class a "kind of" `Random` class. In object-oriented terminology this is considered an "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, but an act of composition.

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 both the `Random` (not shown) and `Exponential` class. Therefore, sampling can be specified as a `Random` object and at run-time, the program will decide from which random variate to sample. This approach of tying 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. With run-time binding, new variate types can be added through inheritance without altering existing simulation code.

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. Under other circumstances, polymorphism allows users to produce the same behavior with different object types. For instance, requesting a material handling does not depend on the the device being sought. For example, one message "request" can be used for AGV or Trucks (e.g. request("AGV") or request("Truck"))rather than a message for each type (e.g., requstAGV and requestTruck).

3 CLASS HIERARCHY AND FRAMES FOR OOS

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, the more concrete elements are added so that, at the highest level, the final product may be a specific simulation model. A specific simulation model is also a kind of simulation model, which is a kind of simulation, which is a kind of C++ program.

An inheritance hierarchy can be viewed as a tree. The base of the tree is the most abstract class and the leaves present the most specific class. In order to collect classes into levels of abstraction, we introduce object-based "frames." A *frame* is a set of classes that provide a level of abstraction in the simulation and modeling platform. A frame is a convenient means for describing various "levels" within the simulation class hierarchy and is a conceptual term (not formalized within C++).

3.1 Foundation Frame

The foundation classes provide a base structure from which more simulation-specific classes may be created. These foundation classes are not simulation specific.

3.1.1 Abstract Objects

The `AbstractObject` forms the *fundamental base class* for the entire design and all other classes are derived from this base class. The `AbstractObject` defines all the essential properties and gives uniform character to the design. It provides for the following common properties and is called a "nice" class (see Carroll and Ellis 1995): a *default constructor* which construct objects without user-specified parameters; a *copy constructor* which establishes a mechanism for creating a new object as a copy of another within this class; an *assignment operator* which allows objects to be the target of an assignment using this operator; an *equality operator* which tests the "equality" of one object with another using this operator; a *less than operator* which test whether one object is "less than" another using this operator; and a *destructor* which provides for the orderly destruction of an object.

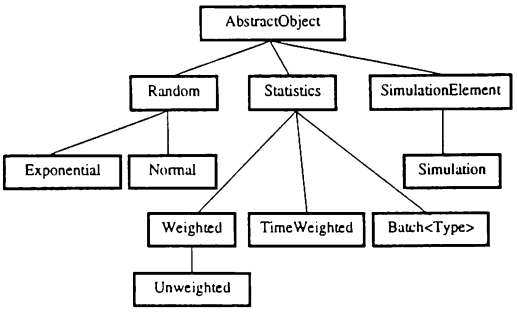
3.1.2 Foundation Support Classes

The foundation support classes provide general manipulation of objects important in the creation of simulation languages/packages including classes for strings, arrays, and linked lists. These are now available in the new standard template library(STL) (Plauger 1995).

3.2 Simulation Frame

This frame provides basic simulation functionality including random number and random variate generation,

statistics collection, and base simulation elements as seen below.



As can be seen, all the classes are derived from `AbstractObject` in order to maintain a common class design throughout any simulation project.

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 by facilitating the use of inverse transform method of random variate generation and by associating each variate generator with its own random number stream, variance reduction through correlated sampling is possible. Random number and variate generation properties include: (1) setting/getting generator parameters, (2) obtaining random numbers/variates, and (3) creating antithetic 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. Batches can be based on time intervals or numbers of observations.

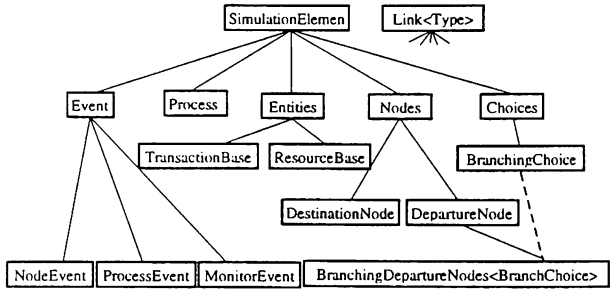
3.2.3 Simulation Component Classes

`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 Frame

To aid in the construction of simulation languages and packages, several simulation modeling classes have been designed and implemented. The components of the modeling frame are events, entities, processes, nodes, and choices. These components are derived from both the `SimulationElement` and the `Link` classes. In the following figure and subsequent figures, the solid lines are inherited relationships ("is-a") while the dashed lines are composition relationships ("has-a").



