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

AN APPROACH TO OBJECT-ORIENTED DISCRETE-EVENT SIMULATION OF MANUFACTURING SYSTEMS

John P. Shewchuk
Tien-Chien Chang

School of Industrial Engineering
Purdue University
West Lafayette, IN 47907, U.S.A.

ABSTRACT

Object-oriented programming is a promising approach for obtaining both modelling flexibility and ease-of-use in simulation software. In the object-oriented approach, the simulation modeler starts with an *object class library* containing the types of objects relevant to his/her application. The modelling task consists of creating, manipulating and destroying objects, using established methods, so as to emulate the operation of the real system. The objects themselves take care of the mechanics of data manipulation and storage. This approach provides a natural framework for developing simulation models using pre-defined object classes, while the concepts of *inheritance* and *sub-classes* can be used to create new objects for specific applications. This paper is concerned with object-oriented discrete-event simulation of manufacturing systems. A hierarchical structure of object classes is proposed, consisting of three class libraries: base classes, Simulation Support (SS) object classes, and Manufacturing Systems Simulation (MSS) object classes. The definition of each class and how the class objects interact with one another are discussed. Finally, an example of a discrete-event simulation model developed using the object classes is presented. The example illustrates the basic nature, merits and drawbacks of this approach.

1 INTRODUCTION

Three approaches have traditionally been available for developing simulation software and performing discrete simulation. *General Purpose Simulation Languages* (SIMSCRIPT, SIMAN, SLAM event modelling language) provide the basic mechanisms necessary for implementing simulations. Simulation models are developed using an *event orientation*: system behavior is described in terms of state changes (events), using the syntax of the particular simulation language (similar to writing a FORTRAN program). *Simulation Modelling Facilities* (GPSS, SLAM network language) emphasize the modelling task. The *network* or *process orientation* approach used consists of defining the time-dependent behavior of the system in terms of entities flowing through pre-defined function blocks (similar to creating a flowchart). *Data-Driven Simulators*, used for specific simulation applications, consist of pre-defined objects capable of performing set operations. The modelling task consists of selecting those objects which best represent the real system elements, and providing the required data for each object.

Two major goals for any simulation approach are *generality* (modelling flexibility) and *ease of use*. These goals have proven difficult to achieve simultaneously. General-purpose simulation languages are very flexible and can be used to simulate practically anything, but are difficult and time-consuming to learn and require a good degree of programming skill. Data-driven simulators are at the other end of the spectrum: they are very easy to use but are restricted to a narrow range of applications and thus offer limited flexibility. The major challenge in simulation language development has been to determine what general programming facilities are required and what pre-defined objects are needed. The problem with this approach is that there is a seemingly endless variety of objects which may be encountered in simulation applications: if required objects are not defined, the usefulness of the language to us is questionable. Roberts and Heim (1988) summarize this idea: "As machines and services increase in their ability to behave in a complex fashion, simulation languages which restrict modelling freedom will become increasingly obsolete".

An alternative approach to discrete simulation, which has the potential to address both goals simultaneously, is the *object-oriented* approach. Object-oriented programming is gaining rapid acceptance both in academia and industry as a fundamental new approach to modelling, designing, and implementing software systems and applications. In this approach, problems are envisaged in terms of *objects* - what objects are used, what types of information they store, and what operations they must be able to perform. This approach is particularly well suited to the problem of discrete simulation, as the operation of any system can basically be viewed as the interaction of objects over time.

With data-oriented (traditional) general-purpose simulation languages, the emphasis is on providing the mechanisms for simulation execution and commonly-used data structures and functions. The modelling task consists of defining data structures (arrays, files) and transforming the interaction of real-world objects to operations on data within these structures. The goal of the object-oriented approach is to eliminate this transformation process, thereby freeing the user to concentrate on the modelling process only. Once suitable *object class libraries* have been developed, the modelling task becomes straightforward: the objective is simply to manipulate objects so as to emulate the manner in which the real-world objects interact. The mechanics of data manipulation and storage are taken care of by the objects themselves: these tasks remain transparent to the user. The result is that the simulation modelling task is easier to perform.

The goal of generality is addressed through the concept of *inheritance*, a mechanism by which existing objects can easily and systematically be modified to provide new objects. The object class library then provides the general facilities (object classes) required for simulation modelling: through inheritance, existing classes can be modified to create the "custom" objects required for various applications.

