

SIMLIB:
A CLASS LIBRARY FOR
OBJECT-ORIENTED SIMULATION

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER
ENGINEERING AND INFORMATION SCIENCE
AND THE INSTITUTE OF ENGINEERING AND SCIENCES
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Oğuz Işıkli
July, 1993

QA
76.64
.I85
1993

SIMLIB: A CLASS LIBRARY FOR OBJECT-ORIENTED SIMULATION

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER
ENGINEERING AND INFORMATION SCIENCE
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By

Oğuz Işıklı

July, 1993

Oğuz Işıklı


tarafından imzalandı.

QA
76.64
.125
1993

B. 122

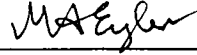
© Copyright 1993
by
Oğuz Işıklı and Bilkent University

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.



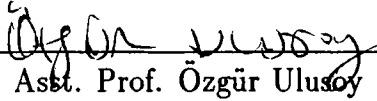
Assoc. Prof. Varol Akman (Co-advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.



Prof. Akif Eyler (Co-advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.



Asst. Prof. Özgür Ulusoy

Approved for the Institute of Engineering and Science:



Prof. Mehmet Baray
Director of the Institute

ABSTRACT

SIMLIB: A CLASS LIBRARY FOR OBJECT-ORIENTED SIMULATION

Oğuz Işıklı

M.S. in Computer Engineering and Information Science

Advisors: Assoc. Prof. Varol Akman and Prof. Akif Eyler

July, 1993

Simulation is one of the most widely used techniques in decision making. Mathematical modeling of a real world system is a major task of the simulation analyst. The selection of a computer language for implementing the model is also important. Recent research in this area has focused on the compatibility between simulation implementations and the object-oriented paradigm. It is the purpose of this thesis to explore the use of an object-oriented approach for the implementation of discrete event simulation applications. We present a class library which provides the skeletal elements of a simulation. The advantages and the disadvantages of the approach are discussed with the help of three prototype implementations: the single-queue/single-server system, the production-line system, and the elevator system.

Keywords: Discrete Event Simulation, Object-Oriented Programming, Object-Oriented Design, Class Libraries, C++ Programming Language, Single-Queue/Single-Server Systems, Production-Line Systems, Elevator Systems.

ÖZET

SIMLIB: NESNEYE-YÖNELİK BENZETİM İÇİN BİR SINIF KÜTÜPHANESİ

Oğuz Işıklı

Bilgisayar ve Enformatik Mühendisliği, Yüksek Lisans
Danışmanlar: Doç. Dr. Varol Akman ve Prof. Dr. Akif Eyler
Temmuz 1993

Benzetim, karar verme sistemlerinde çok yaygın şekilde kullanılan tekniklerden biridir. Bir gerçel dünya sisteminin matematiksel olarak modellenmesi, benzetim analistinin temel bir görevidir. Model gerçekleştiriminde hangi bilgisayar programlama dilinin kullanılacağı da önemli bir konudur. Bu alandaki son araştırmalar, benzetim uygulamaları ile nesneye yönelik programlama arasındaki yakın benzerlikler üzerinde yoğunlaşmıştır. Bu tezin amacı, kesikli-olay benzetim uygulamalarında nesneye yönelik yaklaşımın kullanımını araştırmaktır. Tezde, bir benzetim programının temel bileşenlerini sağlayan bir sınıf kütüphanesi tanıtılmaktadır. Yaklaşımın avantaj ve dezavantajları üç prototip uygulamanın yardımıyla tartışılmaktadır: tek-kuyruk/tek-işgören sistemi, üretim-hattı sistemi ve asansör sistemi.

Anahtar Sözcükler: Kesikli-Olay Benzetimi, Nesneye-Yönelik Programlama, Nesneye-Yönelik Tasarım, Sınıf Kütüphaneleri, C++ Programlama Dili, Tek-Kuyruk/Tek-İşgören Sistemleri, Üretim-Hattı Sistemleri, Asansör Sistemleri.

To my parents and friends

ACKNOWLEDGMENTS

I would like to thank my co-advisors Assoc. Prof. Varol Akman and Prof. Akif Eyler (Department of Industrial Engineering) who have provided a pleasant research environment and motivating support during this study.

I would also like to thank Prof. Mehmet Baray and the rest of the faculty of the Department of Computer Engineering and Information Science for the academic environment they have created.

Finally, I would like to thank my family, my friends, and everybody who has in some way contributed to this study by lending moral support.

Contents

1	Introduction	1
2	Object-oriented simulation	4
2.1	The object-oriented paradigm	4
2.2	Basic concepts of computer simulation	6
2.3	OOP and simulation code development	9
2.4	The use of a class library	11
2.5	Previous and related work	12
3	Design of the class library	15
3.1	Object-oriented design of classes	15
3.1.1	The use of prototypes	16
3.1.2	Class definitions	18
3.2	General form of a simulation application	20
3.2.1	Construction of a model	20
3.2.2	Running the model	24
3.3	Scheduling and execution of events	27

3.4 Mapping a model to a program	30
4 Implementation	32
4.1 Definition of classes	32
4.1.1 The SIMOBJECT class	33
4.1.2 The NODE class	34
4.1.3 The COLLECTION class	35
4.1.4 The QUEUE class	36
4.1.5 The BUFFER class	37
4.1.6 The SOURCE class	37
4.1.7 The SINK class	38
4.1.8 The SERVER class	39
4.1.9 The CARRIER class	40
4.1.10 The ENTITY class	41
4.1.11 The DISTRIBUTION class	42
4.1.12 The SIMULATION class	43
4.1.13 The SIMEVENT class	44
4.1.14 The STABLEEVENT class	44
4.1.15 The MOVINGEVENT class	46
4.1.16 The EVENTLIST class	47
4.2 Communication between objects	47
4.3 User-defined behavior	51

CONTENTS

x

4.4	System services	56
5	Three example systems	59
5.1	The single-queue/single-server example	59
5.2	The production-line example	61
5.3	The elevator example	63
6	Conclusion	71
A	Class declarations in SIMLIB	75

List of Figures

3.1	The hierarchy of system classes in SIMLIB	20
3.2	The definition of SERVER	22
3.3	Constructor of SERVER object for the no-crash case	23
3.4	Constructor of SERVER object for the crash case	23
3.5	Object declarations of the single-queue/single-server application	24
3.6	Object links for the single-queue/single-server application	24
3.7	The definition of SIMULATION	25
3.8	An example <i>ShouldStop</i> method	26
3.9	Sample output from the method <i>Done</i>	27
3.10	Class definitions for the event classes	28
3.11	A sample <i>Execute</i> method	30
3.12	The main body for a derived SIMULATION object	31
4.1	<i>Execute</i> method of ENTITYINTOSYSTEM	49
4.2	<i>Execute</i> method of ENTITYINTOQUEUE	50
4.3	<i>Execute</i> method of ENTITYFROMQUEUE	50
4.4	<i>Execute</i> method of ENTITYINTOSERVICE	51

4.5	<i>Execute</i> method of ENTITYFROMSERVICE	52
4.6	<i>Execute</i> method of ENTITYFROMSYSTEM	52
4.7	A subsystem with a decision node	53
4.8	A possible definition for DECISIONNODE	54
4.9	Method definitions for DECISIONNODE	55
4.10	Method definitions for event class ENTITYFORDECISION	56
5.1	The single-queue/single-server model	60
5.2	The code of the single-queue/single-server example	62
5.3	The production-line model with two servers	63
5.4	The code of the production-line example	64
5.5	Model of the elevator system	66
5.6	The <i>Execute</i> method of ELEVENDSERVICE	69
5.7	The <i>Execute</i> method of NEWENTITYTOELEVATOR	70

List of Tables

3.1	System classes in SIMLIB	19
3.2	Auxiliary classes in SIMLIB	19
3.3	List of event classes provided in SIMLIB	29
4.1	Return conditions for <i>CanSend</i> and <i>CanReceive</i>	48
6.1	Comparison of code length (number of lines) of the three proto- types implemented with and without SIMLIB	72

Chapter 1

Introduction

Simulation is one of the most widely used techniques in decision making. Mathematical modeling of a real world system is a major task of the simulation analyst. The selection of a computer language for implementing the model is also important.

