Proceedings of the 1991 Winter Simulation Conference
Barry L. Nelson, W. David Kelton, Gordon M. Clark (eds.)

OBJECT-ORIENTED SIMULATION

Diane P. Bischak
Stephen D. Roberts

Department of Industrial Engineering
North Carolina State University
Raleigh, North Carolina 27695-7906

ABSTRACT

Object-oriented simulation languages provide simulationists the tools to produce modular, re-usable simulation models. In this tutorial we discuss the major features of object-oriented languages and their application in simulation. We also discuss some of the problems with and areas of special potential for object-oriented simulation.

1 INTRODUCTION

The appeal of object-oriented simulation is that it conforms to the notion that the world is composed of "objects." If you are modeling a factory floor, you see lots of objects—people, machines, pallets, forktrucks, conveyors, etc. If you are modeling a hospital emergency room, you see lots of objects—doctors, nurses, examining rooms, medical records, x-ray machines, etc. It's not difficult to view the real world as a set of objects which interact with each other. Thus an "object-oriented" simulation language or modeling system has great intuitive appeal.

Moreover, it's natural to describe things that are not physical as objects. For example, a record in a data base might easily be seen as an object. The data base would then be a collection of objects. Even mathematical symbols could be viewed as objects. For instance, the symbol y may be an object that represents a variable. The symbol 2 may be viewed as the object representing the numerical quantity 2. Likewise $2y$ would be the object that represents 2 times the variable y. Although the idea that the most elementary notions can be described as objects may seem a little far-fetched, these examples demonstrate the descriptive appeal of the word "object."

In this tutorial we give an overview of the object orientation, its features, and its potential for use in simulation. Like "intelligence" in the artificial intelligence community, the word "object" in the object-oriented community is used for a variety of purposes (Winblad, Edwards, and King 1990; Wirfs-Brock, Wilkerson, and Wiener 1990). We will limit our discussion of object-orientedness to simulation. It is of historical interest to note that Simula, a simulation language, was the first computer language to promote the object orientation (Birtwistle et al. 1979).

2 THE OBJECT ORIENTATION AND SIMULATION

In an object-oriented simulation language, the description of objects and their interaction plays the central role in modeling. Each real-world object is modeled as an "object" in the language. Technically, an object is an abstract data type plus the set of operations or functions that can be performed on that data type. A fundamental property of objects is that they are encapsulated: their data usually cannot be accessed except through the operations defined on them. Thus an object's set of operations forms an interface through which the rest of the objects in the model can interact with it.

An object-oriented simulation language can be distinguished from other simulation languages by its support for the creation and description of objects. Object-oriented languages support the crafting of new objects as an extension of existing objects, and a new object's behavior can be described from the behavior of existing objects or from a combination of the behaviors of several objects. Some proponents of object-oriented simulation will argue that the ability to re-use and extend existing objects and object behavior in this way is the central feature that defines an object-oriented simulation language.

Using an object-oriented simulation language does not force you to use a particular simulation worldview. It is even possible to write an event-oriented simulation using an object-oriented simulation language. The object orientation is a programming lan-

guage orientation, not a simulation orientation, and the language is a tool, not something that forces you into a single mode of operation. Instead of thinking of the object orientation as a separate world-view or simulation orientation, it is more useful to think of it as a different way to define entities—the way they act, interact and communicate with each other—and to organize the types of entities in the model and the data or attributes which each entity possesses.

There is, however, a relationship between the object orientation and the process world-view in simulation. In contrast with procedural-based programming languages such as Pascal or C, the programming constructs of an object-oriented programming language are mainly concerned with the communication among objects which result in changes in the values belonging to an object, rather than with function calls to which variables are sent to have their values changed. The focus is on the objects rather than on the functions. A rich variety of methods of communication and interaction between entities is also readily available in an object-oriented language.

Thus, the greatest benefit from using an object-oriented simulation language accrues when writing a process world-view simulation, because unlike the event-scheduling or activity-scanning world-views, the process world-view is most centrally concerned with the objects or entities in the model and their lives while in the model.