In this paper, object class libraries are developed for the purpose of performing discrete simulations of manufacturing systems. Though object-oriented programming techniques can be used for developing event modelling, network modelling, or data-driven simulation software, the focus here is on the event orientation. This is because the event model is both the easiest to implement in object-oriented fashion and probably has the most to benefit from this approach. The C++ language (Stroustrup, 1987) was chosen for the implementation because it has fast execution times and is highly portable: C++ code is preprocessed to give C object code, which can then be linked and executed like ordinary C programs.

2 OBJECT CLASS HIERARCHY

In any type of computer modelling, similar problems can be grouped together so that they can re-use common code. The scope of problems which can then be handled by that common code can then be referred to as a *problem domain*. The problem domain can be as large or small as desired. If the problem domain is too small, code reusability is limited. If the problem domain is too large, the code may cover such a wide variety of specific problems that it becomes unwieldy and difficult to use.

The object-oriented approach involves considering problem domains in hierarchical fashion. Problems are considered in the most general sense at the top of this hierarchy and domains become more specific at lower levels. Problems are described in terms of the objects encountered and the ways in which the objects interact with each other. The objects themselves are then defined by developing *object classes*. An object class is a representation of the form and function associated with a particular type of object. A class is described in terms of the types of data which are used to define objects of that class (*private data definitions*) and the ways in which objects of that class interact with the outside world (*class methods*). Each class can be thought of as a template for creating new instances of objects of that class.

Classes found at any particular level in the hierarchy will be suitable for use at that level or any lower level. At lower levels, classes can either be used directly, used as the basis for a new class (i.e. singular or multiple inheritance), or used to help define a new class (class instance forms private data element of new class).

This class hierarchy is illustrated in Figure 1. At the top level of this hierarchy are general-purpose object classes which can be used for solving a great variety of problems. Such classes will be referred to here as *base classes*. Below this level are *application support* object classes. These are classes which can be re-used in a variety of application-specific problem domains. The lowest level consists of *application-specific* object classes. These classes can be re-used to solve problems within a specific application area.

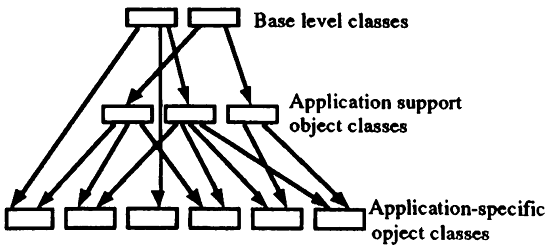
Base level classes
Application support object classes
Application-specific opject classes

Figure 1: Object class hierarchy

The object classes used in this work are developed based upon the class hierarchy of Figure 1. These object classes, and their position within the class hierarchy, are illustrated in Figure 2. The base classes are general classes and, as such, are not unique to simulation applications. At the application-support level are *Simulation Support* (SS) object classes. These are classes which may be used for any simulation program, regardless of the actual nature of the simulation problem. At the bottom level are the *Manufacturing Systems Simulation* (MSS) object classes. Details of each class are discussed in Section 4.

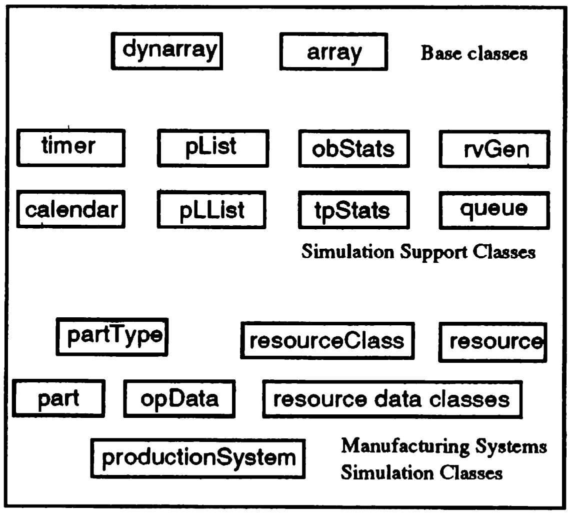

Figure 2: Simulation object class hierarchy

3 GOALS IN CLASS DEVELOPMENT

The main objective is that the simulation classes developed be highly reusable. The emphasis is upon providing a solid platform of classes which can be used as is for simple simulation models, and expanded through inheritance as model complexity increases.

The second objective is to incorporate all tasks associated with each object in the corresponding class definition. For example, statistics collection, an important part of any simulation, is usually implemented by maintaining set statistics collection routines, and specifying what data to supply to these routines, and at what times, during the simulation. The mechanics of the statistics collection tasks are independent of application, and as such it is attractive to shift the responsibility of these tasks from the simulation modeler to the objects themselves. To accomplish this, statistics collection objects can be defined, and instances of these classes can be declared to be part of the private data of any other class. The same procedure can be applied to the routines used to output statistical results. We then have the situation where each object is responsible for maintaining its own statistics during the simulation, and to obtain statistics on any object, we simply ask that object directly. This is a very natural manner in which to model and implement such tasks. Other tasks which can be handled in similar fashion include input routines and error trapping.