Recent research in this area has focused on the compatibility between simulation implementations and the object-oriented paradigm [6]. The basic idea is to model the components of a system as objects and to define the behavior of the components as methods attached to these objects. Objects are treated as “black boxes,” encapsulating code and data. They communicate with the outer world through methods. This enables the representation of any system component in a modular fashion (with the advantages of readability and maintainability).

Once a clear design of the classes involved in the model is at hand, it takes less time to map the system to be simulated to a program in an object-oriented programming (OOP) language. The inheritance mechanism of OOP aids in the development of new classes from existing ones, thus resulting in reusable code. The encapsulation principle of OOP, combined with the use of abstract data types, enhances modularity.

It is the purpose of this thesis to explore the use of an object-oriented approach for the implementation of discrete event simulation applications. The

domain of interest (arguably) covers only a small portion of the simulation applications that are developed in the “real world,” but this is adequate to present the main idea of this study. The general structure which best illustrates our interest is a production-line model employing components like queues, servers, carriers, etc. connected to each other in a linear fashion. The execution of the model can be basically stated as the traveling of entities between components to receive service.

The general components of such a system can be modeled as objects and collected in a library for future use. While the components of the library provide the basic characteristics, it is the task of the programmer to define additional behavior depending on the particular application to be developed.

In our research, we tried to extract a minimal (in some sense, canonical) set of system components to include in our simulation library—SIMLIB. Therefore, we first developed three prototype applications which may be regarded as “classical” in the area of discrete event simulation. These are (i) a single-queue/single-server system, (ii) a production-line system, and (iii) an elevator system. Our aim was to investigate the general concepts of simulation programming along with the most frequently used data structures. Essential insights resulting from this phase of the study facilitated the design of SIMLIB to a great extent.

Later, the same systems were implemented by the classes and routines provided by SIMLIB. Using our previous experience, a comparison between the two development strategies was carried out. Major aspects of the latter approach, such as the class hierarchy, the execution of events, the navigation of entities, and the integration of user-defined behavior into system behavior, were distinguished.

SIMLIB has been implemented using the C++ language [24] on SUN Workstations¹ running under the UNIX² system. C++, inheriting the portability of C, has received wide acceptance. The broad range of functionality of the language and its availability on every UNIX system have been the major reasons for us to choose C++.

¹SUN Workstation is a registered trademark of Sun Microsystems, Inc.

²UNIX is a registered trademark of AT&T Bell Laboratories.

The organization of the thesis is as follows:

- Chapter 2 states the basic notions of OOP. The general concepts of simulation are also described in this chapter. These ideas are combined to define the purpose of the research, i.e., the use of an object-oriented class library for simulation. The chapter ends by reviewing some previous related work.
- Chapter 3 deals with design issues. By “design” we simply refer to the declaration of classes, viz. definition of object types along with their attributes and methods. After listing the classes included in SIMLIB, the development of simulation applications with such a library is illustrated by example code fragments.
- Chapter 4 demonstrates the implementation of SIMLIB. The definitions of the classes are given in this chapter. The communication mechanism that is used to ease the execution of the model by providing a clear interface for the transfer of entities among objects is explained. The system services provided by SIMLIB are introduced.
- Chapter 5 details the three prototype implementations that have been developed. These—a single-queue/single-server system, a production-line system, and an elevator system—are described together with their corresponding models. This chapter aims to demonstrate the efficacy of our approach by providing a comparison between the procedural and the OOP paradigms.
- Chapter 6 concludes the thesis and discusses the advantages and the limitations of our implementation. Finally, prospects for future work are stated.

To provide the reader with the class definitions of SIMLIB, the header file of the system containing the class declarations is listed in Appendix A.

Chapter 2

Object-oriented simulation

Simulation software generally uses certain types of abstract data modules to model the components of a real world system. Examples include entities traveling through the system, queues where entities are forced to wait until the occurrence of a certain event, points where entities are serviced, etc. The implementation of these data modules with an object-oriented approach is the main purpose of our study.

In this chapter, we first provide the basic ideas of the object-oriented paradigm and the general concepts of simulation. Then we concentrate on the use of a C++-based class library to ease the task of coding simulation software. We conclude by reviewing some previous work related to our study.

2.1 The object-oriented paradigm

The concepts of OOP are having a profound impact on software engineering. Advantages of this programming methodology over traditional programming have been documented in [5] and [12]. In [12], Meyer states that

“...object oriented design may be defined as a technique which, unlike classical (functional) design, bases the modular decomposition of a software system on the classes of objects the system manipulates, not on the functions the system performs.”

Our purpose here is not to provide a complete characterization of OOP, but rather to state the general aspects of this methodology, especially focusing on the properties that make OOP a feasible candidate for simulation applications. From our point of view, the four key components of OOP in this respect are abstract data types, encapsulation, inheritance, and late binding.

Abstract data types

Data abstraction allows the programmer to create data structures which can be manipulated in the same way and with the same level of efficiency as language defined types. The key construct in implementing abstract data types in C++ is the *class*. The elements of this data structure are referred to as “member variables.” Member variables are hidden from the outside world except for some specially designated functions. Only the functions of the class and these specially permitted functions can access the data part of a class variable. Functions can be “overloaded,” that is, facing multiple definitions, a compiler selects the correct interpretation at run time.

Encapsulation

The encapsulation principle treats classes as self-contained program units. Unlike conventional programming methods, encapsulation requires that present and future uses for a data structure be explicitly recognized by the developer of the class and the interface between the data and the outside world be declared. So, a class is a “black box” which offers a selection of services while hiding the details of how these services are actually implemented. This leads to the design of a class as a self-contained, encapsulated programming entity, which in turn leads to reusable code.

Inheritance

Inheritance allows the construction of a class to include all the members (data and functions) of another class. The class whose members are included is called the *base* class; a class which is being constructed from the base class is called a *derived* class. Inheritance encourages the design of hierarchical data structures with classes as the building blocks. This also yields reusable code, since a new class is more easily constructed from an existing class which already has some of the desired properties.

Late binding

Binding refers to the process in which a procedure and the data on which it is to operate are related. Traditional languages use early binding which performs this relation during code construction. In contrast, OOP provides late binding which delays the binding process until run-time.

Many of OOP's characteristics can be traced to the SIMULA language [13]. SIMULA has been popular in academic use but has never gained widespread use in the commercial environment. While SIMULA embodies some of the OOP concepts, it is not a pure OOP language. Smalltalk, one of the purest OOP languages, has been heavily influenced by SIMULA's model of computation. Now, there are many OOP languages available in the market. Some of these are special-purpose languages presented as front-end program development environments (especially in the area of object-oriented database management). Most of these hybrid object-oriented languages have extended the definitions of popular languages such as Pascal, C, or LISP. Among these, the C++ language, an extension of C, is known as the leader. C++ was designed by Bjarne Stroustrup at AT&T in the early 80s [24]. Its acceptance spread from AT&T to major universities and to other computer industry firms.

The first implementation of C++ was released as a preprocessor for any C compiler. This decision was made to ease the adoption of C++. Currently, there are many C++ compilers.

2.2 Basic concepts of computer simulation

All real world systems have something in common: they contain interacting subsystems and the task of these subsystems is to convert system inputs to system outputs. So, we define a system as a relation between inputs and outputs.

Simulation is the "imitation" of the operation of a real world system over time [1]. This process has the purpose of generating an artificial log of the system, and results derived from this history can be used to draw conclusions

on the operating characteristics. Regardless of its complexity, the successful simulation of a system depends on an understanding of its structure. Such an understanding is required to study how an organized collection of subsystems can process the inputs.

The behavior of a system is studied by the development of a simulation *model*. The model is usually a collection of statements and assumptions about the operation of the system to be simulated. These assumptions are expressed in mathematical, logical, and symbolic relationships between the objects of the system.

In some cases, the developed model can be “solved” by mathematical methods. Such solutions require the help of techniques like differential calculus, probability theory, algebra, etc. However, many real world systems are so complex that their models cannot be solved analytically in a precise way. In those cases computer simulation is used to imitate the behavior of a system. The output of simulation is an envisionment of the real system and is accompanied by necessary statistical data.