Prior to a more detailed discussion, let's address several questions which are often raised regarding object-oriented simulation (OOS).

3 ISSUES IN OBJECT-ORIENTED SIMULATION

What are the central virtues of OOS?

Without going into too much detail, we will suggest that OOS offers significant potential over existing popular simulation languages in several respects. First, perhaps the most convenient feature is reusability. Although a language like GPSS (Schriber 1990) offers extensive re-use of predefined objects like transactions and gates, it does not permit the modeler to create their own versions of a transaction or gate. Objects defined within an object-oriented language are inherently extensible. You can craft new objects out of existing ones. For example, an AGV (automated guided vehicle) object may be crafted from a fork-truck object, since many of the properties of the two are similar. Second, object-oriented simulations are modular, with objects being the modules. With modularity, all the information known about the object is held in one place; you don't need special procedures to find information. This encapsulation of information means that changing the meaning of an object or modifying its behavior is easy to do, and changes to the object can be easily maintained.

What is the history of OOS?

Early simulation languages like GASP (Pritsker 1974) and Simscript (Russell 1989) focused on time-flow control through the handling of simulation "events." In an event-oriented simulation, the modeler identifies circumstances when a system changes state (i.e., events) and describes what happens when an event takes place (in event procedures). SIGMA (Schruben 1991) embodies this approach with simulation (event) graphs. The next generation of simulation languages, like GPSS, SLAM (Pritsker 1986), and SIMAN (Pegden, Shannon, and Sadowski 1990) emphasized the flow and processing of entities within a routing (network). The notion of entities flowing through a process is a natural paradigm for production systems and forms the basis for several "manufacturing" simulation languages. The concept of a process having its own lifetime was prominent in Simula and the later version of Simscript. Object-oriented simulation languages like Simula, MODSIM (Belanger 1990), Sim++ (Lomow and Baezner 1990), and Smalltalk-80 (Goldberg and Robson 1989) continue the emphasis on entities but provide the user with simulation facilities and the ability to craft their own objects, rather than relying on pre-defined entities.

Aren't entity flow languages like SIMAN, SLAM, and GPSS object-oriented?

It might be argued that these are object-oriented since transactions, resources, queues, activities, etc. are objects. In many ways these languages are immediate predecessors to OOS languages. But while users can "specify" the nature of the transaction, the queue, or the sink, the user cannot create their own objects. Instead users must resort to either finding ways to use the pre-defined objects as a way to "model" something else, or they must write their own programs (typically through FORTRAN or C) to achieve their ends. With an OOS language the user can define new types of objects very easily. There is no longer the necessity of trying to match what you want to do in the simulation model to the limited number of constructs available in the simulation language.

Isn't OOS just simulation using an object-oriented programming language?

Any general-purpose programming language can be used for simulation, but we wouldn't call it a simulation language. Similarly, just because a language is object-oriented, we won't say it is an object-oriented

simulation language, even though it may be used for simulation. We prefer to reserve the name OOS to those languages that have convenient simulation use and have object-oriented facilities.

In the following two sections we discuss features of an object's data, its operations, and other properties. The examples given are in a "pidgin" C++ (Lippman 1991) which is meant to show the general nature of declarations in an object-oriented language.

4 OOS: CRAFTING OBJECTS

Before creating an object, the type of the object must be described. Another word for "type" is "class." Note that built-in object types like `int` (integer) and `float` (real) in general programming languages do not need to be described since their meaning (i.e., their type) is generally well-understood. Object types like `queue` and `activity` in a simulation language do not need to be described since the designers of that language have already determined what such objects are like. However, when no pre-existing description exists, an object description is needed.

