Proceedings of the 1997 Winter Simulation Conference
ed. S. Andradóttir, K. J. Healy, D. H. Withers, and B. L. Nelson

AN INTRODUCTION TO OBJECT-ORIENTED SIMULATION 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) consists of a set of objects that interact with each other over time. This paper provides a thorough introduction to OOS, addresses the important issue of composition versus inheritance, describes frames and frameworks for OOS, and presents an example of a network simulation language as an illustration of OOS.

1 INTRODUCTION TO OBJECTS

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 immediately 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

Previous WSC tutorial papers (see Joines and Roberts 1995) presented an object-oriented simulation through a network queuing simulation language while Joines and Roberts (1996) provided fundamental structures for the design of a complete simulation system. This paper will be more of tutorial and introduction to OOS. Everything within this paper is illustrated and implemented in C++ which is an object-oriented extension to the C programming language (Ellis and Stroustrup 1990). Our OOS is strongly influenced by the design of C++.

2 WHAT IS AN OBJECT?

It is very easy to describe existing simulation languages using object terminology. A simulation language provides a user with a set of pre-defined object classes (i.e., resources, activities, etc.) 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.

Therefore, an object can be described by an entity that holds both the descriptive attributes of the object as well as defines its behavior. For example, suppose you are modeling an exponential random variable in a simulation. The random variable may be described by a standard exponential statistical distribution which has a set of parameters (e.g., a mean in this case). This mean would be considered an attribute of the exponential random variable object. It maybe important to obtain observations from this random variable via sampling. One may want to obtain antithetic samples or to set the random seed. Sampling from the exponential random variable defines a particular behavior**.**

2.1 Encapsulation

The entity "encapsulates" the properties of the object because all of its properties are set within the definition of the object. In our example, the exponential random variable's properties are contained within the definition of the random variable so that any needs to understand or revise these properties are located in a single "place." Any users of this object need not be concerned with the internal "makeup" of the object. This also facilitates the ability to easily create multiple instances of the same object since each object contains all of its properties.

In C++, the keyword "class" is used to begin the definition of an object followed by the name of the object class and then the properties of the entity are defined within enclosing {}. For example, the following object defines the `Exponential` class.

```
class Exponential{
…// Properties of the Exponential
};
```

Without encapsulation, properties could be spread all over, making changes to the object very difficult.

2.1.1 Class Properties

The class definition specifies the object's properties, the attributes and behaviors. The attributes define all the singular properties of the object while the behaviors define how the object interacts within the environment with other objects.

Attributes are considered the data members of an object. In the case of our `Exponential` random variable, its mean (given by the identifier `mu`) would be a real number attribute.

```
double    mu;
```

Other attributes would be similarly defined.

The behaviors (sometimes referred to as methods) of an object represent actions the object can perform or take. For example, if the exponential random variable needed to obtain a sample, the following member function can be used:

```
double   sample(){
   return –mu * log( 1.0 – randomNumber() );}
```

where the `randomNumber`() function yields a uniform random variable between 0 and 1. By representing behavior with functions, the object can react to parameters passed in the function argument as well as change variable values within the function.

2.1.2 Classes and Instances

Notice, the word "class" not "object" is used in defining the object which can be confusing since it would seem that we are defining objects. Lets consider the more complete definition based on our prior discussion of encapsulation and properties (ignore the "public" for now), the `Exponential` class is defined as follows.

```
class Exponential{
   public:
      double mu;
      double sample(){ return –mu * log( 1.0 –
                        randomNumber() );}
};
```

Rather than defining an object directly, a class is defined where the class provides a "pattern" for creating objects and defines the "type." By defining a class (of objects), rather than a single object, the opportunity exists to use the class to create many objects (i.e., re-use existing code). Furthermore, as seen later, the class is a description of a pattern for constructing objects which can be easily extended.

Now that the class is defined, objects can be created directly from this class. These created objects are called "instances" of a class. For example, `serviceTime` is an instance of the `Exponential` class.

```
Exponential   interarrivalTime, serviceTime;
```

2.2 How Do Objects Communicate?

An OOS models the behavior of interacting objects over time. However before we can consider a simulation, we needs to understand how objects interact or communicate with each other. The interaction among objects is performed by communication called "message passing." One object sends a message to another and the receiving object then responds. An object may simply publish a message which may be responded to by one of several objects. For example in a bank simulation, a customer arrives at a bank and may be served by any of several tellers. In a O-O context, the customer publishes their arrival and waits for service by a teller. There are several ways in transmitting messages in an object-oriented program and it depends on the programming language.

