Proceedings of the 1992 Winter Simulation Conference
ed. J. J. Swain, D. Goldsman, R. C. Crain, and J. R. Wilson

OBJECT-ORIENTED MODELING AND SIMULATION WITH C++

Jeffrey A. Joines
Kenneth A. Powell, Jr.
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 build and simulate object-oriented models in C++. An object-oriented network based simulation language called YANSL, which is fully compatible with C++, is introduced and is used to create a network queuing model of the TV inspect and repair problem. YANSL has the "look and feel" of existing network simulation languages, but possesses the benefits of an object-oriented design including the use of classes, inheritance, encapsulation, polymorphism, run-time binding, and parameterized typing. These concepts are used to implement several seemingly difficult embellishments to the example in such a way as to extend the language. Object-oriented simulations provide full accessibility to the language, faster simulations, portable models and executables, a multi-vendor implementation language, and a growing variety of complementary tools.

1 INTRODUCTION

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

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

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

1.1 The Problem

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

1.2 A Solution

Object-oriented simulation deals directly with the limitation of extensibility. The principle reason for the development of object-oriented concepts is to permit 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.

1.3 Purpose of this Paper

The purpose of this paper is to illustrate object-oriented simulation using the C++ language. C++ is an object-oriented extension to the C programming language (Lippman 1991). We will use C++ to illustrate the "extensive/intensive" nature of object-oriented simulation (OOS) within the framework of the popular network-based simulation approach. Through common simulation examples, the utility of classes, encapsulation, inheritance, overloading, run-time binding, and parameterized typing are demonstrated. OOS is shown to support extension in a fashion that grants each user vendor access to changing internal function and adding new objects.

2 YET ANOTHER NETWORK SIMULATION LANGUAGE (YANSL)

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

2.1 Basic Concepts and Objects in YANSL

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

2.2 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. Just as in other network languages, 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.

2.3 The YANSL Model

The YANSL network has all the graphical and intuitive appeal of any network based simulation language. A graphical user interface could be built to provide "convenient" modeling with error checking and 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, DET >
       tvSource( interarrival, 0.0, 480 );
     // Begin at 0.0 and quit at 480.0

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

   /** Repair **/
Queue< FIFO >         repairQueue;
     Repairman.addQueue( repairQueue );
Activity< DET >       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.

The statements in YANSL are very similar to those in SIMAN, SLAM, or INSIGHT. By the way, this is all legitimate C++ code -- which we will discuss in detail later. Also this model runs in half the time a SIMAN model runs on the same machine! But the real advantage of YANSL is its extensibility.

2.4 The Objects and their Specification

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

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

2.4.1 Object Declarations

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

```
Exponential     interarrival( 5 ),
                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." 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 method 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. Further, in an effort to give our YANSL users a full range of "nice" basic classes, such classes as `string` and dynamic `array` with range checking are also available. Because an object-oriented language doesn't distinguish any differently between the C++ classes and the ones we have added, use of all these classes is very similar. In the computer literature, this property of having user objects treated like built-in objects means everything is treated as a "first class" object.

2.4.2 Using the Objects

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

```
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.

2.5 Running the Simulation

The prior model is compiled under a C++ compiler(a compiler should be AT&T version 3.0 compatible), linked with the YANSL simulation library, and executed. Currently, the YANSL simulation library has been compiled under Borland C++ 3.1 (Borland 1992) and GNU C++ (GNU 1991). C++ is strongly typed, so error checking is very good. Also, if an environment such as Borland is used, the language can be used under Windows or DOS and take advantage of all the Borland tools such as the object browser and interactive debugger.

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

3 CLASSES AND THEIR USE

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

```
#ifndef EXPON_H
#define EXPON_H

#include "random.h"

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

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

#endif
```

The class definition determines the properties of an object.

3.1 Class Properties

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

3.2 Inheritance

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

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

3.3 Run-time Binding

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

3.4 Construction and Initialization of Objects

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

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

3.5 Polymorphism

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

4 EMBELLISHMENTS TO THE TV MODEL

To illustrate the broader use of an object-oriented simulation language, we present several embellishments to the TV inspect and repair problem. Although these embellishments may appear very complicated, they are handled easily and provide direct extensions to YANSL.