4.1 Describing an Object

Object descriptions are similar to record descriptions in Pascal or structure descriptions in C in that an object type is viewed as a composite of more elemental types. An object description is a template for creating objects. For instance, objects of type `vehicle` might be described as a class of objects whose component objects consist of the make of the vehicle, its capacity, and its speed:

```
class vehicle
        {
        string make;
        int capacity;
        float speed;
        };
```

A newly-defined class becomes a new type. Thus `vehicle` is now a data type within the language in a fashion analogous to an integer or real data type. Object classes may be stored in a library that can be drawn upon in order to write a simulation; this enhances the possibility of re-use of objects across models.

4.2 Creating Objects

A created object is said to be an "instance" of a class. Just as in the above example `capacity` is an instance of type `int`, an instance of the class `vehicle` might be obtained by:

```
vehicle fork_truck;
```

where `fork_truck` is one object instance from the class of vehicles.

Instances can be obtained statically (where the language permits) or dynamically. The previous example is an illustration of a statically created object, obtained when the instantiation occurs and continuing until the context is destroyed. Instances of objects may be created dynamically in a fashion similar to the use of the "new" construct in Pascal or "malloc" in C. In Pascal or C, an object created with the "new/malloc" function is a representative of the data type referred to with the memory management function. The same is true in an object-oriented language. One construct to create a `fork_truck` dynamically might be:

```
fork_truck = new vehicle;
```

This construct causes an object of the class `vehicle` to be created and assigned the name `fork_truck`. In this case, `fork_truck` may be a "reference" to the object.

4.3 Using Other Objects

Often in simulation, one object is similar to another. For example, an AGV is a vehicle but is somewhat different from a fork-truck in its use, so that instances of AGV cannot be created from class `vehicle` defined above. AGVs need their own class. It would be efficient and more error-resistant if we could form the AGV class by modifying the design of the `vehicle` class. Re-use of a "parent" class in this way is called "inheritance." Inheritance is one of the foundational ideas of object-oriented software.

4.3.1 Inheriting From One Parent Class

If a new class you wish to create has similarities to a previously-defined class, you can exploit that fact and avoid duplicating your efforts. For example, suppose you wanted to model vehicles that were restricted to operating in aisles. Thus you might use the following description:

```
class fixed_path
        {
        string make;
        int capacity;
        float speed;
        float xloc, yloc;
        float distance();
        };
```

where the function `distance()` computes some rectangular distance based on aisles and the location is given by (`xloc, yloc`). But this description includes all the properties of a `vehicle`, which we have previously described. Taking advantage of that prior description and "inheriting" its properties, one might then use the description:

```
class fixed_path: vehicle
        {
        float xloc, yloc;
        float distance();
        };
```

with the understanding that now the `fixed_path` class is derived from the `vehicle` class.

In a more complicated (and realistic) example, you may be interested in designing an `AGV` class and realize that the simulation library you are using already has a `vehicle` class for vehicles such as fork-trucks. You know that vehicles have a "pick up load" function and your class for AGVs will have a "pick up load" function also. You could create a "superclass" called `conveyance` that has all the data and functions of general conveyances. The three classes can then be set up in a hierarchical structure so that when you create your `AGV` class, you call it a "subclass" of `conveyance`. Immediately and with no further work your `AGV` class has inherited the component data and functions of a `conveyance`. You can then alter the inherited functions to make them appropriate for an AGV (this is called "overloading", meaning "loading over the previous or default meaning") or add data or functions as required for the special properties of an AGV. The superclass is then the "parent" class for both the `vehicle` and `AGV` classes.

4.3.2 Multiple Parents

It is possible in certain OOS languages such as those based in C++ to allow a class to have more than one parent class. Thus, for instance, an AGV could be considered both to be a `conveyance` (because it moves loads) and to be a `resource` (because it can become idle, perhaps needs to be recharged, etc.) and to have the properties of each (a conveyor might be considered to be a conveyance but not be considered a resource; a fixed machine would be a resource but not a conveyance). This may be useful if, for example, different statistics are gathered concerning conveyances and resources, and we want to gather both on AGVs. If inheritance from more than one parent class is possible in the language, portions of several classes may be re-used to obtain a new class. This use of more than one parent class is called "multiple inheritance".