2.2.1 Direct Reference

Perhaps the simplest form of message passing is direct reference to the object's attributes or data members. For example, if the `interarrivalTime` object needed to have a mean of 5.5, then the simplest means to communicate this message is through direct assignment.

```
interarrivalTime.mu = 5.5;
```

This message causes the object to receive the value and set its variable `mu`. This is a forced message because the object has no choice but to perform the action.

2.2.2 Data Methods or Functions

Rather than forcing a value upon an object, a value could be communicated to the object and then let it determine how to deal with the value. For example, if a new "member function" or data method to the Exponential class called `setMu()` was added as follows.

```
void setMu( double initMu ){
          mu = initMu;   }
```

Now the object is sent the setMu message with a message value of 5.5 which "communicates" our interest in changing the mean. The `interarrivalTime` object receives the message and changes its internal value of `mu`.

```
interarrivalTime.setMu(5.5);
```

Although this example really does the same thing as the direct reference, there are important distinctions. First, in our function call we simply "passed" the value of 5.5 to the object. Second, we didn't tell the object how to change the attribute `mu`. The object has a function written by the designer of the `Exponential` class that causes the mean parameter to change. Notice, the user of the function does not need to know how the function

inside the class works. In fact, the class designer could change the internal name of `mu` to `expMean` within the class, and all exiting user code would remain the same. This encapsulation of the data is extremely important in OOS. Also, the same message can be made to respond to several different message value types often referred to as "polymorphism."

2.2.3 Pointers

Another way to communicate is indirectly through pointers which are simply addresses of the location of an object. For example, a pointer to the `interarrivalTime` object can be created as follows and the setMu message can be sent via the pointer.

```
Exponential * rnPtr = &interarrivalTime;
rnPtr->setMu(5.5);
```

Pointers have the advantage of not needing to know the particular object ahead of time, but only the address of the object. Thus, if we change the pointer to point to the `serviceTime` object, the format of the message remains the same. With a more complex message, use of pointers becomes very convenient.

```
RNPtr->setMu(3.5);
```

2.3 How Are Objects Formed?

In our example, the exponential object has no ability to be created with different means. Instead, the object's mean was changed to a specific value. Although an object can be instantiated from a class without special instructions, often we want the creation to accomplish certain objectives. Likewise, we also might want to do something special when an object is destroyed.

2.3.1 Constructors and Destructors

Special member functions can be defined that act when an object is created and destroyed which are called constructors and destructors, respectively. The constructor is recognized by having no return type and having the same name as the class. For example, the following could be a constructor for the exponential object.

```
Exponential( double initialMu ) {
             mu = initialMu; }
```

This function accepts the invocation argument and sets the internal mean to it. An object whose initial mean is 4.3 can be specified upon creation as follows.

```
Exponential   serviceTime(4.3);
```

In C++, functions can be "overloaded" so that they differ only in their formal arguments (i.e., "polymorphism"). Therefore, a class can have multiple constructors. For example, if we wanted the exponential to accept an integer specification of its mean.

```
Exponential( int startMu ){
            mu = startMu;}
```

Now, exponential objects with either a double or an int as arguments can be specified (actually C++ will make appropriate conversions among its built-in types, but this example illustrates the way a user could provide conversions among user-defined classes). The following creates two objects using different argument types.

```
Exponential   arrival(9.3), inspect(6);
```

Users can also define a special member function called a destructor that acts when the object is destroyed. For example, a destructor for the exponential class has the following form.

```
~Exponential(){
        // print out how often used? }
```

Only one destructor can be defined since a destructor has no arguments.

2.3.2 Visibility of Properties

It should be clear that a user of a class does not really need to know the internal workings of the class. For example, they do not need to know what algorithm is used to obtain the sample (they may want to know for their own assurance). Furthermore, the designer of the class may not want the user of the class to know everything about the class. Thus, the class designer has the option of causing properties of the class to become invisible to users of the class and to provide a public interface to those hidden properties. The two most frequently used labels are "public" and "private." Properties within a class that are public can be accessed directly by a user while those that are private are available only to the designer. For example, the variable containing the mean is made private within the class to prevent improper use (i.e., direct manipulation). Our class would then look like the following.

```
class Exponential{
 private:
  double mu;
 public:
  Exponential(double initialMu ){mu=initialMu;}
  double sample();
  void setMu( double changeMu ){mu = changeMu;}
  double getMu( ) { return mu; }
};
```

Now `mu` cannot be changed directly by a user. Thus the direct reference to `mu`, as done earlier, will fail. Now communication to the exponential objects must be performed through member functions. The designer of the class can now protect the class data members from unwanted changes while the user of the class is unaffected.