4 DESCRIPTION OF CLASSES

4.1 Base Classes

The base classes are used both by other object classes and by the simulation modeler (i.e. in application programs). The base classes are:

- class *array*: used for performing standard manipulation tasks on lists of values (arrays). Examples of these tasks include: sorting a list of values from largest to smallest or vice-versa, filtering out a particular value from a list of values, and ordering a list of values based upon another list.
- class *dynarray*: dynamically-sized array; an array implemented as a linked list. These objects are used extensively for tasks where an array is required, but the size of the array can change after the array is initially declared.

4.2 Simulation Support (SS) Classes

The simulation support (SS) object classes are used by other SS object classes, by MSS object classes, and in the application program. The simulation support object classes are:

- class *timer*: used for creating a system clock object for executing the simulation.
- class *calendar*: used for creating an event calendar object. The event calendar maintains event time, event code, and an object pointer (address) for each event posted to the calendar.
- classes *tpStats* and *obStats*: used for creating objects for collecting time-persistent and observation-based statistics respectively. Instances of these classes are used by *pList* objects (described below), MSS support objects (so all MSS objects generate their own statistics), and in the application program (e.g. for statistics over multiple runs).
- class *pList*: a linked list used for keeping track of objects. Object pointers (addresses) are added to the list when created and removed from the list when destroyed. To access an object, we tell the *pList* object what object number we are interested in and the list returns the object address. In conjunction with object handling, each *pList* object has a *tpStats* and

an *obStats* object as private data for maintaining quantity-on-list and time-on-list statistics respectively. Class methods are included for printing statistics to a specified output file pointer, and for printing a standard header. Objects of class *pList* are used as private data elements in objects of the class *pLList* (described below).

- class *pLList*: used for creating an object-handling object. The object handler is used to keep track of all objects in the system. It maintains a series of *pList* objects, one for each separate object category used in the simulation (part, tool, machine, part queue, etc.). The object handler is then used to track and maintain statistics on each object category. Figure 3 shows an example of a *pLList* object having two *pList* objects - one for part objects and one for machine objects.

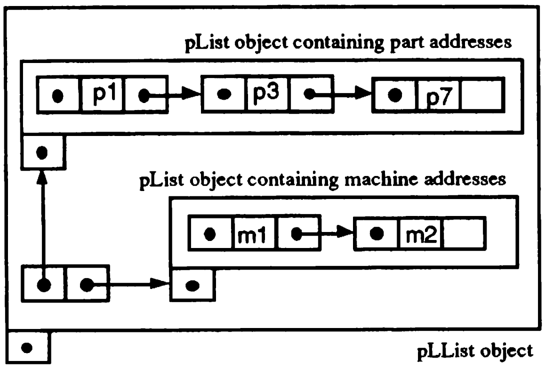
pList object containing part addresses
p1
p3
p7
pList object containing machine addresses
m1
m2
pLList object

Figure 3: Data structure for objects of class *pLList*

Class methods are included for printing statistics to a specified output file pointer, and for printing a standard header. This print method simply sends the statistics print method to the corresponding *pList* object.

- class *rvGen*: used for creating a random variate generator object. One object of this class can be used for all random variate generation required in an application program. The user specifies the number of separate random number streams required. Each stream is initialized separately can be re-initialized whenever desired. Each time the stream is sampled, the type of distribution (normal, triangular, etc.), distribution parameters, and seed reset option selection are passed as arguments, and the *rvGenerator* object returns a random variate of the corresponding distribution. If the user wanted to use a particular random value, the seed would be updated. If the user wanted to simply check what the next random variate would be, the seed would not be updated.

- class *queue*: used for maintaining an array of integer values. Items can be added to and removed from the queue according to a specified queueing discipline (e.g. FIFO), or the location in the queue can be specified. The latter approach is useful when using queue object to model objects such as tool magazines, where tools are loaded into pre-defined positions within a magazine. Queues are considered to be simulation-support object classes as each queue has *tpStats* and *obStats* objects as private data elements for maintaining queue length and time-in-queue statistics respectively. Class methods are including for printing statistics to a specified output file pointer, and for printing a standard header.