To illustrate with a simple example, suppose you have the `fixed_path` class that was described previously. Now suppose you have a `resource_statistics` class that collects resource utilization statistics, such as percent of time in busy and idle states. You might create a new class called `fork_truck` by:

```
class fork_truck: fixed_path,
                  resource_statistics
        {
        int id_number;
        string name;
        enum {BUSY, IDLE} status;
        int state();
        };
```

where the class `fork_truck` "inherits" all the properties of `fixed_path` vehicles and the properties of `resource_statistics` collectors. The `id_number` and `name` are needed to know the identity of the `fork_truck`, and the `state()` function determines the status of the truck, which is needed both for statistics collection and to establish the availability of the truck.

With multiple parents, a problem can arise if a name is not unique across parents. If two parents have the same name for different properties, the OOS language must have a way to resolve the inheritance conflict or to request that the user make a resolution.

5 OOS: OBJECT PROPERTIES

We have spoken about the properties of objects rather loosely. You should notice that object-oriented terminology tends to treat an object's data properties (like integers) and functions in a similar fashion. They are both viewed as properties. This perspective is used because it gives the designer of object classes complete flexibility over how a property is implemented. For instance, consider the property `time_busy` of a resource. A designer of a `resource` class may implement that property by accumulating each unit of time busy in a floating point variable. On the other hand, suppose the designer has already implemented the collection of `time_idle` and `total_time` in the system for a `resource`. Thus `time_busy` might be determined by a function that computes:

```
float time_busy(void)
        {
        return total_time - time_idle;
        };
```

The manner of implementation is irrelevant to the user, who only needs to know that a resource has a

time_busy (assuming all they need is its value and do not need to change it).

5.1 Visibility

Since users of a class don't need to know how things are implemented, most object-oriented languages give class designers control over what users need to see. This is often called the "class interface." The interface describes what users of the class can access. For example, in C++, the keywords "public" and "private" are used in class descriptions, and only the "public" properties become visible to the users of the class. A class that is described as:

```
class simulation_time
        {
        private:
                float t_now;
        public:
                float begin;
                float current();
        };
```

has "private" and "public" properties. An object's data will usually be private, since class designers generally do not want users to be able to change that information directly. Read-only access to information is then provided by public functions. In this case the interface to the user of the class becomes:

```
class simulation_time
        {
        public:
               float begin;
               float current();
        };
```

Here users have read/write privileges for **begin** but read-only for **current()**. Now if a user creates an instance of the **simulation_time** class as:

```
simulation_time time;
```

they can refer to the properties of their **time** object directly as:

```
time.begin = 0.0;
        /* set beginning time to 0.0 */
queue_entry = time.current;
        /* set queue_entry variable
           to value of current time */
```

Most object-oriented languages permit variables and functions to be referenced in the same format (note we dropped the parentheses for **current**).

5.2 Scope

While the visibility of object properties relates to what the user can access, there is also the question of scope. In traditional programming languages scope refers to the visibility of a variable or function in or out of its procedural context, whereas in an object-oriented language scope will include reference to an object context. The most common scope differentiation with respect to objects relates to instance and class variables.

5.2.1 Instance Variables

Each instance of an object class has its own variables, which are usually declared as private data. For example, each instance of a **fork_truck** has its own i.d. number and name. A user may have access to the i.d. and name through public functions as in the following description:

```
class fork_truck
        {
        private:
                int id_number;
                string name;
                ...
        public:
                int id_number();
                string name();
                ...
        };
```

Here **id_number** and **name** are "instance" variables, and each **fork_truck** object will have its own **id_number** and **name**.