Our study covers the simulation of discrete event systems. In a *discrete system*, the system state changes only at discrete points in time. For example, in a queueing system, the variable representing the number of entities in the queue changes only when a new entity arrives at the queue or when an entity leaves the queue. The major concepts of a discrete-event model of a system are briefly defined as follows:

System state	A collection of variables that contain all the information necessary to describe the system at any time
Entity	Any object or component in the system which requires explicit representation in the model (e.g., a server, a customer, a machine)
Attributes	The properties of a given entity
Set	A collection (list) of associated entities, ordered in some logical fashion (e.g., a queue of customers waiting for service or a set of parts traveling in a conveyor)

Event	An instantaneous occurrence that changes the state of the system (e.g., arrival of a new customer)
Activity	A duration of specified length, with a known beginning and end (e.g., a service time or an inter-arrival time that is defined in terms of a statistical distribution)
Delay	A duration of unspecified length, with an unknown end (e.g., the waiting time for a customer in a queue, which depends on the total of service times of the customers that are ahead of that customer in the queue)

The task of the modeler is to extract these components from the real world system and to construct a computational framework in a programming language. Modeling is the first and the most important phase of simulation. Once a validated model is ready, the other phases are carried out more easily. In [19], Pidd describes the stages of a simulation process as:

- modeling
- programming
- experimentation

Modeling usually employs a number of approaches to characterize the system: the use of logic to represent the rules which govern the behavior of the system, the representation of stochastic behavior by taking samples from a probability distribution, the comparison of alternative policies in terms of modularity and scalability, and finally validation.

Many simulation applications are large, i.e., they have many entities, undergo many state changes, and comprise numerous possible interactions between entities. So, a modular model must lead to a modular program for the overall performance of the project. Considering that the program can be used by an unskilled client, the interface between the program and the user should receive special emphasis. Generally such programs employ three main sections: a parameter editor which allows the user to set the values of the variables of the system, a simulator which simulates the operation over time using the logic and rules of the system, and a report generator for the presentation of results.

Instead of experimenting with a real system, it is easier to perform trials on a dynamic computational model (where the attributes of the system are easily changeable). If the model is a valid representation of the system, the results of the simulation can be transferred to the real world.

The three stages that are described briefly above are not entirely distinct. When developing a system model, the analyst should consider the programming implications of the model. Similarly, when programming, the ease of experimentation should be kept in mind. In the experimentation phase, potential errors or new ideas may lead to revisions in the model and the program. Our study is aimed at the development of an approach that covers the boundary between the modeling and the programming phases.

2.3 OOP and simulation code development

The purpose of modeling a real world system is to identify the components (entities) of a process and to define the interfaces of these entities with each other. For example, in a single-queue/single-server system, the entities are the customers (entities to be served), the queue (the FIFO list where customers are forced to wait), and the server (the entity where customers get service and leave the system). Each of these entities have attributes and behavior to characterize their operation in the system. To model this process, the system analyst should extract this characterization. Once the abstraction of the entities are clearly defined, the complete system can be modeled easily.

OOP focuses on the objects that make up the system. The behavior of each object is encapsulated in the object itself rather than the main program. This results in modular and reusable code because specific details are embedded in the declarations rather than the main code. The guidelines of OOP are stated as follows [10]:

- Identify the classes in the system
- Define the interface of each class with other classes and the system
- Implement and validate the classes

- Write the main program which creates and manipulates the objects according to the interfaces provided

In fact, the above guidelines can be applied to simulation code development. The entities (object classes) coming from the model are the components of the system. Their interaction (interface) with other classes is also defined in the model. Mapping this formalization to code in an object-oriented fashion will lead to a program which will ensure the correct representation of the system. After the correct implementation of the classes, the main program initiates the simulation and invokes the necessary methods to run the complete process.

From this point of view, the above guidelines can be mapped to object-oriented simulation code development as follows:

- Identify the components and processes (entities) of the system under study
- Define an object class to represent each entity of the system along with its interface
- Characterize the conditions that lead to changes in the system state, treat these as events and specify the actions of scheduling, occurrence, and results of these events in the object classes they are related
- Develop the main program which creates the entities of the system

Once initiated correctly, the overall running of the program is completely determined by the behavior of the entities. For if our major purpose in simulation can be stated as observing the changes of a system with respect to some parameters, these changes are embedded in the entities and these events invoke the necessary methods of the classes to represent the state evolution of the system. For example, the queue in a single-queue/single-server system will contain the necessary data structures for keeping the customer entities and will provide the necessary services for the arrival of a new customer into the queue and the removal of the next customer for service. During the execution of these events, necessary statistics, e.g., the number of entities inserted into the queue or the total waiting time in the queue, will be collected automatically.

Two aspects of OOP, *inheritance* and *reusability*, are also worth noting here. Clear definition of system entities would allow developers to minimize their programming efforts. Reusable components and submodels can provide the necessary services without a need for inspecting or verifying the implementations of these previously defined classes. Such a clear definition will focus on the characterization of an entity along with its interface and will not probably contain any piece of code that is specific to an application. For example, a queue has its major attributes and behavior, and once defined, this class can be used in a variety of applications that need to implement such FIFO lists. If extra or different behavior of a previously defined entity is required, then inheritance can be used to refine the old definition.

Inheritance allows a new class to be derived as an extension of an existing class. The derived class, in addition to inheriting the attributes and operations of the base class, can add new members of its own as well as restricting access to the definitions of the base class. Inheritance promotes modularity because it allows designers to express the ways in which objects are similar (or different).

2.4 The use of a class library

The term *library*, as a software engineering concept, is used to denote a collection of data structures and subroutines kept in an orderly fashion. The aim of this collection is to allow the reuse of previously defined data types and functions whenever possible. A carefully designed library can reduce code development time in a programming environment.

The most distinguishing advantage of object-oriented libraries comes from the inheritance property. In procedural languages, it is not easy to borrow the characteristics of a data structure without redefining it entirely. But OOP allows the definition of new classes as extensions of existing classes. Therefore, having a core set of class definitions for a particular purpose, the programmer can derive new classes with a higher level of complexity without rewriting all the definitions.

Pidd [19] states that the notion of establishing a simulation software library will increase the modularity of the applications developed. Rather than

devising subroutines from scratch, these structures can be devised and kept in a library as reusable code. For example, complex queue handling may be a component of many models so the necessary class definitions can be made available in a library. They are then included as necessary in new programs; when more specialized attributes or actions are needed, appropriate derived classes might be constructed.

Simulation programs make heavy use of certain abstract data types like queues, conveyors, servers, etc. Therefore, the existence of a class library which provides the skeletal elements of a simulation application is clearly useful. Such a library should contain the most basic elements. Another important point is to allow the programmer to extend the services of the library. This can be accomplished by defining *subclasses* from the existing classes. These new elements can use the actions of their parent classes or override them.

In summary, major roles of a simulation class library can be stated as follows:

- Provide a set of basic objects and actions related to these objects that are commonly implemented in simulation software and include services such as gathering statistical data and error-checking
- Allow the user to make extensions to this library to obtain more specialized components and actions

2.5 Previous and related work

Looking at the history of simulation, we can state that early applications have been implemented through the use of general purpose programming languages such as FORTRAN, ALGOL, Pascal, C, PL/1, and BASIC. Now, there are more than one hundred special purpose simulation languages available. The most popular of these for discrete event simulations are GASP [20], GPSS [23], SIMSCRIPT [22], SIMAN [17], and SLAM [21].

The availability of these languages has not changed the fact that FORTRAN is the most popular language for simulation. Most authors attribute this to

the fact that the choice of language is primarily based upon the availability and the user's knowledge of the language [21].

Continual search for better approaches for implementing simulation models has shown that there is a promising relationship between the concepts of simulation and expert systems [11, 14, 15, 16, 18]. In particular, it seems that the OOP approach of expert systems is appropriate for implementing simulation models [3, 25].

It is not possible to cover all the research on object-oriented approach to simulation in this section. In the sequel, we review some studies that are most relevant.

MODSIM II [2] and Sim++ [10] are two object-oriented languages that can be used for simulation purposes. The syntax and structure of MODSIM II is based on that of Modula-2. It has additional constructs for object types and simulation. The compiler emits C code and its simulation constructs are based on SIMSCRIPT II.5 [22] language. Its graphical capabilities provide the user with a variety of interface tools. From the OOP view, it supports inheritance, dynamic binding, polymorphism, data abstraction, and information hiding.