In a simulation program, a single object instance is required from classes *timer*, *calendar*, *pLList*, and *rvGen*. These objects are called Clock, EventCal, objectList, and rvGenerator respectively. Each of these objects is created in the mainline program and declared to be global so that any other class which requires use of the object (to implement one or methods) will have access to it. Such classes must then have the object declared as an external variable in the class definition. For example, the *calendar* object requires the use of the system clock, so Clock must be declared an external variable in the class definition for class *Calendar*. Note that the names of these simulation support object classes cannot be changed in the mainline program.

There are many objects methods which require access to the simulation support objects described above. For example, the method getFreeResources(int resource_class) in class *part* is used to determine which resources of a given class are available and have sufficient life remaining (finite-life resources such as tools) for the part's next operation. In order for the part object to be able to determine this, it must check all the resources of this class. This can be done by giving part objects access to the linked list where resource object addresses are maintained, i.e. the address of the objectList object. The method checkProcessTime (int resource_class) in class *part* is used to check the processing time required, at the part's next operation, using a resource of the given class. To calculate this value, the part must be able to access the rvGenerator object (as well as determine the processing time distribution type and parameters).

4.3 Manufacturing Systems Simulation (MSS) Classes

The library of classes which have been developed for simulating manufacturing systems are referred to as the Manufacturing Systems Simulation (MSS) library of object classes. Object classes have been developed for modelling each of the two major types of entities found - parts and resources (queues are modelled at the simulation support level). A special object class was also developed for handling the production aspects of the system. MSS classes are used by the application program and by other MSS object classes. The MSS classes are defined as follows:

1. *Part Object Classes* are used for modelling parts in the system. Part objects must know their process routing (what resources they require for processing and in what sequence), how to determine the processing time with each resource required, and how to seize and free such resources. The object classes used for modelling parts are as follows:

- class *partType*: objects of this class contain the data which is common to all parts of a given type: part type name (description), part type number, number of operations required for processing, and the objectList list numbers where *partType* and *part* objects are stored. Because part process routing data requirements can vary greatly from one application to another, routing data is not included in *partType* objects themselves. Instead, separate routing data objects are used, and *partType* objects contain pointers to these objects. Through this linkage, *partType* objects know their routing data, while the physical class separation of routing data from part type data facilitates the development of different forms of routing data classes for various applications.
- class *opData*: the default routing data class (in conjunction with *partType* objects) used to specify part process routing data. One *opData* object is used to describe each operation of a given part type. Each operation is defined in terms of the quantity of resource classes required for processing, what these classes are (e.g. tools, machines, fixtures), the quantity of resource types within each class which can be used and what these resource types are (e.g. alternate machine routings), and the processing time distribution type, parameters, and random number stream to use in generating process time values. The use of a separate *opData* object for each operation of each part type allows for great flexibility in describing part process routings.
- class *part*: objects of this class are used to represent the actual parts in the system. Each part object contains as private data its part type, what operation number it is currently on, which resources it currently has seized (if any), what queue number it is currently in (if any), and a data link (pointer) to the *partType* object defining this part.

Figure 4 depicts the objects required and data links for a system having two part types and three parts.

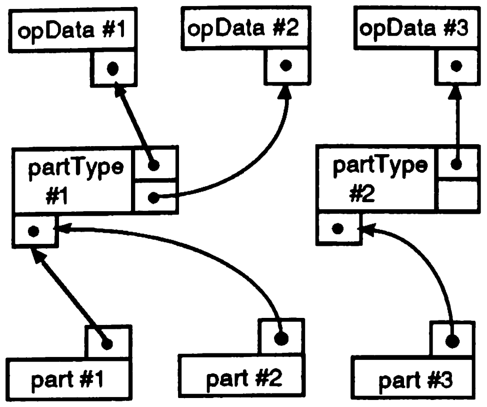

Figure 4: Data links for objects used in modelling parts

2. *Resource Object Classes* are used for modelling resources used in manufacturing systems, such as machines, tools, etc. Three object classes are used:

- class *resourceClass*: objects of this class contain the data which is common to all resources of a given class: resource class name (description), how many resources of this class are allowed in the system, and the objectList list numbers where *resourceClass* and related *resource* objects are stored. Because the data requirements for one resource class may be quite different from those of another (e.g. data for an AGV may contain AGV speed, battery recharge time, etc., - this data does not similarly apply to a resource class such as tool), resource class data is not included in *resourceClass* objects themselves. Instead, separate resource type data objects are used, and *resourceClass* objects maintain pointers to these objects. This physical class separation of *resourceClass* and resource type objects facilitates the development of different types of resource classes in an application.