5.2.2 Class Variables

Sometimes a variable is a property of all the objects in a class. Such variables are "class" variables. As an example, consider how the i.d. number for the objects from the **fork_truck** class are established. We would like to keep some variable such as "number" visible to all the objects in the **fork_truck** class. That way, when assigning an i.d. for a new object, we can be certain not to duplicate the i.d. of another object from the class. For instance, when an object from **fork_truck** is being created, we would like to do the following:

```
id_number = number + 1;
       /* set the id for this object */
++number;
       /* recognize another object */
```

In such a case the variable **number** is a class variable and we might recognize it in the class description as:

```
class fork_truck
        {
        int number;
        private:
                int id_number;
                string name;
                ...
        public:
                int id_number();
                string name();
                ...
        };
```

One place a class variable is useful in simulation is to seed the random number generators so they do not all begin at the same place in the random number sequence.

5.3 Object Creation and Destruction

Although we have described how to create objects statically and dynamically, we have not described how the class designer can affect the creation (and destruction) of an object. Generally, whenever an object is created the process of creation involves execution of a "constructor." Whenever an object is destroyed, a "destructor" is executed. Although object-oriented systems usually provide a default constructor and destructor for any class, class designers may provide their own.

5.3.1 Constructors

A constructor could be used to assign a name to created instances of the `fork_truck` class:

```
class fork_truck
        {
        int number = 1;
             /* initialize to one! */
        private:
                int id_number;
                string name;
                ...
        public:
                int id_number();
                string name();
                fork_truck(string id_name);
                ...
        };
```

The function `fork_truck()` is the constructor, taking a string argument, possibly implemented by:

```
fork_truck::fork_truck(string id_name)
        {
        copy(id_name, name);
              /* copy id_name to name */
        id_number = number++;
              /* use and increment */
        };
```

which sets the name and the i.d.

5.3.2 Destructors

A destructor is used to handle some kind of finalization associated with the destruction of an object. For example, upon destruction of a `fork_truck`, some resource statistics might be printed. A destructor for a `fork_truck` might look like:

```
fork_truck::~fork_truck()
        {
        resource.print();
          /* print statistics on this object */
        };
```

Destructors provide a convenient means of cleaning up when the object is destroyed.

5.4 Polymorphism

When we think about the behavior of objects, we may want different objects to employ the same behavior. For example, `fork_trucks` and `stacker_cranes` may both have the ability to pick up and deliver unit loads. Perhaps the function `pick_up()` should apply both to `fork_trucks` and to `stacker_cranes`. The ability to make different objects appear to do the same thing is called polymorphism.

Much of arithmetic in procedural programming is polymorphic. For example, two integers or an integer and a float are both acceptable addition operands. The addition operator (and other operators) works for many combinations of atomic data objects.

The same should be true for functions. For instance, it would be convenient if the square root function, `sqrt()`, was defined for integers as well as for floats. Then `sqrt(2)` would have as much meaning as `sqrt(3.2)`. (It doesn't in most programming languages, even though it is a natural thing to do.)

The ability of an object-oriented program to interpret what function to call at run-time, rather than resolving the call at compile-time, is related to the concept of "dynamic binding." With dynamic binding the actual function called depends on the argument given at run-time rather than the type of a pointer or reference. Dynamic binding, also called "late binding," is a major contributor to polymorphism since the actual behavior of a function is determined only after the arguments are known.

5.4.1 Function Overloading

The use of the same function name to do something similar to different object classes is called "function overloading." Generally all that is needed is that the function have the appropriately specified parameters. Thus if the appropriate functions are declared in the `complex` class, the following would work:

```
double d = 4.8;
double t = sqrt(d);
     /* show sqrt for a double */
complex c (1.0, 3.4);
complex z = sqrt(c);
     /* show sqrt for a complex */
```

Here the name `sqrt()` is overloaded but performs a similar function on different object types.

5.4.2 Operator Overloading

Operators can be similarly overloaded to accept object classes as operands. For example, again assume `complex` is a user-defined class, but that the `+` operator is "overloaded" to accept complex objects as operands. Thus,

```
complex a (1.0, 4.0), b (3.6, 3.2);
complex c = a + b;
     /* add two complex numbers */
```

Operator overloading and operator definition facilities generally have limited use in simulation. However, one might find it convenient to have a time scale operator that converts common time units to a standard internal time. Thus end users could write an activity time as 3 hr or 180 min or .125 day at various points in the model.