`Events` contain the properties related to simulation 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. `Node`, `Process`, and `Monitor` events provide specialized properties that are needed when events occur within a node or a process, or are independently specified.

`Entities` provide active elements for the simulation, whether permanent or temporary. The properties of entities include: (1) getting the entity's creation time, (2) obtaining its status, (3) getting its current location, (4) obtaining the entry time of the entity's current state, and (5) getting the entity's time in the system. `TransactionBase` and `ResourceBase` classes are derived from `Entities` and extend the entities for use in general networks. The `TransactionBase` class provides entities that may need service and has properties for: (1) getting/setting the node entry time, (2) getting the creation node, and (3) getting/setting the identification number. The `ResourceBase` class provides entities that can provide service and has properties for: (1) getting/setting the resource name, (2) getting/setting resource states, and (3) defining the resource states.

`Processes` provide an encapsulated means for describing simulation processes (not computer tasks) such as seizing and releasing resources and reneging at queues. The process class is generally used to provide a means of decomposing a complex simulation activity, like preempting a resource, and is a form of a "helper" class for the simulation.

`Nodes` are used for network modeling and contain the properties which include: (1) getting/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 (5) finding the entities at the current node. Derived from `Node` are the `Destination` and `Departure` nodes. A destination node can be entered while a departure node may be exited. Often departure nodes have branches connected to them and therefore need a "BranchingChoice" and are called `Branching-DepatureNodes` The properties of the destination node include the "entering process" while the departure nodes provides the "exiting process", the branching choice, and related branching specifications.

`Choices` are used to give the simulation model "intelligence." Routes, rules, and policies may be modeled through the various choices.

3.3.1 Frames and Frameworks

While frames provide a convenient means to describe the levels of abstraction within the entire object-oriented simulation platform, another means of encapsulation is needed to deal with the broad simulation modeling concepts and features contained within the design. In a sense, the frames are quite similar to class libraries which can be called upon in the development of an actual simulation modeling language or package. However, for the higher level modeling classes, these library-like collections of classes are too complexly interrelated to be represented simply as a single level of abstraction. A better approach to the description of these higher level complex interactions is the notion of "frameworks." For our purposes, *frameworks* are used to describe those collections of classes that provide a set of specific modeling facilities. The frameworks may consist of one or more class hierarchies. These collections make the use and reuse of simulation modeling features more intuitive and provide for greater extensibility.

There are several frameworks that compose the modeling frame and include: (1) ***Transaction Framework*** which establishes the basic properties of the transaction and provides a means to create transactions, to bring them into the network, to branch them from node to node, to cause them to exit the network, and to destroy them; (2) ***Resource Framework*** which establishes the basic properties of resources, provides for resource teams and resource groups, provides for preemption of resources, seize resources, and release resources; (3) ***Queuing Framework*** which stores transactions awaiting resources, ranks transactions in the queue, provides conditional and unconditional reneging from queues, and gates transactions in queues until conditions are appropriate for their future movement; and (4) ***Activity Framework*** which delays transactions for some specified time, determines among resource alternatives and requires resources, and abort transactions from the activity.

4 CREATING A SPECIFIC OOS

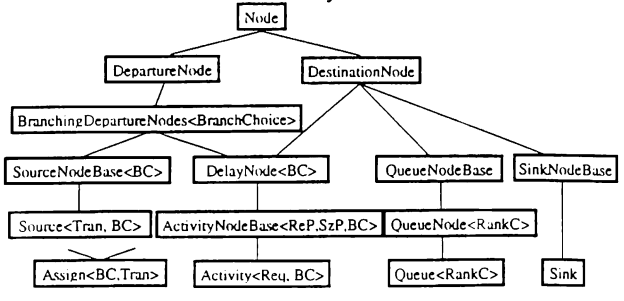
Special simulation languages and packages may be created from these object classes. In this section we present the YANSL network queuing simulation language that has been presented in prior WSC papers. YANSL is an acronym for "Yet Another Network Simulation Language." YANSL is just one *instance* of the kind of simulation capability that can be developed within an object-oriented simulation environment.