Sim++ is a C++ package of object types and routines specially designed for writing object-oriented parallel simulations that execute on multiprocessors. Besides portability, the system provides facilities for parallel I/O, built-in user level and system level tracing, performance analysis, and run-time mapping of entities to processors to support parallel simulation. Standard simulation libraries for random number generation, data collection, and linked list manipulation are also available.

In [7], Basnet et al. describe their research to develop an object oriented modeling environment. Highlights of their work are as follows:

- Physical and information/decision components of a system are modeled separately
- Set theoretic formalisms can be constructed to represent system intelligence
- A model specification language to describe the fundamental structure of

system elements is necessary

- It is desirable to provide the system modeler with a graphical environment that allows the building, running, and analyzing of a model without directly interacting the language

Kaylan [9] focuses on a framework for discrete event simulation by supporting the model construction and experimentation phases with a graphical user interface. This framework is implemented in Smalltalk-80 [8] language with the OOP approach. He states that OOP incorporated with discrete event simulation reveals a modular and expandable simulation application environment which can be modified for modeling a range of production systems from job shops to flexible manufacturing systems.

The appropriateness of OOP languages for developing discrete event simulations is also discussed in [6]. Examples of implementations are presented using a library of C++ classes. As a result, Eldredge et al. state that OOP is highly compatible with the representation used in simulations. The paradigm enables a program to be written with a focus on the description of the problem rather than the algorithms for solving the problem.

Blair and Selvaraj [4] present DISC++ (Discrete Event Simulation in C++), a library of routines written in C and C++ to support the design and programming of simulation models. DISC++ allows the simulator to construct simpler models from standard library objects or design more complex models by deriving specialized and sophisticated objects from the library objects.

Chapter 3

Design of the class library

The design of a class library based on object-oriented principles mainly consists of the declaration of basic classes. By declaration, we mean the definition of the object classes along with their attributes and methods. In this chapter, first, we introduce the classes we have included in our design. Then, we present the details of developing simulation applications in our paradigm. This is followed by a section on the scheduling and execution of events. The last section treats the mapping of simulation models to source code.

3.1 Object-oriented design of classes

OOP treats each data unit as an object. In fact, this is not the case for most of the OOP languages of today as these languages have built-in data types like integer, character, etc. with predefined operations on them. Besides, from a utilitarian viewpoint it is wasteful to declare an object for each piece of data used in an application. A better interpretation for the declaration of objects might come from the notion of “necessity.” In other words, one should define an object class for a particular purpose if this data type is really most appropriately represented and used in the form of an object. Then, the question is to determine the “appropriateness.”

Before introducing a new class for a data type of interest, the following issues should be studied:

- Does the associated data type contain information that should be kept out of the “sight” of the main program? In other words, is it necessary to provide access to all attributes of the data or would a clean interface be adequate?
- Is it possible to represent the data as a self-contained unit? This might be owing to the fact that not all the users of a class should care about the overall aspects of the class. Objects derive their power from their “privacy” and this should be kept in mind in order not to declare classical record structures in the form of objects.
- Can the data type later be used for similar purposes? If so, is it possible to declare new data types and functions depending on the existing definition with little effort?

The questions above correspond to the principles of abstract data types, encapsulation, and inheritance, respectively. The difference between an object and other structures finds its explanation in these principles. For object-oriented simulation applications, it is therefore necessary to consider the use of objects in such a fashion. Before mapping the components of a model to source code, the designer/programmer should decide about the data representation for these components.

3.1.1 The use of prototypes

In the early stages of our study, we developed three prototype simulation applications with the sole purpose of determining the basic object types of a general simulation application. The development languages were C and C++ but the design was not object-oriented. These applications were later used as a testbed to investigate the effectiveness of the object-oriented approach. We now briefly describe these prototypes in order to give an idea of the skeletal elements of a simulation application.

- **A single-queue/single-server system**

In this model, the parameters of the system are the inter-arrival time of the customers and the service time of the server. The execution is quite

simple: customers entering the system join the queue with respect to a given distribution of inter-arrival time and the server provides service to the customers by removing them from the queue one by one. At the end of the service time, customers leave the system.

- **A production-line system**

This is a modified version of the preceding system. Entities enter the system and travel through a number of server units before they leave. Transportation between two consecutive servers is performed by carriers. To avoid the blocking of servers, input and output buffers are placed before and after each server. After the last server, entities leave the system. The parameters of the model are the inter-arrival time of entities, the number of servers, the service time distributions of servers, the capacities of buffers, and the travel-time and capacities of the carriers.

- **An elevator system**

The model consists of a number of floors and an elevator. A customer arriving at the elevator indicates a request and starts waiting. The elevator visits the floors to meet the requests. This is done by picking up the customers who wish to travel in the current direction of the elevator. Customers leave the system when they reach their floor.

In addition to determining the basic components of a general simulation application, the implementation of these prototypes has also helped in characterizing the execution principles of such applications. Briefly, the steps of execution of a simulation application are as follows:

-Setting up the model: In this phase, the components of the system are created and the necessary parameters are set. This also includes the definition of the interface between the components by declaring the actions to be taken at the time of the events.

-Running the model: The program is run for a certain length of time. Once the model has been defined, this phase is used for validation of the system and collection of the results.

-Output of results: After the execution of model, the results should be presented in a proper fashion. The results are mainly statistical information

representing the system performance but other information like a transcript of the overall run can be helpful.

Our primary aim in the implementation of the prototypes has been to determine these characteristics. Using the ideas acquired from this phase, our fundamental task—the determination of the classes in SIMLIB—has been eased to a great extent.

3.1.2 Class definitions

Inspired by the ideas of the prototyping phase, we have divided our classes into two categories:

- **System classes:** These represent the real-world components of a model. Examples include servers, queues, or the entities traveling through the system.
- **Auxiliary classes:** These do not have physical counterparts but rather serve as control and support structures. Events, event lists, and distributions are in this category.

The complete list of classes we have decided to include in our library can be found in Tables 3.1 and 3.2. (These do not reflect the initial structure of our design. As we proceeded with the development of SIMLIB, we used the three prototypes as a testbed. This resulted in contributions and changes in the design.) It may not be easy to meet the requirements of each and every simulation application but a library which provides the general objects with a flexibility to allow the tailoring of the provided services is the purpose of this study.

Using the inheritance principle, the system classes have been designed in a top-down manner. In fact, all these objects share some common characteristics like attributes (that denote system-related information) or actions (that are performed to keep statistical data). For example, each system object should have an ID number, a status to represent the current state of the object, and a name for debugging purposes. In addition to these attributes, methods to set

Class	Represents
SIMOBJECT	the base object for the generic behavior of system objects
ENTITY	the entities that are serviced and transferred in the system
NODE	a particular point in the system
COLLECTION	a list of entities
QUEUE	a collection of entities under a FIFO regime
BUFFER	a queue with a predetermined capacity
SOURCE	the nodes where entities are generated
SINK	the nodes where entities leave the system
SERVER	the nodes in the system where entities are processed
CARRIER	the objects that transfer the entities through the system

Table 3.1. System classes in SIMLIB

Class	Purpose
EVENT	store the time point and type of changes in system state
EVENTLIST	keep a list of future events in chronological order
DISTRIBUTION	provide statistical distributions of durations for activities
SIMULATION	hold the current state of the system

Table 3.2. Auxiliary classes in SIMLIB

and retrieve the values of the attributes can also be shared by the classes. For example, once a queue and a buffer are considered, the only difference between these two is seen to be the behavior of the objects when an entity is inserted into these collections. The queue always places the incoming entity to the end of the list while the buffer does this only if its capacity is not exceeded. So, the *Remove* method of the buffer can easily use the corresponding method for the queue with a small amount of extra code.

The hierarchy of system classes in SIMLIB is depicted in Figure 3.1.