6 COMMUNICATION AMONG OBJECTS

Throughout the preceding we have assumed that communication among objects takes place through the standard practice of function calls or variable fetching. Some object-oriented languages such as Smalltalk-80 provide for "message passing" as a more general way to communicate. When object-oriented programs in these languages execute, the objects send, receive, and interpret messages. For instance, a message from a unit load object may be sent to a fork truck object that it is ready to be picked up. The fork truck object will receive the message, interpret it, and act as it deems appropriate: it may respond directly, queue the message, or reply that it is busy and can't serve the sender.

In a message-sending environment, the sender doesn't tell the receiver how to interpret the message or what to do. The receiver will similarly act independently. The set of messages to which an object can respond is called a "protocol" and generally involves the functions contained in the object (just like a function call). Sometimes the functions are called "methods" to describe how an object responds to messages.

The message-passing paradigm may be more natural for a beginning programmer because it allows you to avoid thinking in terms of functions, variables, operators, and so forth. For experienced programmers it may be awkward to talk about `3 + 4` as the object `3` receiving the message `+` with the argument `4` (which is another object), but outside mathematically operations, message passing is a useful abstraction of communication among objects.

7 SIMULATION WITH OBJECTS

While the primary contribution of object-oriented technology to simulation is in the modeling orientation, there are several issues of specific concern to simulation. Generally speaking, simulation involves communication among objects over simulated time. It is convenient to talk in terms of discrete-event simulation, although this discussion easily extends to continuous or combined simulation.

There are three key questions that are a source of controversy in object-oriented simulation. They are (1) the treatment of processes, (2) the control of simulation time, and (3) the collection of statistics.

7.1 The Treatment of Processes

Simulation processes are familiar to everyone who does simulation modeling. However, a process in simulation has two interpretations, one of which is common in queuing simulations and the other in computer performance analysis.

In queuing, a process is a network of queues and activities through which pass transactions (also called entities or customers). As a transaction works its way through a network, it encounters queues where it must wait for service, and it encounters activities where it is served. In both cases simulation time advances—for an unpredictable amount of time in queues and for a predictable (usually by a known random variable) amount of time in activities. Here the process is the network. The network itself could be regarded as a process object that contains permanent objects for its nodes and branches. The transaction objects would be temporary, since they arrive and

leave. This first process view corresponds to network simulation models such as those of GPSS, SLAM, and SIMAN. In a sense the network has global scope while individual transactions have local scope and properties. The relation between objects and process are that the objects flow through the process.

The second process interpretation argues that process should be a property of the object. Such a view is offered by Simula and its successors. A process could be viewed as a function that can consume simulation time. Thus an object could act out a process just as it executes a function. However, its actions would depend on the advancing of simulation time. For instance, a given object could be running (utilizing time), idle (waiting to be activated), or terminated (not on the simulation clock). When an object contains its process in this manner, that object is able to act independently. Even in a queuing network, transactions could contain their own routing (process).

The value of permitting an object to possess its own processes is that processes can be regarded as a property of the object. Thus processes are encapsulated, they can be inherited, and, in general, a process is treated just like a function except that the execution of that "special" function can consume simulation time.

7.2 The Control of Simulation Time

Simulation time is typically kept centrally in a simulation, since synchronization with respect to a single clock is a key feature of a computer simulation. In an OOS model, time can be managed throughout the simulation within a single time object which would advance time and activate processes or call functions.

Another approach is to permit individual objects to control their own time. Thus an object would execute and advance its own simulation clock. Of course, all these object clocks would have to be synchronized with respect to some central clock, but that would only be done periodically. If a simulation model can be divided up into objects and objects can manage their own time, then there is the possibility of "parallel simulation," which we discuss in the next section.