  Objects of class *resourceClass* keep a list of utilization statistics for resources of that class which were used and then destroyed (e.g. tool which was removed after cutting edge became dull). Average utilization statistics can be obtained for all resources of that class, or for all resources of a certain type within that class.
- resource type classes: objects of these classes contain the data which defines a particular type of a given resource. A different resource type class is defined for each resource class, e.g. class *machineType* for machines, class *toolType* for tools, etc. The resource type class definition will contain private data describing the distribution type and parameters used for determining resource life (finite-life resources such as tools) and time-between-failures (machines).

- class *resource*: objects of this class are used to represent the actual resources in the system. Each resource object contains as private data its resource class, resource type, useable life remaining, a flag indicating whether busy or idle, a pointer indicating where it is located in the system (with respect to other objects), and pointers to the *resourceClass* and resource type objects which define this resource. In addition, each resource object contains *tpStats* and *obStats* objects, as private data, so they can maintain their own utilization statistics.

For each resource class, there will be as many resource type objects defined as there are types of that resource. (e.g. five tool types - five *toolType* objects are required). Figure 5 illustrates the objects required and data links for a system containing one resource class (for tools), two resource types within this class, and three resources.

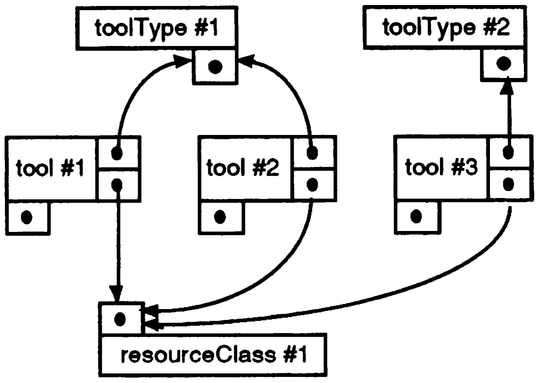

Figure 5: Data links for objects used in modelling resources

3. class *productionSystem*: an object of this class is used to store and manipulate production-related data which does not fit naturally within the domain of any other object class. This data includes the shop calendar, production shift length, and data related to the method by which parts are introduced into the system. Parts may be introduced either in batches or continuously. For batch production, the *productionSystem* object keeps track of the batch mix, expected completion time per batch, and which batches are completed on time and which are late. For continuous production, the data consists of the rate at which parts are introduced into the system (interarrival time distribution). Batch or continuous production is specified for each part type in the system. One *productionSystem* object is used for each simulation model. All resource and parts objects are both created and destroyed by *productionSystem* class methods: the *productionSystem* object maintains a pointer to the objectList object for this task. The *productionSystem* object maintains *tpStats* and *obStats* objects for maintaining statistics on production-related variables such as batch completion time, number of late parts per batch, and the quantity of resources and quantity of duplicate resources of any class used during a production period.

5 OBJECT-ORIENTED DISCRETE-EVENT SIMULATION

Once the base, SS, and MSS class libraries have been developed, we can use them to write object-oriented discrete-event simulation programs. The first step in program development is to create the objects required for the simulation. This will include such application-support objects as the system clock and event calendar, as well as application-specific objects such as machines and part queues. The framework then used for executing an object-oriented discrete-event simulation is the same as that used in a data-oriented (traditional) simulation. The simulation is driven by pulling events off the calendar, advancing system time, and executing the events. Events are again used to define system interactions, but the emphasis is now on defining object interactions rather than performing data manipulation, as discussed previously.

The shift in emphasis from data manipulation to object interactions produces a more natural framework for discrete-event simulation. For example, consider the task of posting events to the event calendar. The usual approach is to post the attribute (i.e. dynamic) data representing the state of the system to the event calendar along with the event time and event code, a practice which encourages us to think in terms of data manipulation. In the object-oriented approach, only the identification (address) of the object upon which the event is based (e.g. machine for end-of-machine-service event) needs to be posted along with the event time and event code. All data defining the state of the system is maintained with the objects themselves.

6 IMPLEMENTATION EXAMPLE

An artificially-small example shall be used to illustrate the concepts of object-oriented discrete-event simulation, using the classes described in Section 4. The example consists of a flow line having M unique machines. An input queue precedes the first machine; buffer queues separate the remaining machines in the line. A total of N part types are produced; parts arrive continuously over time according to user-specified interarrival time distributions. FIFO (first-in, first-out) queueing discipline is used for part queues, which have infinite capacity. The system is described in terms of the following input variables:

$maxops_i$	number of operations for part type i, $1 \le i \le N$.
$pparm1_{ij}$	Process time distribution parameter #1 for op. j of part type i, $1 \le i \le N$, $1 \le j \le maxops_i$ (similarly define $pparm2_{ij}$ and $pparm3_{ij}$).
$aparm1_i$	Arrival time distribution parameter #1 for part type i (similarly define $aparm2_i$ and $aparm3_i$).
pdist	Distribution type to use for process time (1-normal, 2-triangular, etc.).
pstrm	Random number stream to use for process time.
adist	Distribution type to use for interarrival time.
astrm	Random number stream to use for interarrival time.