2.4 How Are Objects Formed From Others?

One of the fundamental benefits of an O-O design is the ability to make other objects out of existing ones. We have already seen how to design a class of objects using the built-in types from C++. Suppose the following random number class has been defined which generates uniformly distributed numbers between 0 and 1.

```
class RandomNumber{
      long   seed;
   public
      RandomNumber( long seed = -1);
      void setSeed(long sd){seed=sd;}
      virtual double sample();
};
```

In this definition, the constructor argument can be specified or left blank to default to their initial values (i.e., -1 means use the next seed). The public member function `sample`() is used to obtain a sample and we will assume that the seed will be updated appropriately with each call to `sample`(). The "virtual" keyword will be discussed later.

There are two ways this random number generator could be used with our `Exponential` class. The first method is called **composition**, in which a random number object is included within the exponential class. The second method of using the random number generator is through **inheritance** which makes the exponential class a kind of random number. Inheritance is one of the major features that distinguish a "object-based" language from a true "object-oriented" one.

2.4.1 Composition

First, consider the case of composition where we simply compose the new class from the existing class:

```
class Exponential{
   private:
      double mu;
      RandomNumber rn;
   public:
      void setSeed(long sd){RN.setSeed(sd);}
… //Public Properties
};
```

Notice that the `Exponential` is defined simply to "have" a `RandomNumber`. In O-O parlance, the relationship between the `Exponential` and the `RandomNumber` rn is called a "has-a" relationship. The data member rn is used in the `sample`() function of the exponential. Notice, a `setSeed`() needs to be defined in order to access the one in the random variable.

2.4.2 Inheritance

The second kind of relationship among classes is called an "is-a" relationship and is based on inheritance or a parent-child relationship. In our example, the exponential can be considered a kind of random variable. It would be useful for the `Exponential` to be a child of `RandomNumber` and thus inherit all the random variable properties. Hence, what could be done to the random variable could also be done with the exponential. No additional `setSeed`() is required since the one in the random class can be used.

For example, sometimes a sample from an exponential is needed while other times a basic uniform generator is required. Suppose the following two objects and pointer are defined:

```
RandomNumber   uni;
Exponential   exp(5.5);
RandomNumber * pRN = &uni;
```

If at an activity in our simulation, a sample from a random variable is needed, the following message is sent to obtain an activity time.

```
pRN -> sample();
```

However, because Exponential is also a `RandomNumber`, the pointer `pRN` could be assigned to either an `Exponential` or a `RandomNumber` and the same message applies. In the **composition** example, two separate activities would be required (i.e., one which used an exponential and another one which used a uniform).

```
pRN = &exp;
```

In an true O-O language with inheritance, the message would be sent to the proper object and the sampling would be from the correct sampling function. In O-O terms, determining which variate to sample at run-time is called "run-time" binding and is performed by specifying the `sample`() to be "virtual" in the parent class.

To specify that `Exponential` inherits from `RandomNumber`, the header for the class definition would be modified as:

```
class Exponential: public RandomNumber{...
```

showing that `RandomNumber` is the parent and its visibility is "public".

Under inheritance, the child class inherits the public (and protected) properties of the parent. Now these properties are directly available to the child class and the class type resolves any conflicts. C++ also permits multiple inheritance, meaning a child can inherit from several parents.

3 OBJECT-ORIENTED VS. OBJECT-BASED

Because many simulation languages offer pre-specified functionality produced in another language, the user cannot access the internal function of the language. Instead, only the vendor can modify the internal function-

ality. 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.