7.3 The Collection of Statistics

As with the control of simulation time, statistics collection could be delegated to objects or dealt with centrally. Most simulation languages are written in a procedural language, so there is a tendency to focus on the functionality of statistics collection with the attendant statistics accumulation and display of tables and histograms. Centralizing statistics collection simplifies the creation of statistical methods, so that one function for computing a mean or a standard deviation can be applied to a variety of statistical interests.

On the other hand, decentralization of statistics collection to the object has its advantages. If statistics collection is done in an object, polymorphism can be accomplished since what appears to be a similar function can be obtained across objects. For example, the mean utilization of vehicles, fork trucks, AGVs, and so forth could be obtained in the same fashion:

```
fork_truck.utilization
```

or

```
agv.utilization
```

or

```
vehicle.utilization
```

would provide the same functionality.

8 PROBLEMS WITH OBJECT-ORIENTED SIMULATION

While this tutorial has explored some of the potential of object-oriented simulation, the approach is not without some drawbacks. Perhaps foremost among the difficulties with OOS is the fact that it represents a major paradigm shift from the "usual" procedural orientation. In simulation we are concerned with what happens among objects because most simulation languages take that orientation. This might be called a procedural orientation and, in fact, most languages permit only procedural extension, such as calling C or FORTRAN subroutines. Allowable changes to the basic object representations are very limited and are often restricted to changes in the variables associated with the transactions or globally associated with the model. Typically, the ability to change basic object representations is held by the software house.

Object-oriented simulation means that the user must now become a kind of language designer. No longer will the user have a set of pre-defined objects to employ; instead, there will be tools for crafting one's own objects. The user must not only know how to employ the objects but must know how to describe classes and implement them. The user must carefully consider the objects to be created as well as their use. Furthermore, the object orientation tends to demand that everything be represented as objects. Sometimes it's not easy to think of a queue as an object containing objects or a server as an object servicing objects in a queue. A good deal of time must be spend in

OOS just deciding what the object classes are and how they are to be used. Such a responsibility may be a burden.

A related issue is object management. Since users have the responsibility of describing and creating objects, they also have object management responsibilities. Two of the more complicating issues are the "gone/forgotten" problem and "garbage collection."

As an example of the first problem, suppose you have a reference to an object. In the process of handling the object, you may destroy the object but not the reference ("gone but not forgotten") or you may destroy the reference but not the object ("forgotten but not gone"). Such situations can create serious memory and storage problems. Many users don't like to deal with such problems.

Object management also means that the user often must keep a tidy model by getting rid of unneeded objects. This responsibility is often called "garbage collection" and it represents a very difficult problem. Most users don't like to deal with simulation details like file management. Object management can be even more complicated since more complicated structures like objects now exist.

Not every OOS provides a means of dealing with these problems. Hopefully better garbage collection algorithms and technology will reduce this burden in the future.

Finally, dynamic binding can be a curse as well as a benefit. Sometimes late binding slows execution, and this can be exacerbated by message passing. Late binding places additional responsibility on run-time software support to identify the appropriate properties (variables and functions) to be obtained. Message passing can be slower than function calls or variable fetching. Thus execution speed of a simulation can be decreased beyond acceptable limits. However, many object-oriented programming languages are hybrid combinations of procedural approaches combined with the object orientation and retain some of the efficiencies of traditional programming language implementations.

9 POTENTIAL FOR OBJECT-ORIENTED SIMULATION

While object-oriented simulation technology is just beginning to flourish, its potential is very great. At a minimum, it can be expected that "objectness" will be added to existing simulation languages to some extent as a means of offering extensibility. Most simulation languages have some form of procedural extension via programming. By adding object-oriented facilities, these languages can provide data abstraction so that users can add their own objects to existing simulation procedures.

There are four areas in which the object orientation has special potential. First is in the area of graphical representation. Objects in an object-oriented simulation tend to represent "real-world" entities and they "encapsulate" these real-world behaviors. An object-oriented simulation model has a close correspondence to what one might "see." Hence, a graphical representation can have an almost one-to-one correspondence with the objects in the simulation. Furthermore, during execution such a correspondence can produce a very convenient basis for animation. Because the object orientation requires the user to design object classes, the design of graphical icons or sprites corresponding to those objects can be a "property" of the class. When the simulation executes, the execution may be directly reflected in an animation.