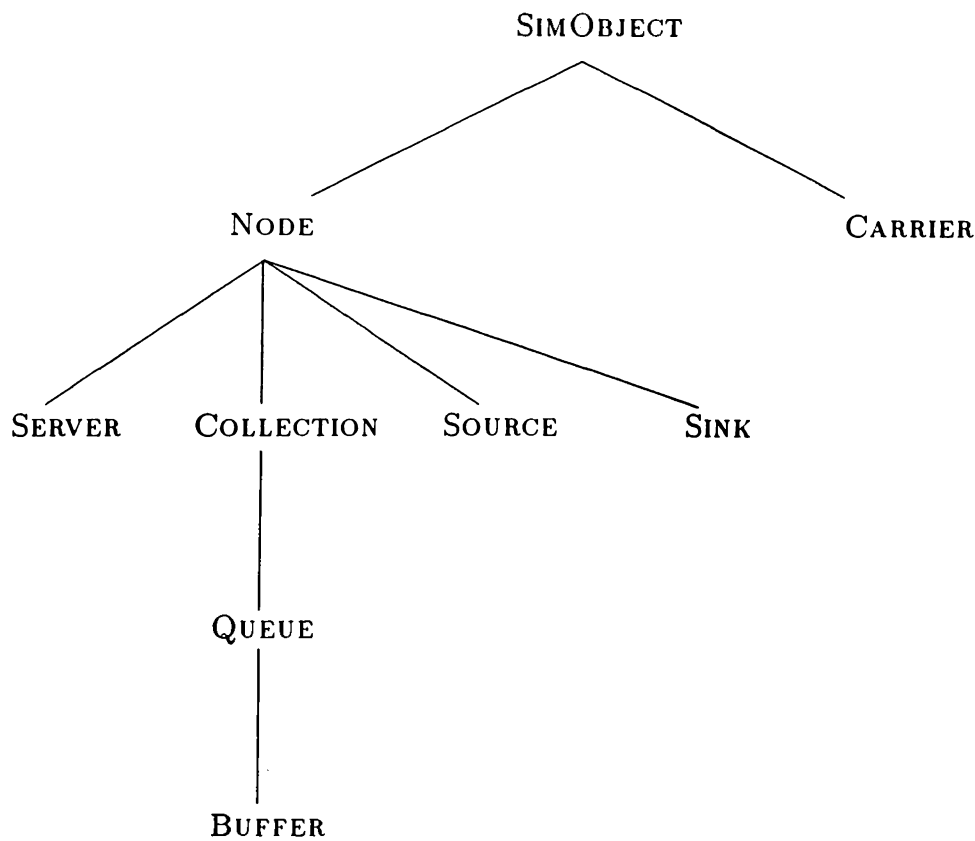


Figure 3.1. The hierarchy of system classes in SIMLIB

3.2 General form of a simulation application

3.2.1 Construction of a model

A system consists of objects, each serving a particular purpose. When we consider a single-queue/single-server model, the objects are:

- A **SOURCE** where the incoming entities are generated with respect to a statistical distribution for determining the inter-arrival time
- A **QUEUE** where entities are forced to wait under a FIFO regime until the server is ready to provide service
- A **SERVER** where entities get processed after leaving the queue

- A SINK where entities leave the system

All these objects, besides their methods to process the entities of the system, hold the necessary statistical information for the final stage of the implementation. As for the source, the number of entities generated to the system or for the queue the total time that entities have spent waiting in the queue or for an entity the total time that has been spent in the system (from source to sink) are examples of such statistical data. At each state change of the objects (namely, at each event) the attributes representing the statistical data are updated.

Definitions of these objects are provided in our library for ease of use. But before using an object, the programmer should “construct” the object. This construction process consists mainly of the allocation of memory space for the object. (It also sets the values of the attributes of the object.) There can be any number of constructor methods for an object class with the constraint that the types or number of the specified attributes are different. The following example for the construction of a server will make this point clear.

SERVER, in addition to those it inherits from SIMOBJECT and NODE, has extra attributes to represent the structure and state of a processing entity:

- the total number of entities processed,
- the total time that has been spent idle,
- the time point that the server has most recently started waiting,
- the distribution of service time for the incoming entities,
- the distribution of repair time in case of crashes,
- the probability of crash for the server.

The corresponding class definition is given in Figure 3.2.

In C++, a method having the same name with its class is treated as a constructor. In Figure 3.2 there are three constructor methods. The first of these is the default constructor that is called when no parameters are supplied by the program. The other two methods specify different ways of creating a server object, the first with five parameters and the second with three parameters. The reason for this is that in some applications the crash probability and the repair

```

class Server:public Node
{
    int total_entities;
    double total_idle_time;
    double last_wait;

public:
    Distribution *dist_service;
    Distribution *dist_repair;
    double prob_crash;

    Server(){}; //constructor methods
    Server(char*,int,Distribution&,Distribution&,double);
    Server(char*,int,Distribution&);
    // .
    // . other methods
    // .
};

```

Figure 3.2. The definition of SERVER

time distribution of the server may not be used. Then the parameters to be supplied to the constructor are the name and the id of the SERVER object (attributes coming from the superclasses) and the service time distribution only. The no-crash and crash cases are represented in the constructor declaration as in Figures 3.3 and 3.4, respectively.

The declarations of the single-queue/single-server application are given in Figure 3.5. This code fragment is used to create SOURCE, QUEUE, SERVER and SINK objects.

System objects, after their declaration, can be linked to each other to automatically navigate the entities through the system. These links are declared by the *SetLinks* method of the objects. Each object can specify its input and output links by this method. The input link denotes the node entities are received from, and the output node denotes the node entities are sent to.

```
Server::Server(char* n,int sid,Distribution& ds)
{
    name=malloc(strlen(n)+1);
    strcpy(name,n);
    id=sid;
    status=IDLE;
    dist_service=&ds;
    total_entities=0;
    total_idle_time=0.0;
    last_wait=0.0;
    type=oServer;
    insert_to_objlist(this);
}
```

Figure 3.3. Constructor of SERVER object for the no-crash case

```
Server::Server(char* n,int sid,Distribution& ds,
               Distribution& dr,double p)
{
    name=malloc(strlen(n)+1);
    strcpy(name,n);
    id=sid;
    status=IDLE;
    dist_service=&ds;
    dist_repair=&dr;
    prob_crash=p;
    total_entities=0;
    total_idle_time=0.0;
    last_wait=0.0;
    type=oServer;
    insert_to_objlist(this);
}
```

Figure 3.4. Constructor of SERVER object for the crash case

```
Source SO("Source",0,dist_arrival);  
Queue QU("Queue",0);  
Server SE("Server",0,dist_service); //no crashes  
Sink SI("Sink",0);
```

Figure 3.5. Object declarations of the single-queue/single-server application

For the single-queue/single-server case, SOURCE sends the generated entities to QUEUE, QUEUE sends the entities coming from the source to SERVER and SERVER, after servicing the entities coming from QUEUE, sends them to SINK. The declaration of these links is as in Figure 3.6.

```
SO.SetLinks(SO,QU);  
QU.SetLinks(SO,SE);  
SE.SetLinks(QU,SI);
```

Figure 3.6. Object links for the single-queue/single-server application

The use of constructor and *SetLinks* calls make up the representation of the model in terms of system objects. At this point, the objects are ready to process as defined in their methods. But the synchronization and execution of the processes are determined by the events of the system. To run the model, the programmer should also indicate the events of the system.

3.2.2 Running the model

The running of a model can roughly be described as the continuous execution of events in the system. Before describing this process, it is necessary to define the object which controls the state evolution of the model. This object is of the SIMULATION class; each simulation application either uses an object of this class or derives a new class from it.

The attributes of a SIMULATION object are the current time point of the simulation run, a chronological list of events to occur, and a status variable (Figure 3.7). The methods are as follows:

```
class Simulation
{
public:
    double SimClock;
    EventList FEL;
    int SimStatus;

    friend SimEvent;

    Simulation();
    virtual void Init();
    virtual void Step();
    virtual void Run();
    virtual void Done();
    virtual int ShouldStop();
};
```

Figure 3.7. The definition of SIMULATION

- a void constructor
- Init* is called to set some application specific values before the execution of the simulation
- Step* executes the next event to occur
- Run* executes events until the stopping condition of the simulation is fulfilled
- Done* is called after the simulation to display statistical results
- ShouldStop* checks, before the execution of each event, whether the simulation is to continue

Run, after calling *ShouldStop*, invokes *Step* to carry out the simulation. The user can override *ShouldStop* to define the stopping condition of the program. Examples of the stopping condition can be a particular status of one of the system objects, a bound on the total number of entities processed by the system, or a time limit. The code fragment in Figure 3.8 denotes the *ShouldStop* method of our single-queue/single-server example which stops the execution after 1000 units of time have elapsed.


```
int QueueSimulation::ShouldStop()
{
    if ( (SimClock>=1000) || (SimStatus!=SIM_OK) )
        return(TRUE);
    else
        return(FALSE);
}
```

Figure 3.8. An example *ShouldStop* method

The simulation is carried out by executing the events that are retrieved from the eventlist of `SIMULATION`. These events are either inserted to the list (scheduled) by the programmer explicitly or the execution of an event can schedule another event in a triggered fashion. For example, considering the arrival of an entity to a queue, if the queue is empty at that time and the server that takes input from the queue is idle to accept the entity for service, then the removal of the entity from the queue and the start of service for that entity can be automatically scheduled. (Otherwise the entity is forced to wait in the queue.)