4.1 Add a "floating" Resource

YANSL is capable of modeling "floating" resources which can service more than one queue. Suppose we add a third worker who can inspect but will help repair when there is nothing to inspect. The following additions are made to the model, which add the worker, specifies the worker decision process when the worker finishes a job, and specifies the selection among alternative resources at the activity nodes:

```
//Add the new Resource, specify served queues
Resource<  PRIORITY > inspectRepairman;
   inspectRepairman.addQueue(inspectQueue);
   inspectRepairman.addQueue(repairQueue);

//Add the Resource Selectors for activities
ResourceSelection< ORDER >   inspectList;
   inspectList.addResource( inspector );
   inspectList.addResource( inspectRepairman );

ResourceSelection< ORDER >   repairList;
   repairList.addResource( repairman );
   repairList.addResource( inspectRepairman );

//Add at the Inspect Activity
inspection.addRequirement( inspectList );

//Add at the Repair Activity
repair.addRequirement( repairList );
```

A new resource called `inspectRepairman` is now declared and the `addQueue` function states that this person will serve, in `PRIORITY` order, the `inspectQueue` and `repairQueue`. Since both the inspection and the repair activities now have a choice of resources, two resource selector objects called the `inspectList` and the `repairList` are created which will be used to specify how the resource is chosen from the alternatives. In this case, the resource is selected on the basis of `ORDER`. Finally, at the two activities, the `addRequirement` function specifies the resource selector object rather than the resource object. This overloading of the `addRequirement` function argument is an example of polymorphism applied to user-defined classes. Therefore, a user of YANSL now may specify a requirement involving several resource alternatives with the exact same form used to specify a single resource and new decision rules may be easily included.

4.2 Derive a New Type of Transaction for TVs

So far we have used the YANSL transaction class to represent TVs, but now we would like some way to distinguish the TVs that are newly arrived from those that have been inspected to those that have been repaired. In a network simulation language, this distinction would be obtained by assigning attributes to the transaction. The same can be done by extending YANSL as follows:

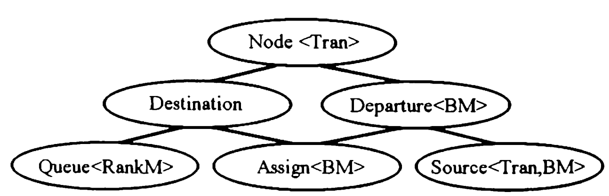
```
#ifndef TVTRANS_H
#define TVTRANS_H

#include "transact.h"

class TV : public Transaction
{
  public:
    TV(){ numRepairs = 0; }
    void setColor( int cr ) { color = cr; }
    void incrementRepairs(){ numRepairs++; }
    int getNumRepairs(){ return numRepairs; }
    int compare( void * );

  private:
    int numRepairs;
    int color;
};

#endif
```

First, we derive a new type of transaction called a TV and give the TV two private properties corresponding to the number of repairs and the color of the TV. The public functions set the value of color, increment repairs, and get the value of the private data containing the number of repairs.

Although this is a simple change, the TV could be given more complex properties, such as some kind of repair order object (a has-a relationship). The TV is a derived class from Transaction (an is-a relationship). Because a TV is a kind of transaction, all the things transactions can do TVs can do. Thus, there is no need to write any special code or do anything special for TVs as they inherit all the functionality of the transactions.

4.2.1 Add Assignment Node

Now that there is a property of TVs for repairs, there needs to be some kind of assignment node that can cause the property to be changed. We need to add a node to YANSL. In YANSL, node classes are formed in a class "hierarchy." This hierarchy starts with a broad division of nodes and specifies more specific nodes lower in the hierarchy. Nodes lower in the hierarchy inherit the properties of the nodes above them. A portion of that hierarchy is given below:



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

```
#include "node.h"

template< class BM >
class Assign : virtual public Destination,
virtual public Departure< BM >
{
  public:
   virtual BOOL executeEntering(
               Transaction* tptr )
   {
    ((TV*)(tptr))->incrementRepairs();
    branch.nextNode()->executeEntering( tptr );
    return TRUE;
   }
   virtual void executeLeaving(
               Transaction* tptr ){}
};
```

Multiple inheritance is specified in the header of the class definition. The `executeEntering` and `executeLeaving` are virtual functions in departure and destination classes that act as placeholders, permitting the assignment node special functionality as transactions enter and leave (remember that TVs are simply a kind of transaction and thus they can use the assignment node). In this case the assignment node simply increments the number of repairs.

4.2.2 Add Ranking Method to the Queue

Now that TV objects remember their repairs, let us show how to extend the Queue node so it handles ranking of TVs according to the number of times they have been repaired. Recall from the original model that queue nodes are parameterized by a `RankingMethod`. So far all we have specified is the `FIFO` class. Because of our foresight in having a "parameterized" queue class, we can easily add a new ranking method. Ranking methods are encapsulated as classes so they can be easily modified. Again, a new class is needed:

```
class SORT : virtual public RankingMethod
{
public:
  virtual void addtoQueue( Transaction *tptr );
  virtual Transaction* removeFromQueue();
  virtual int rankInQueue( Transaction *tptr );
};
```

The virtual functions in this class must be completed to perform the sort. Now the queue at the inspection activity would be specified by:

```
Queue< SORT >        inspectQueue;
```

Parameterized types create templates for classes so that the ultimate specification of a class is not known until that class is declared to create the object. 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. Some network simulation languages approach this issue by having more general node types, like an "operation" node, but these general types cannot, in general, yield specific objects -- only their subtypes create objects (in C++, such a class would be "pure virtual class").