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 1995), SIMAN (Pegden, Shannon, and Sadowski 1995), 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.

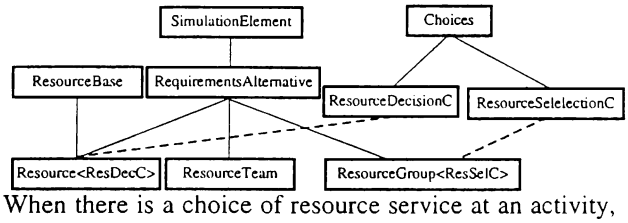
4.1.1 Classes Specific to YANSL

Several classes are chosen from the modeling frameworks to create the YANSL modeling package. These classes are collected together to form a "simple" modeling/simulation language which can be extended to create more complicated features. The general simulation support classes, such as variate generation, statistics collection, and time management, are used indirectly throughout the modeling frameworks. The network concepts are somewhat enhanced, but are taken from the modeling framework. The node hierarchy for YANSL is:



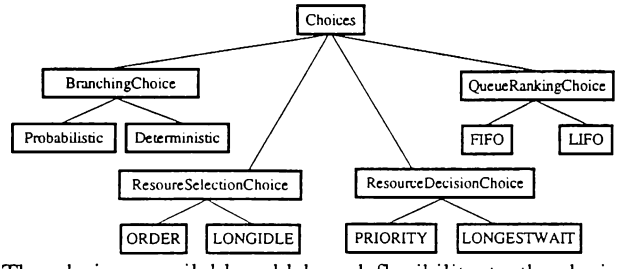
The higher level nodes (`Assign`, `Activity`, `Queue`, `Source`, and `Sink`) are used directly by the YANSL modeler. Lower level nodes provide abstractions which are less specific thus allowing specialization for other simulation constructs (e.g., the QueueNodeBase class excludes ranking and statistics). Sink and queue nodes can have transactions branched to them and are therefore destination nodes, while the source node is a departure node. The delay and assign nodes are both departure and destination nodes, so they inherit from both the departure and destination node classes. This inheritance from multiple parents is called "multiple inheritance." An activity is a "kind of" delay but includes resource requirements. 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 framework for YANSL allows resources to be identified as individuals, as member of alternative groupings, or as members of teams.



When there is a choice of resource service at an activity, then a resource selection method is used. The ability to request a resource service at run-time without specifying it explicitly is another example of polymorphism.

`Choices` available in YANSL extend those in the modeling frameworks (see the following class hierarchy):



The choices available add broad flexibility to the decision-making functions in the simulation, without needing different classes for each different function. Instead, classes are parameterized with these choice classes and the choices consist of several methods. Specifically in YANSL, they allow for the selection of alternative branches from a departure node, selection among alternative resources in requirements 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 the same interpretation they have 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.

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, the inspect service time, and repair service time are exponentially distributed with a mean of 5, 3.5, and 8 minutes respectively, and the probability that a TV is good after being inspected is .85.

4.3 A 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); //1 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("Inspect Queue");
     inspector.addQueue( inspectQueue );
Activity<RequirementSet,PROBABILITY>
                        inspect( inspectTime );
     inspect.addRequirement( inspector );
     inspectQueue.addActivity( inspection );

   /** Repair **/
Queue< FIFO >  repairQueue("Repair Queue");
     repairman.addQueue( repairQueue );
Activity<RequirementSet,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 an almost one-to-one correspondence to the problem entities. The statements are highly readable and follow a simple format. The pre-defined object classes give 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

The YANSL "statement" model is enclosed in the 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 since even this elements could be camouflaged using the C preprocessor.

There are two types of YANSL "statements." The first is the declaration of objects in the model. These statements describe the elements in the simulation. The second type of statement is member function calls or messages 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.