Done is used to display the results of the simulation run. This is accomplished as follows. All system objects are kept in a list that is maintained by the system. Insertion of objects to this list is handled by the constructors. *Done* sends a message to all these objects in the list to invoke their *StatPrint* methods which output statistical information related to their object. A sample output for our single-queue/single-server application is presented in Figure 3.9.

The body of a simulation program mainly consists of *Init*, *Run*, and *Done*. These methods correspond to the modeling, running, and experimentation phases of a simulation application, respectively. The catch is that *Init* performs only some initializations while the components of the model are defined as object declarations browsed from the library.

```
Simulation stopped with status 0 at time 50001.4
***Statistics for source <Source>***
Number of entities generated: 10000
Mean arrival rate: 5.00014

***Statistics for queue <Queue>***
Total entities inserted in the queue: 10000
Maximum queue length: 3
Mean time spent in the queue: 1.1794
Mean number in the queue: 0.235873

***Statistics for server <Server>***
Number of entities served: 10000
Total busy time: 39877.4
Mean service time: 3.98774
Utilization: 0.797526

***Statistics for sink <Sink>***
Number of entities processed: 10000
Total waiting time for processed objects: 23588
Total system time for processed objects: 51671.4
Mean time spent in the system: 5.16714
Mean number in the system: 1.0334
```

Figure 3.9. Sample output from the method *Done*

3.3 Scheduling and execution of events

There are two types of system-event objects: `STABLEOBJECT` denotes the state changes for the system-objects derived from `NODE` and `MOVINGEVENT` is associated with `CARRIER`. The class definitions for these two classes and their base class are shown in Figure 3.10.

The base class `SIMEVENT` includes attributes denoting the time and type of the event and the system object which the event effects. There are also links to the next event to occur and to the `SIMULATION` object that owns the event. Derived from this class, `STABLEEVENT` has an extra attribute to hold the entity that is the object to be processed by the event. The other subclass, `MOVINGEVENT`, has two extra attributes to specify the departure and arrival

```
class SimEvent
{
public:
    double time;
    int type;
    SimObject* owner;
    SimEvent* next;
    Simulation* sim;

    friend EventList;
    friend Simulation;

    virtual void Print(ostream&){};
    virtual void Execute(){};
    virtual void SetSim(Simulation*);
    double GetTime();
};

class StableEvent:public SimEvent
{
protected:
    Entity *object;
    friend EventList;
public:
    void Print(ostream&);
};

class MovingEvent:public SimEvent
{
protected:
    SimObject *departure_point;
    SimObject *arrival_point;
public:
    void Print(ostream&);
};
```

Figure 3.10. Class definitions for the event classes

nodes for transportation events.

SIMLIB provides a list of standard events (Table 3.3). The programmer is free to define events for other actions. Each user-defined event object should incorporate the methods for constructing the object properly. Another method that should be defined is *Execute*. This method declares the actions to be performed when the event occurs.

Type	Object	Event
STABLEEVENT	SOURCE	ENTITYINTOSYSTEM
	QUEUE	ENTITYINTOQUEUE
		ENTITYFROMQUEUE
	SERVER	ENTITYINTOSERVICE
		ENTITYFROMSERVICE
	SINK	ENTITYFROMSYSTEM
MOVINGEVENT	CARRIER	BEGINTRANSPORT
	CARRIER	ENDTRANSPORT

Table 3.3. List of event classes provided in SIMLIB

The events are kept in the *FEL* (Future Event List) attribute (an instance of *EVENTLIST*) of the *SIMULATION* object in chronological order. *FEL* includes two mostly used methods: one to insert new events with respect to the value of their time attribute and another to remove the next event from the list for execution.

Each event class has its *Execute* method which is invoked when the *Step* method of the simulation object retrieves it from the event list. The *Execute* methods perform the actions to represent the state changes of the system. They mainly call the methods of their owner objects for this purpose. In many cases, the execution of an event triggers another event and new events can be scheduled. To illustrate the scheduling and execution of events, we conclude this section with the code of the *Execute* method of *ENTITYINTOSERVICE* (Figure 3.11).

```
void EntityIntoService::Execute()
{
    double duration;

    //invoke the server's BeginService method
    owner->BeginService(object,time);

    //compute the service time
    duration=(owner->GetDist())->generate();

    //schedule the event for end of service
    sim->FEL.schedule(new EntityFromService
        (time+duration,owner,object));
}
```

Figure 3.11. A sample *Execute* method

3.4 Mapping a model to a program

In the preceding sections, the steps of developing a simulation application within our framework were briefly described. In this section, we combine these ideas in the form of a methodology.

Before starting the coding process, a verified model of the system to be simulated should be ready. The model should focus on two aspects of the model: system components like queues, servers, or carriers, and events of the system. If no extra behavior is required, then system objects and events can be directly browsed from the library. Otherwise, the programmer should derive new classes representing the application-specific elements of the model.

Objects, whether system-defined or user-defined, are introduced to the program along with their constructor calls. The constructor calls will insert the objects in a system-maintained list that can be used for debugging and output purposes. For compatibility with the system-objects, user-defined objects should inherit the necessary methods (or they would override the inherited methods with proper implementations).

The links between objects can be declared with the *SetLinks* methods of the objects. To do this call, each object should have one input and output link. If there are multiple links, then the decision for the navigation of objects should be coded into the methods of the derived system-object or into the *Execute* method of an associated event object.

User-defined events can be employed to add extra behavior to the system. This is done with the implementation of the *Execute* methods of the defined events. Most system-events trigger other events for the continuity of the execution; the same paradigm applies to user-defined events. (The event list should never be empty.)

Once the definition of the system-objects and events are coded, the last step is to define or derive an object (of type *SIMULATION*) for the control of execution. The *Init* method of this object schedules the initial events which enforce the *SOURCE* objects of the system to generate new entities. These events trigger other events which in turn invoke the system-objects to simulate their associated behavior. The execution lasts until the stopping condition (available in *ShouldStop*) occurs.

Once the above steps are carried out, only a small amount of code is required to finish the implementation. An example of such a main body that uses a derived simulation object for a single-queue/single-server is presented in Figure 3.12.

```
QueueSimulation QS;  
    //control object derived from Simulation class  
  
int main()  
{  
    QS.Init();  
    QS.Run();  
    QS.Done();  
}
```

Figure 3.12. The main body for a derived *SIMULATION* object

Chapter 4

Implementation

The purpose of this chapter is to study the implementation of SIMLIB. The most important task in this regard was the definition of the classes. First we give a detailed description of these classes—their attributes and methods. Then, communication among the objects of an application is described. The addition of user-defined behavior to applications is treated next. The last section considers the system-maintained services provided by SIMLIB.

4.1 Definition of classes

SIMLIB has been implemented using the C++ programming language on SUN Workstations running under the UNIX system. C++, inheriting the portability of the C, has received wide acceptance from both academic and industrial users. The broad range of functionality of the language (combined with its availability on every UNIX system) has been the major reason for our preference.

Before studying the classes in SIMLIB, we want to describe the notion of *virtual functions*. The use of dynamic binding, i.e., the relating of a procedure with its data at the time of execution, is one of the key properties of OOP. C++ supports dynamic binding through virtual functions. A virtual function allows overriding of a function name. Each subclass within the hierarchy can choose a different implementation for this function. For example, in a graphical environment that uses a hierarchy of classes to represent different shapes like

circles, triangles, etc., a member function to draw these on the screen can be defined. Declaring this function as virtual in the base class and writing the code of the function for each class will result in the appropriate displaying of each shape object without knowledge of the type of the object. The compiler will bind the correct definition of this function at run-time. Each derived class may either inherit the virtual function from its base class or define a function of its own. The restriction is that the returned value and the arguments of a virtual function must remain the same for each class derived in the hierarchy.