For simplicity, machine setups, breakdowns and material handling are ignored. The system can be described in terms of two events:

1. *start_next_operation*: a part has just arrived at a machine for an operation. If the machine is free, processing of the part begins; otherwise the part enters the machine's part queue. If the part is new, a request for the next part of that type is generated.
2. *end_of_machine_service*: a machine has just finished an operation on a part. That part is moved to the next operation on its routing. If there are more parts waiting in the machine's part queue, processing on one of these parts begins. Otherwise, the machine is set idle.

To model this system using the classes previously developed, the simulation modeler needs to know *what* tasks objects of each class can perform, and *how* to get the objects to perform these tasks. This information is maintained in *class definition* files. These files define the private data and messages which can be sent to objects of each class, and describe the action which results (i.e. operation of method invoked) when each message is sent. The methods themselves are maintained separately (*class implementation* file), the idea being that the modeler only requires access to the message definitions.

Though the definitions for the classes developed in Section 4 are not presented here, the program can still be understood, as the messages used in this example are by and large self-explanatory (this is of course highly desirable in object-oriented programming as it makes classes easy to use).

The main program for this example is shown in Figure 6. The include file "mdecls.h" contains i) #include statements for all SS and MSS class definition files, ii) object class list assignments (#define PTYPE 1, #define PART 2, etc.), iii) declaration of the required SS objects (timer* Clock, calendar* EventCal, etc.), and iv) declaration of pointers used for object manipulation in the program (part* Part, resource* Machine, etc.).

Following creation of local data input variables and data input, the first task in the main program is to create the SS objects. These are the simulation support objects (Clock, objectList, EventCal and rvGenerator). Following this, the static manufacturing objects (*partType*, *resouceClass*, etc.) are created. Note that each time an MSS object is created, its address must be added to the objectList object. The initial part arrival events are posted to the event calendar by the method ProdSystem->setArrivalRate(...). Whenever this event is pulled off of the event calendar, the method ProdSystem->newPartArrival() creates a part of the required type, adds its address to the objectList object, schedules the arrival of the next part of that type, and returns the parts address. A while loop is used to drive the simulation, which continues until either the event calendar is empty or the simulation run length has been reached. Following simulation, output statistics on part throughput, machine utilization, and queue statistics are printed.

The event functions *start_next_operation* and *end_of_machine_service* are shown in Figures 7 and 8 respectively.

7 DISCUSSION OF EXAMPLE

Several general observations can be made regarding the example:

1. The code is generally easy to read. Object manipulation using well-designed methods and messages results in a pseudo-English appearance to the code. Event logic is very easy to follow: few supporting comments are required for the reader to figure out what is happening.
2. Statistics collection and other file manipulation tasks are not performed explicitly. As previously described, the objects are manipulated to emulate the actual system operation and the statistics are taken care of by the objects.
3. Output of performance parameters is very straightforward. Statistics are obtained by sending the desired messages to the appropriate objects, e.g. part queues report their average queue length. Quantity and time-in-system statistics for any object class are maintained by the objectList: these are used to obtain average quantity of parts in the system and the average time each part was in the

```
#include "mdecls.h"

main()
{
< Declare input variables and initialize >

/****STATIC OBJECT CREATION****/

// 1. Create simulation support objects:
Clock        = new timer();
objectList   = new pLList(6);
EventCal     = new calendar();
rvGenerator  = new rvGen(4);

//2. partType and associated routing data objects:
for(int i=0;i<N;i++) {
   PartType = new partType(i+1,maxops[i]);
   PartType->setLists(PART_TYPE,PART,QUEUE);
   for(int j=0;j<maxops[i];j++) {
      RData = new opData(1);
      RData->setClassData(MACH,0);
      RData->setCandidateResourceQuantity(MACH,1);
      RData->setResourceType(MACH,1,machs[i][j]);
      RData->setProcess(MACH,1,pparm1[i][j],
               pparm2[i][j],pparm3[i][j],pdist,pstrm);

      PartType->setOpData(j+1,RoutingData);
   }
   objectList->add(PTYPE,PartType);
}

//3. resource class, type and resource objects:
MachineClass = new resourceClass(MTYPE,MACH);
objectList->add(RCLASS,MachineClass);

for(i=0;i<M;i++)
  objectList->add(MTYPE,NULL); //no machine type data

for(i=0;i<M;i++) {
   Machine = new  resource(RCLASS,MACH,i+1);
   objectList->add(MACH,Machine);
 }

//4. Part queue objects:
for(i=0;i<M;i++) {
   PartQueue = new queue("PartQueue",pqcap,PQ);
   PartQueue->setQDiscipline(FIFO);
   objectList->add(PQ,PartQueue);
 }

//5. production control object:
ProdSystem = new productionSystem(N,1);
ProdSystem->setPartLists(PART_TYPE,PART);
for(i=1;i<=N;i++) {
   ProdSystem->defineProductionType(i,CONT,1);
   ProdSystem->setArrivalRate(i,aparm1[i],aparm2[i],
                                  aparm3[i],adist,astrm);
 }

/****SIMULATION****/

float etime; int ecode; void* eobj; event* evntPtr;
int runOK = 1;

while(runOK)
{
  if(EventCal->numberOfEvents()==0)
    runOK = 0;       //simulation is complete.
  else {
    //Pull top event off of calendar:
    evntPtr  = EventCal->get();
    etime    = evntPtr->event_time;   //event time
    ecode    = evntPtr->event_code;   //event code
    eobj     = evntPtr->obj_id;       //object id

    if(etime>runlength)
      runOK = 0;       //simulation is complete.
    else {
      Clock->setTime(etime); //advance system clock

      //Execute the event:
      switch(ecode) {
      case 1: if(eobj == NULL)
                 Part = ProdSystem->newPartArrival();
              else
                 Part = (part*)eobj;
              start_next_operation(Part);
              break;
      case 2: Machine = (resource*)eobj;
              end_of_machine_service(Machine);
              break;     }
      }
    }
}

/****OUTPUT RESULTS****/
char name[20];
//part throughput:
objectList->printStatsHeader(stdout);
objectList->printStats(stdout,PART,"Parts");
objectList->endPrintStats(stdout);

//machine utilization:
MachineClass->printUtilizationHeader(stdout);
for(i=1;i<=3;i++) {
   sprintf(name,"Machine %d",i);
   MachineClass->printStats(stdout,&name[0],i);
 }
MachineClass->endPrintStats(stdout);

//queue statistics:
for(i=1;i<=3;i++) {
   PartQueue = (queue*)objectList->get(PQ,i);
   sprintf(name,"part queue %d",i);
   PartQueue->printStats(stdout,&name[0]);
 }
PartQueue->endPrintStats(stdout);

/* end main */
}
```

Figure 6: Example program

```
void start_next_operation(part* Part) {
 int resourcesfree  = Part->moveToNextOp();
 int mno            = Part->getResourceType(MACH);
 PartType           = Part->getPartType();

 if(resourcesfree) {
  //processing of Part can continue.
  Machine    = (resource*)objectList->get(MACH,mno);
  float svctime = Part->getProcessTime(MACH);
  Part->seizeResource(MACH,Machine);
  Part->setBusy(svctime,2,Machine); //posts event 2
 }
 else
  Part->goIntoQueue(mno); //queue Part.

/* end start_next_operation */
}
```

Figure 7: start_next_operation event

```
void end_of_machine_service(resource* Machine) {
 Part       = (part*)Machine->seizedBy();
 PartType   = Part->getPartType();
 int mno    = Machine->getNumber();
 PartQueue  = (queue*)objectList->get(PQ,mno);

 Part->endOfOperation(); //frees machine.

 //Determine disposition of part :
 if(Part->getCurrentOp()<PartType->maxOps())
  EventCal->add(0.0,1,Part);   //move part to next op.
 else
  ProdSystem->removeFinishedPart(Part);

 //Determine disposition of machine:
 if(PartQueue->numberOfItems()>0) {
  int pno       = PartQueue->remove();
  Part          = (part*)objectList->get(PART,pno);
  float svctime = Part->getProcessTime(MACH);
  Part->seizeResource(MACH,Machine);
  Part->setBusy(svctime,2,Machine);
 }

/* end end_of_machine_service */
}
```

Figure 8: end_of_machine_service event

system. These statistics can also be obtained for resources which flow in and out of the system over time (e.g. tools).

The above observations all constitute advantages of the object-oriented approach. The advantages become much more pronounced when complex object interactions and decision logic must be modelled. For example, consider a system where part routings are defined in terms of both tools and machines, and where tools have finite life and alternate machine routings exist. As the same objects and methods can be used, this system is no more difficult to model using the object-oriented approach, only more time-consuming.