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. 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" and provide "has-a" relationships. 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. As an example, consider

```
Queue< FIFO > inspectQueue( "Inspect Queue" );
```

where the `Queue` class needs some ranking choice class called `FIFO`, while the object `inspectQueue` is initialized with a string, the name of the queue. Notice that a class will be parameterized with another class, while an object is parameterized with another object.

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++ 5.0 (Borland 1995). C++ is strongly typed, so error checking is very good. Also, the simulation is easily linked into other C++ libraries.

4.6 Embellishments

The lack of distinction between the base features of YANSL and any extensions illustrate the "seamless" nature of user additions. Many embellishments are possible. For example, the embellishments shown in the earlier papers (Joines and Roberts 1992, 1993) could be applied here. Such 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.

5 CONCLUSIONS

Modeling and simulation in an object-oriented language possesses many advantages. As shown, 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 O-O approach provides a consistent means of handling these problems.

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 ANSI standard, all C++ compilers are expected to accept the same C++ language. 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

Borland. 1995. *Borland C++ version 5.0*. Borland International, Inc. 100 Borland Way, Scotts Valley, CA.

CACI. 1995. *A quick look at Modsim III*. CACI Products Company, La Jolla, CA.

Ellis, M, and B Stroustrup. 1990. *The annotated C++ reference manual*. Reading, Massachusetts: Addison-Wesley.

Fishwick, P. A. 1995. *Simulation Model Design and Execution*, Englewood Cliffs, N.J. Prentice-Hall.

Geuder, D.. 1995. Object-Oriented Simulation Modeling with Simple++ In *Proceedings of the 1995 Winter Simulation Conference*, ed., C. Alexopoulos, K. Kang, W. R. Lilegdon, and D. Goldsman, 519-523. Institute of Electrical and Electronics Engineers, Washington, D.C.

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

Hammann, J. E. and N. A. Markovitch. 1995. Introduction to Arena. In *Proceedings of the 1995 Winter Simulation Conference*, op cit.

Henriksen, J. O. 1995. Introduction to SLX. In *Proceedings of the 1995 Winter Simulation Conference*. 502-509, op.cit.

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, 154-162. 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, 205-212. Institute of Electrical and Electronics Engineers, Los Angeles, CA.

Joines, J.A. and S. D. Roberts. 1994. Design of Object-Oriented Simulations in C++. In *Proceedings of the 1994 Winter Simulation Conference*, ed., Jeffrey Tew, S. Manivannan, Deborah Sadowski, and Andrew Seila, 157-165. Institute of Electrical and Electronics Engineers, Orlando FL.

Joines, J.A. and S. D. Roberts. 1995. Design of Object-Oriented Simulations in C++. In *Proceedings of the 1995 Winter Simulation Conference*, 82-92, op.cit.

Little, M.C. and McCue, D.L., 1994, "Construction and Use of a Simulation Package in C++,"*C User's Journal*, 12(3).

Pegden, C. D., R. E. Shannon, and R. P. Sadowski. 1995. *Introduction to simulation using SIMAN*, Second Edition, New York: McGraw-Hill.

Plauger, P. 1995. *The draft C++ library*, Englewood Cliffs, N.J. Prentice-Hall.

Pritsker, A. A. B. 1995. *Introduction to simulation and SLAM II*, Fourth 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.

Schwetman, H. 1995. Object-Oriented Simulation Modeling with C++/CSIM17. In *Proceedings of the 1995 Winter Simulation Conference*, 529-533, op.cit.

AUTHOR BIOGRAPHIES

JEFFERY A. JOINES is a Ph.D Candidate in the Department of Industrial Engineering and a Research Associate in Furniture Manufacturing and Management Center 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 INFORMS, IIE, and IEEE. His interests include object-oriented simulation, cellular manufacturing, and genetic algorithms.

STEPHEN D. ROBERTS is Professor and Head of the Department of Industrial Engineering at NCSU. He received his B.S.I.E., M.S.I.E., and Ph.D. from Purdue University. He was the recipient of the 1994 Distinguished Service Award. He has served as Proceedings Editor and Program Chair for the Winter Simulation Conference. He is an INFORMS/CS representative to and past Chair of the Board of Directors of WSC.