3.1 Object-Based Extension

The object-based approach only allows extensibility in the form of composition (i.e., new objects can only be created out of existing objects). The simple `Event` object will demonstrate the limitations of extensibility only through composition. The Event object is used to move the simulation from one time to the next. Events are placed on the calendar and when an event is removed from the calendar the `processEvent`() function is called to handle the event. The following gives a portion of the `Event` class that can be used to process arrival of entities into the network and end of service events. Notice that depending on the type of event, the appropriate event handling function is called. This is an example use of composition.

```
class Event{
   private:
     double eventTime, eventType;
     Source *source;
     Activity *activity; //… More properties
public:
void processEvent(){
   select EventType{
      case ArrivalEvent:
     source->newArrival(Entity);  break;
       case EndofService:
     activity->endofService(Entity) break;}}
//… Additional Properties
};
```

If the user wants to add additional events (e.g., a monitor event), it would require the designer to add an appropriate data member, data methods, and then provide an additional case statement. Therefore, the designer has the impossible problem in anticipating every kind of event.

3.2 Object-Oriented Extension

An object-oriented simulation deals directly with the limitation of extensibility by permitting full data 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 and can extend existing types. For example, 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. There are two basic mechanisms in C++ that allow OOS to provide for extensibility: **inheritance** and **genericity**.

3.2.1 Inheritance

Inheritance allows classes to exploit similarity through specialization of parent classes (i.e., child classes inherit the properties of the parent and extend them). All event types have an associated `eventTime` and `eventType` and the appropriate data methods to specify these properties. Therefore, specific event types would inherit these properties and provide additional ones (see Figure 1).

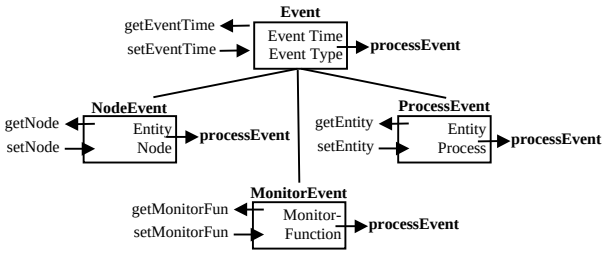

Figure 1: Inheritance Hierarchy

For example, `NodeEvent,` which provides events that occur at nodes (e.g., end of service at an activity), provides a pointer to the `Node` of interest and the `Entity` which caused the event. The `processEvent()` is declared virtual so that the appropriate `processEvent` is fired when the event is pulled off the calendar. The Event's `processEvent`() is a pure virtual function meaning any child classes must re-define it. The `NodeEvent`'s invokes the nodes `executeLeaving`() (another virtual function in the node hierarchy).

```
//Event's processEvent
void virtual processEvent() = 0

// ProcessEvent's processEvent
void virtual processEvent(){
  processPtr->executeProcess(entityPtr);}

//NodeEvent's processEvent
void virtual processEvent(){
      nodePtr->executeLeaving(entityPtr);}
//ExecuteLeaving -virtual function in Node
```

Now the designer does not have to anticipate every type of event. Users have the ability to define their own events provided they inherit from an existing event class and provide an appropriate `processEvent`() function.

3.2.2 Parameterized Types

Even with inheritance, many O-O languages can still be limiting in terms of extensibility. C++ provides an addi-

tional method of extensibility called genericity or parameterized types (i.e. templates). Parameterized types are special forms of composition that exploit commonality of function. For example, most simulations would declare a source object which is used to place entities into the network. In an OOS environment, the user may want TVs or Orders to arrive rather than generic entities. The user can create several different source nodes by inheriting from the base Source class as seen in Figure 2. Each of the new classes defines a new type of object to be created (i.e., TV, Order) and the "virtual function" `executeLeaving`.

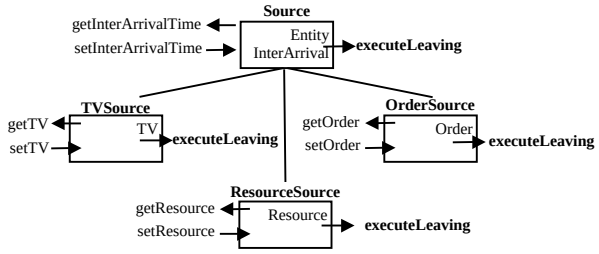

Figure 2: Inheritance Hierarchy versus Commonality

Notice, only the `Interarrival` object and methods are re-used in the child class. Each of the child classes must define its own `executeLeaving`() when the only difference is the type of object released into the network. When objects provide the same functionality, parameterized types are used (see Figure 3.). Now, the user specifies the type of entity to be released into the network and all remaining code is used. This ability is further demonstrated when a user wants to add statistics to the source node. The user only has to inherit from one class rather than create a TVSourceStat, OrderSourceStat, etc.