Of course, successful object-oriented programming applications are contingent upon one very big factor: how easy it is to become familiar with and use the object classes required for the application. This becomes a question of how "good" the object classes are in terms of faithfully representing the actual objects and object interactions. If the representation is authentic, the classes are likely to be easy to use, as objects can then be manipulated in what seems to be the most "natural" or "obvious" manner. The modelling task can then be approached more from an application specialist's point-of-view (here, manufacturing systems engineer) and less from a computer programmer's perspective. Consequently, classes become more likely to be reused in both applications and in the development of new classes.

8 CONCLUSIONS

In this paper, we have shown how the object-oriented approach can be applied to discrete-event simulation, and, in particular, discrete-event simulation of manufacturing systems. Various object classes were developed, in hierarchical fashion, for these tasks. The classes are relatively simple and straightforward to use, as demonstrated by the implementation example. They provide for a great deal of modelling flexibility, but not in the usual sense of the word. There exists a fixed domain of manufacturing systems simulation problems which can be modelled using these classes alone. This is not by consequence, however, but by design: the emphasis in class development was to keep the classes simple and "generic" in nature. Flexibility then results from exploiting this characteristic to create, through inheritance, whatever application-specific object classes we require. For example, class *toolMagazine* from class *queue*, or class *transporter* from class *resource* and then class *AGV* from class *transporter.*

Much class development work can still be done, however. As creation of sub-classes through inheritance is expected to be a common process, a *classCreation* object class would prove very handy. At the simulation-support level, an *eventDriver* object could be employed to drive the simulation, i.e. pull events off the calendar, execute the events, and determine when to end the simulation. An object for performing and controlling multiple replications of a given system, and maintaining statistics over such replications, would also be very useful.

The task of designing object classes which can be used successfully (both directly in applications and as the base for new classes) can prove to be very difficult. During development of the classes presented here, many object classes were re-defined, some classes became unnecessary or inappropriate and were abandoned, and some tasks gave rise to new classes. It is felt that class design and development will always be a very difficult and time-consuming task. It seems that explaining what happens during an event is one thing: determining which tasks are the responsibility of which objects can be quite another. Nonetheless, it is believed that the object-oriented approach is here to stay. Simulation will continue to be an attractive field for this approach to problem solving, software development, and program maintenance.

ACKNOWLEDGEMENTS

This research is part of an M.S. thesis developed under the guidance of Dr. T. C. Chang, and was partially supported by a National Science Foundation Presidential Young Investigator Award no. DDM-8552699 to Dr. Chang. Special thanks are also due to Prof. J. Wilson, Associate Professor of Industrial Engineering, Purdue University, West Lafayette, IN 47907.

REFERENCES

Cox, B.J. 1987. *Object-Oriented Programming: An Evolutionary Approach.* Addison-Wesley.

Duff, C.B. 1986. Designing an Efficient Language. *BYTE*, Vol. 11, No. 8, 211-214.

Eckel, Bruce. 1989. *Using C++.* Berkeley: McGraw Hill.

Pritsker, A.A.B. 1984. Introduction to Simulation and SLAM. Third edition. New York: Halsted Press.

Roberts, Stephen J. and Heim, Joe. 1988. A Perspective on Object-Oriented Simulation. In *Proceedings of the 1988 Winter Simulation Conference*, eds. M. Abrahms, P. Haigh, and J. Comfort, 277-281. Institute of Electrical and Electronics Engineers, San Francisco, California.

Shewchuk, John P. 1990. Multiple-Resource-Based Part Scheduling and Tool Allocation Heuristics for Automated Manufacturing Systems. Master's Thesis, School of Industrial Engineering, Purdue University, West Lafayette, Indiana.

Stroustrup, Bjarne. 1987. *The C++ Programming Language.* Addison-Wesley.

Wiener, Richard S. and Pinson, Lewis J. 1988. *An Introduction to Object Oriented Programming and C++.* Addison-Wesley.

AUTHOR BIOGRAPHIES

JOHN P. SHEWCHUK is a Ph.D. student in Industrial Engineering at Purdue University. He received is B.S. in Mechanical Engineering from the University of Manitoba in 1984, and his M.S.I.E. degree from Purdue University in 1990. His research and consulting interests include manufacturing systems simulation, production control, cellular manufacturing and FMS, and quality control.

TIEN-CHIEN CHANG is a professor of Industrial Engineering at Purdue University. His research interest is in computer-aided manufacturing. He has authored and co-authored four books and many articles in the CAD and CAM areas.