SIMLIB makes use of virtual functions in order to improve readability. The same function can be used with different objects. For example, an entity removed from a queue can be sent to a server or another queue vis-à-vis a decision that is made during the execution of the program. Instead of writing lengthy conditionals, the code may be written by calling a single function (like `SENDEntity`), and the appropriate implementation is invoked by the compiler with efficient use of virtual functions.

Another concept we want to highlight is the use of *friend classes*. A class declared as “friend” can have access to the private instance variables and methods of a class. This feature is useful since without it, all class declarations would be “black boxes.” The notion of friend classes provides a method of communication between classes that need the data of each other.

4.1.1 The SIMOBJECT class

This is the base class of all the system objects (Table 3.1). SIMOBJECT member variables are the common attributes for these classes and SIMOBJECT functions are mainly the virtual definitions for methods that are implemented by the classes in the hierarchy of Figure 3.1. Being a base class, this class is not directly used in applications. It has access to SIMEVENT, STABLEEVENT, and MOVINGEVENT with friend declarations.

Attributes

-*type*: Denotes the type of the object. It takes values from an enumerated set of system-object types like `oServer`, `oQueue`, `oCarrier`, etc.

-*name*: Stores a name for the object in a string. This is used for display and debugging purposes.

-*status*: The status of an object is kept in this. A typical example is the *BUSY* and the *IDLE* states of a server.

-*id*: Denotes an identification number for the object. This can be used to unify objects, especially when more than one instance of a class is used in an application.

-*next*: Since all system objects are kept in a list by the system, this is used as a link to the next *SIMOBJECT* in the list.

-*inlink*: This is a link to another *SIMOBJECT* which provides the entities for the object.

-*outlink*: Similar to *inlink*, this denotes the *SIMOBJECT* to which the object sends the output.

Methods

As the *SIMOBJECT* class performs no specialized action, its methods are designed to set and retrieve the values of the attributes:

-*SimObject*: The constructor method.

-*GetStatus*: Returns *status* of a *SIMOBJECT* instance.

-*SetStatus*: Sets the value of *status*.

-*GetId*: Returns *id* of a *SIMOBJECT* instance.

-*GetType*: Returns *type* of a *SIMOBJECT* instance.

-*SetNext*: Sets the *next* object in the system list for a *SIMOBJECT*.

-*GetNext*: Returns the *next* *SIMOBJECT* in the system list.

-*GetName*: Returns *name* of a *SIMOBJECT*.

-*SetLinks*: Sets the values of *inlink* and *outlink*.

-*GetInLink*: Returns a pointer to the *SIMOBJECT* identified by *inlink*.

-*GetOutLink*: Returns a pointer to the *SIMOBJECT* identified by *outlink*.

-*Print*: Outputs a brief description of the *SIMOBJECT* by indicating *name* and *id*.

4.1.2 The *NODE* class

This is designed to represent a particular point in a system. Being the parent of all other system classes except *CARRIER*, it denotes the objects that are

stable throughout the execution of a model. It may not be used by most applications but is useful to represent points of the system where special actions are performed (e.g., decision making). It also serves as a superclass for user-defined classes to represent stable objects.

Attribute

-*passerobject*: Denotes the entity that is currently active at a NODE object. Examples include an entity that is being processed at a server, or an entity that is to be removed from a queue.

Methods

- Node*: The constructor method.
- SetPasser*: Sets the value of *passerobject*.
- GetPasser*: Returns the entity that is active at the node.

4.1.3 The COLLECTION class

This is used to keep sets of entities such as the contents of a carrier. It stores the entities in a linked list. Insertions and removals are performed at the head of the list. QUEUE and BUFFER classes are derived from this class.

Attributes

- head*: Holds a pointer to the first entity in COLLECTION.
- count*: Stores the number of entities that are currently in COLLECTION.
- totalentries*: Stores the total number of entities that have been inserted into COLLECTION since the creation of the object.
- maxentries*: Stores the maximum number of entities that have been stored in COLLECTION at a time point since the creation of the object.
- totalwait*: Holds the total of waiting times for all the entities that have been stored in COLLECTION.

Methods

- Collection*: The constructor method.
- Insert*: Inserts a new entity to COLLECTION. It has two implementations; one implementation requires an extra argument to specify the time of insertion for waiting time computations. For example, the contents of a transportation vehicle can be represented by a COLLECTION object and the time

that entities travel in the vehicle may not be treated as waiting time. For such cases only the functionality of *Insert* is required to load the entities, rather than statistical bookkeeping.

- Remove*: Removes the most recently inserted entity from *COLLECTION* and returns it. Just like *Insert*, it has a second form which performs the necessary statistical data management operations when a time argument is provided.

- IsEmpty*: Returns *TRUE* if *COLLECTION* has no entities.

- GetCount*: Returns the number of entities that are currently in the *COLLECTION* object.

- Print*: Outputs the entities that are in the *COLLECTION* object. This is done by invoking the *print* method of every entity in the list.

- GetTotalWait*: Returns the value of *totalwait*.

- BeginWaiting*: Performs the necessary actions to transform the entities to waiting state. (All the entities in the list receive a message to set their *lastwait* to the current time of the simulation.)

- EndWaiting*: Ends the waiting phase of all the entities in *COLLECTION*.

- IncTotalWait*: Increments the value of *totalwait*.

- UpdateTotalWait*: Sets the value of *totalwait* to a new value by adding the current waiting time of every entity in *COLLECTION*.

4.1.4 The QUEUE class

Derived from *COLLECTION*, this is used to keep entities in the form of a list such that insertions and removals are performed in a FIFO fashion. As it differs from *COLLECTION* only in functionality, *QUEUE* has no additional attributes. But due to its likelihood to be included in an application, extra behavior for the navigation of entities and statistical data output has been granted.

Methods

- Queue*: The constructor method; creates a *QUEUE* object with *name* and *id* provided as arguments.

- Assign*: Sets the values of *name* and *id* for an existing *QUEUE* object. This function has been provided for objects that are declared in object arrays. In such cases, the compiler performs the memory allocation by invoking the default constructor but the programmer may need to set these values explicitly.

-*Remove*: QUEUE overrides the *remove* method to represent the behavior of a queue by returning the least recently inserted entity.

-*ReceiveEntity*: This method forces QUEUE to schedule an event to receive a new entity.

-*RequestEntity*: This method answers a request that is made to QUEUE by scheduling an event to remove the entity in turn.

-*CanSend*: Returns TRUE if QUEUE can provide any entities to a request, FALSE otherwise.

-*CanReceive*: Since there is no restriction for insertion into a QUEUE, this always returns TRUE.

-*StatPrint*: Outputs statistical information about QUEUE.

4.1.5 The BUFFER class

Derived from QUEUE, BUFFER represents FIFO lists with a predetermined capacity. This restricts the use of *Insert* since an additional check on the current number of entities is required prior to an insertion.

Attribute

-*buffersize*: Denotes the maximum number of entities that can be stored in BUFFER at a time.

Methods

-*Buffer*: Constructs a BUFFER object with a given capacity.

-*Assign*: Sets the value of *buffersize*.

-*Insert*: Overrides *Insert* by performing a comparison between the values of *totalentries* and *buffersize* before an insertion. If the capacity has been reached, the entity is not placed in the list.

-*CanReceive*: Returns TRUE if an entity can be placed in the BUFFER, FALSE otherwise.

4.1.6 The SOURCE class

This is responsible for generating entities. It represents an input point where “outside” entities pass the boundary of the model. The generation of entities is

determined with a particular statistical distribution associated with the object.

Attributes

-*distgeneration*: Points to a DISTRIBUTION object which determines the inter-arrival time distribution of the entities generated.

-*totalentities*: Holds the total number of entities generated since the creation of the SOURCE object.

Methods

-*Source*: Creates a new SOURCE object with the values of *name*, *id*, and *distgeneration* provided.

-*Assign*: Sets the values of *name*, *id* and *distgeneration*.

-*GetDist*: Returns a pointer to the DISTRIBUTION object in *distgeneration*.

-*GetEntityInterval*: Returns the next inter-arrival time determined by *distgeneration* of the source object.