A second area where the object orientation has special potential in simulation is in the combination of simulation and artificial intelligence. Objects encapsulate their functionality and that functionality could include "intelligence." To the extent that programs can exhibit learning and adaptability, objects can perform the same tasks. A resource in a simulation language could learn to determine the importance of performing various tasks in providing service and eventually create an algorithm or neural net or something that allows it "intelligence" in what it does. The encapsulation of objects encourages exploration of these properties in the design of simulations.

A third area where the object orientation is being actively explored is in the parallel execution of simulations. Again, because of the encapsulation of information needed for an object, individual objects could be assigned their own processors to execute their behavior. Dividing a system up into objects generally means that those objects behave somewhat independently. It is this independence among objects that gives rise to the potential for parallel execution. The speed of a simulation could be greatly accelerated by exploiting such parallelism. Research and development is currently underway to formalize this approach (e.g., Sim++).

Finally, the notion that users can build their own simulation elements and then use those elements gives rise to the possibility of simulation software engineering. In programming, a systems programmer provides tools for applications programmers. Through object-oriented technology, a new category of simulation professionals may emerge that develop simulation tools for simulation applications engineers. Currently such expertise only resides at simulation software houses that have access to the internal work-

ings of simulation technology. Object-oriented simulation languages would make that capability generally available. One could envision companies with simulation software engineers that develop objects and their functionality for use by others who require specific simulation modeling facilities. This would provide the opportunity for greater "vertical-specific" simulation facilities such as those now gaining popularity in the areas of manufacturing and materials handling. With object-oriented simulation, companies and individuals would have the opportunity to develop their own simulation technology.

REFERENCES

Belanger, R., and A. Mullarney. 1990. *MODSIM II tutorial*, Revision 8. La Jolla, California: CACI Products Company.

Birtwistle, G.M., O.J. Dahl, B. Myhrhaug, and K. Nygaard. 1979. *Simula begin*, Second Edition. Lund, Sweden: Studentlitteratur.

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

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

Lomow, G., and D. Baezner. 1990. A tutorial introduction to object-oriented simulation and Sim++. In *Proceedings of the 1990 Winter Simulation Conference*, eds. O. Balci, R.P. Sadowski, and R.E. Nance, 149–153. Institute of Electrical and Electronics Engineers, San Francisco, California.

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.

Russell, E.C. 1989. *Building simulation models with SIMSCRIPT II.5*. La Jolla, California: CACI Products Company.

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

Schruben, L. 1991. *SIGMA: a graphical simulation system*. South San Francisco, California: Scientific Press.

Winblad, A.L., S.D. Edwards, and D.R. King. 1990. *Object-oriented software*. Reading, Massachusetts: Addison-Wesley.

Wirfs-Brock, R., B. Wilkerson, and L. Wiener. 1990. *Designing object-oriented software*. Englewood, New Jersey: Prentice Hall.

AUTHOR BIOGRAPHIES

DIANE P. BISCHAK is an assistant professor in the Department of Industrial Engineering at North Carolina State University. She received the B. Music and the M.S. and Ph.D. degrees in Industrial and Operations Engineering at the University of Michigan. Her research interests are the statistical analysis of simulation output and the use of object-oriented languages in the simulation and modeling of material handling systems. She is a member of the TIMS College on Simulation and a member of ORSA, IIE, and ACM.

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. degrees at Purdue University. He is the Modeling Area Editor for the *Transactions on Modeling and Computer Simulation*, an Associate Editor for *Management Science*, and an Associate Editor for *Simulation*. He has served as *Proceedings* Editor and Program Chair for the Winter Simulation Conference and is the TIMS College on Simulation member of the Board of Directors of WSC.