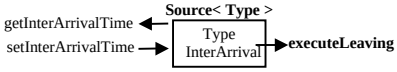

Figure 3: Parameterized Type

The following would declare two different source nodes.

```
Source<TV> tvSource(…);
Source<Order> orderSource(…);
```

4 CREATING A SPECIFIC OOS

A key to the creation of a fully integrated simulation package is the use of a *class inheritance hierarchy*. 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.

While frames provide a convenient means to describe the levels of abstraction within the entire object-oriented simulation platform, another means of encapsulation is to place higher level complex interactions into "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.

Special simulation languages and packages may be created from these object classes. For more information, see Joines and Roberts (1996). YANSL is an acronym for "Yet Another Network Simulation Language" and is just one *instance* of the kind of simulation capability that can be developed within an OOS 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 similar languages, but without the "bells and whistles."

4.1.1 Classes Specific to YANSL

Several classes are chosen from the modeling frameworks (Joines and Roberts, 1996) 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 shown in Figure 4. 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.

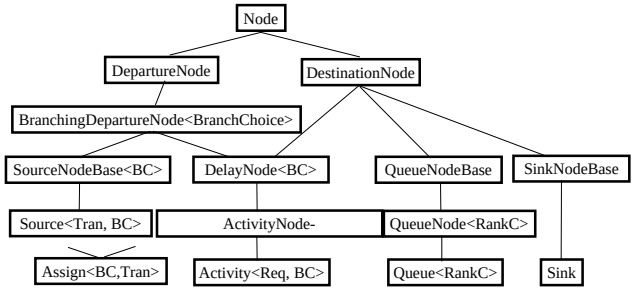

Figure 4: YANSL Node Hierarchy

The delay and assign nodes are both departure and destination nodes, so they inherit from both the departure and destination node classes. Departure nodes may need a branching choice and called `BranchingDepartureNodes`. 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, shown in Figure 5 allows resources to be identified as individuals, as member of alternative groupings, or as members of teams.

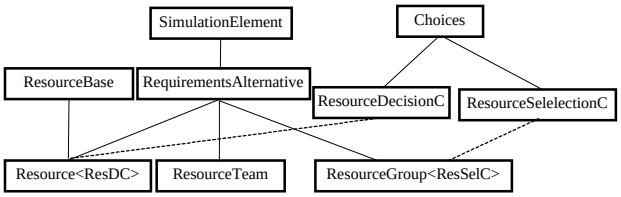

Figure 5: Resource Framework

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, shown in Figure 6, extend those in the modeling frameworks.

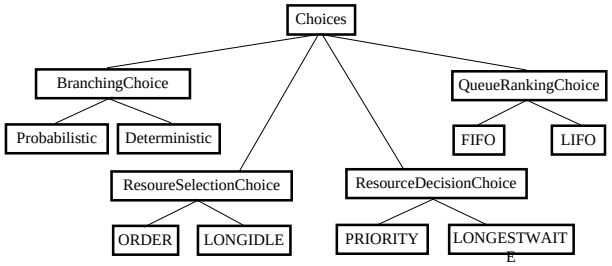

Figure 6: YANSL Choice Hierarchy

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. 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 features. 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 predefined 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.

The lack of distinction between the base features of YANSL and any extensions illustrate the "seamless" nature of user additions. Many embellishments are possible (e.g., see Joines and Roberts 1996) could be applied. 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. For those familiar with some existing simulation languages, consider the difficulty of doing the same.

5 CONCLUSIONS

Modeling and simulation in an O-O 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 become a language of choice by many computer users. 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

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

Joines, J.A. and S. D. Roberts. 1995. Design of object-oriented simulations in C++. In *Proceedings of the 1995 Winter Simulation Conference*, ed. Christos Alexopoulos, Keebom Kang, William Lilegdon, and David Goldsman, 82-92,Washington, D.C.

Joines, J.A. and S. D. Roberts. 1996. Design of object-oriented simulations in C++. In *Proceedings of the 1996 Winter Simulation Conference*, ed., John Charnes, Douglas Morrice, Dan Brunner, and James Swain, 65-72. Institute of Electrical and Electronics Engineers, San Diego, CA.

AUTHOR BIOGRAPHIES

JEFFERY A. JOINES is a Research Associate in the Furniture Manufacturing and Management Center at NCSU. He received his B.S.I.E, B.S.E.E, M.S.I.E and Ph.D. I.E. from NCSU. He is a member of INFORMS, IIE, and IEEE. His interests include O-O 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 WSC.