4.2.3 Change Inspection Time to Depend on Repairs

Another interesting change in the basic model is to make the inspection time depend upon whether it had been repaired or not. We add a new kind of activity:

```
template< class BM >
class InspActivity : public virtual
                      Activity< BM >
{
  public:
    InspActivity( Random*, Random *);
    virtual BOOL
            executeEntering( Transaction * );

  protected:
    Random   *repairVariate;
};

template< class BM>
InspActivity< BM >::InspActivity( Random
       *actTime ,Random *repTime)
       : Activity< BM >(actTime)
{
   repairVariate = repairTime;
}

template< class BM >
BOOL InspActivity<BM>::executeEntering(
       Transaction *tptr )
{
  //... same as activity class

  /* If the TV has been repaired Inspection
     time is different */

  ( !((TV*)tptr)->getNumRepairs() ) ?
        scheduleEvent( tptr,
               actVariate->sample() ) :
        scheduleEvent( tptr,
               repairVariate->sample() );

  return TRUE;
}
```

This new kind of activity, called the `InspActivity`, uses all the properties of the activity, but provides a different activity time if the TV has been repaired. Notice that only a placeholder for the new time variate is needed, along with a definition of the `executeEntering` function.

4.3 Add "grouped" Transactions

Suppose that good TVs are accumulated on a conveyor in front of a palletizer where eight are gathered into a group and palletized. Now a single pallet leaves the palletizing activity rather than eight TVs. This problem requires that we specifically accumulate and then combine eight TVs into a single object. Also the activity should not process eight objects but only one, and only one object should leave the palletizer. A new kind of node for grouping and managing transactions might be defined as:

```
template< class T, class BM >
class Group : virtual public Destination,
       virtual public Departure< BM >
{
  public:
    Group( int max )
       { current = 0;  target = max; }
    virtual BOOL executeEntering(
                        Transaction* tptr )
    {
      if( ++current == target )
      {
       T* tnew = new T;
       branch.nextNode()->executeEntering(
                                        tnew );
       current = 0;
      }
      delete tptr;
      return TRUE;
    }
    virtual void executeLeaving(
                        Transaction* tptr ){}

  private:
    int target;
    int current;
};
```

C++ provides a simple means to create and destroy objects through its `new` and `delete` memory management operators. These operators can be overloaded to apply to specific objects, if needed.

4.4 Related Embellishments

Many more embellishments are simply parallel application of the approaches used in the prior sections. For example, a new branching method or a new resource selection method or a new decision method can be added just as we added the ranking method. Various methods for processing network objects is common in most network simulation languages, but adding new methods can be a difficult chore since most of the internal functioning of the language is unavailable to the user. Notice that new kinds of transactions, different types of resources, and new nodes are simply added. 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 AGV systems might be created and called YANSL-AGV for those special users. Perhaps the AGV users would get together and share extensions and create a more general YANSL-AGV II. And so it goes! For those of you familiar with some existing network simulation language, consider the difficulty of doing the same.

5 CONCLUSIONS

Modeling and simulation in an object-oriented language like C++ 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.

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. Many vendors are offering compilers, using the latest compiler technology to produce optimized code for many machines. Your C++ code is portable. We have run YANSL on personal computers under DOS and Windows using the Borland compiler and on workstations using GNU G++. With the new C++ standard (Ellis and Stroustrup 1991), all C++ compilers are expected to accept the same C++ language. Executables are standalone and portable. 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 simulation languages require some proprietary executive to run.

By having many vendors, the price of C++ compilers is low, while the environments are first class. 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. At this writing, more CASE (Computer Aided Software Engineering) tools are appearing to make development more organized and orderly. Also numerous class libraries for windowing, graphics, and so forth are appearing that are fully compatible with C++. It is easy to see graphical user interfaces for simulation modeling, animation of simulation, and statistical analysis of simulation results completely captured by several vendors. Their interoperability would be insured by their use of a common means for defining and using objects.

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

REFERENCES

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

Ellis, M, and B Stroustrup. 1991. *The Annotated C++ Reference Manual*. Reading, Massachusetts: Addison-Wesley.

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

GNU C++ Version 2. 1991. Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139.

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

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. 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 real time control.

KENNETH A. POWELL, JR. is currently finishing a M.S.I.E in the Department of Industrial Engineering at North Carolina State University. He received his B.S.I.E in December 1990 from NCSU. He is a member of Alpha Pi Mu and IIE. His interests include simulation, object-oriented programming, and experimental design.

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