-*SetPasser*: Sets the *passerobject* (inherited from NODE) to the entity just generated.

-*StatPrint*: Outputs statistical information about the object.

4.1.7 The SINK class

This represents the points where entities leave the system and mainly collects statistics about the entities which have traveled through the system. The entities are destroyed upon completing their system life.

Attributes

-*totalentities*: Holds the number of entities that have left the system at the SINK object.

-*totalwaittime*: Holds the total value of waiting times for all the objects that have left the system at the SINK object.

-*totalsystemtime*: Holds the total of time values that have been spent in the system for all the objects that have left the system at the SINK object.

Methods

-*Sink*: Constructs a SINK object with the *name* and *id* values provided.

- Assign*: Sets the values of *name* and *id*.
- TerminateObject*: Updates the values of the statistical information attributes before destroying the entity that has arrived at the SINK.
- CanReceive*: This function always returns TRUE. It has been implemented to complete the family of virtual functions designed for the navigation of objects through the system.
- ReceiveEntity*: This method forces SINK to schedule an event to extract an entity from the system.
- StatPrint*: Outputs statistical information about the SINK object.

4.1.8 The SERVER class

This is designed to represent the system locations where entities receive service. These processing units determine the amount of time which the entities will spend for the required operations. (Examples include the service provided by the cashier in a bank or the milling process applied to a part in a factory.) The amount is generated from a service time distribution associated with the object. Optionally, the cases in which the server is not functional for some time (i.e., crashes) can be implemented.

Attributes

- totalentities*: Holds the number of entities that have been processed by SERVER.
- totalidletime*: Holds the total amount of time that SERVER has been in IDLE state.
- lastwait*: Denotes the time point that SERVER has most recently started waiting.
- distservice*: A DISTRIBUTION object to denote the service time distribution for SERVER.
- probcrash*: The probability of a crash after each service.
- distrepair*: A DISTRIBUTION object to denote the repair time distribution for SERVER in case of a crash.

Methods

- Server*: The constructor method. It has two forms: one requiring the *name*, *id*, and *distservice*, and another which additionally takes *prob_crash* and

dist_repair for servers that are subject to crashes.

- Assign*: Sets the values of the attributes of the SERVER object.
- BeginService*: Performs the actions to denote the beginning of service for an entity.
- EndService*: Invokes SERVER to end service after making the necessary statistical computations.
- IsIdle*: Returns TRUE if SERVER is IDLE, FALSE otherwise.
- GetDist*: Returns the *distservice* object of the SERVER.
- CanReceive*: Returns TRUE if SERVER can receive any entities for service, FALSE otherwise.
- ReceiveEntity*: This invokes SERVER to schedule an event to start servicing the received entity.
- StatPrint*: Outputs statistical information about the SERVER object.

4.1.9 The CARRIER class

This denotes objects that perform transportation between system objects. Examples include conveyors in a production system, or trucks carrying goods in a transportation network.

Attributes

- contents*: This attribute is a COLLECTION object to represent the set of entities that are currently in CARRIER.
- capacity*: The maximum capacity of CARRIER.
- entitiescarried*: Holds the total number of entities that have been carried by the object.
- distance*: The length of the path which the CARRIER object is traveling. This is used if the path of CARRIER is a single link between two system objects.
- speed*: The speed of the CARRIER object. The value of the attribute can be changed according to the status and the load of the CARRIER.
- totalwaittime*: Holds the total amount of time that CARRIER has been IDLE.
- lastwait*: Denotes the time point that CARRIER has most recently started waiting.

-*from*: Denotes the system object that CARRIER has most recently departed from.

-*to*: Denotes the current destination of CARRIER.

Methods:

-*Carrier*: Constructor method for CARRIER. Required attributes are *name*, *id*, *capacity*, *distance*, and *speed*.

-*Assign*: Sets the values of the attributes for existing CARRIER objects.

-*GetContents*: Returns the COLLECTION object of CARRIER to provide the list of entities being carried.

-*CanReceive*: Returns TRUE if CARRIER can load any entities for transportation.

-*CanSend*: Returns TRUE if CARRIER can provide any entities. In other words, this method can be used to detect whether CARRIER has reached its destination with its contents or not.

-*ReceiveEntity*: This method instructs CARRIER to load an entity into *contents*. If the capacity of CARRIER has been reached, then an event to start the transportation process is also scheduled.

-*RequestEntity*: Upon receiving this message, CARRIER removes an entity from its *contents*. If all the entities have been removed, then an event to travel to the loading point of CARRIER can be scheduled.

-*BeginWaiting*: Sets CARRIER to the waiting state.

-*EndWaiting*: Ends the waiting of CARRIER. The necessary computations to keep the statistical time values are also performed.

-*Move*: Performs the actions to denote the start of transportation for CARRIER.

-*Stop*: Performs the actions to denote the end of transportation for CARRIER.

-*StatPrint*: Outputs statistical information about the CARRIER object.

4.1.10 The ENTITY class

This represents the units that are serviced in the system, e.g., customers in a bank or parts in a production line. Their generation is performed at SOURCE and at any time point during the execution there can be many instances of entities. Upon completing their life after traveling in the system from one

object to another, they are destroyed at SINK.

Attributes

- entrytime*: Denotes the time point that ENTITY has been generated.
- totalwaittime*: Holds the total amount of time that ENTITY has spent waiting for service.
- lastwait*: Keeps the value of the time point that ENTITY has most recently started waiting.
- next*: Since entities can be kept in collections, this attribute is used as a link to denote the next ENTITY in the list.

Methods

- Entity*: Constructs a new ENTITY object.
- BeginWaiting*: Sets ENTITY to waiting state.
- EndWaiting*: Invokes ENTITY to receive service (processing, transportation, etc.) after a waiting phase.
- Print*: Outputs information about the ENTITY object.

4.1.11 The DISTRIBUTION class

This is designed to provide statistical distributions for say, inter-arrival times of customers or service times of servers. Using a random number generator, objects of this class return a duration of time with respect to the type and parameters of the required distribution. The class currently can provide statistical values for constant, normal, and exponential distributions.

Attributes

- type*: Denotes the type of DISTRIBUTION. It can take a value of `dConstant`, `dNormal`, or `dExponential`.
- mean*: For constant distributions, this denotes the constant duration value to be returned at each invocation. For normal and exponential distributions, this holds the value of the mean of the distribution.
- deviation*: For normal distributions, this holds the value of the standard deviation.

Methods:

- Distribution*: Constructs a DISTRIBUTION object. It has two forms; one of which requires the value of *deviation* for normal distributions.
- Generate*: Invokes DISTRIBUTION to return a time length with respect to *type*, *mean*, and *deviation*.
- Print*: Outputs information about the DISTRIBUTION object.

4.1.12 The SIMULATION class

This is used to keep the current state of the simulation. Each application either uses an instance of this class or derives a new class from it. SIMULATION controls the execution by invoking events at the time they are expected to occur. The stopping of the execution is also checked by this object.

Attributes

- SimClock*: Holds the current time of the simulation run.
- FEL*: Denotes the list of chronologically ordered events that are expected to occur.
- SimStatus*: The current status of the simulation run. Examples of system-defined states for simulation are SIM_OK and SIM_FEL_EMPTY. The programmer can define states depending on the characteristics of the application.

Methods

- Simulation*: The constructor method.
- Init*: This is used to perform some initializations before the running of the program. Initial events are scheduled with the help of this method. (System object links are also defined here.)
- Step*: Executes the next event from the event list.
- Run*: Runs the program by calling *Step* in a loop.
- Done*: Called after *Run* to display the statistical results.
- ShouldStop*: Checks whether the stopping condition of the simulation, viz. reaching a predetermined time limit, has occurred or not.

4.1.13 The SIMEVENT class

Events of a simulation application are represented by this class. In fact, SIMEVENT is the base class for the events of the system. The two subclasses, MOVINGEVENT and STABLEEVENT, inherit most of their attributes from SIMEVENT. This class provides access to the SIMULATION class as a friend.

Attributes

- time*: Denotes the time of the event.
- type*: Denotes the type of the event.
- owner*: Stores a link to the object at which the event occurs.
- next*: A pointer to the next event.
- sim*: Denotes the SIMULATION object executing the event.

Methods

- Print*: Prints information about the event.
- Execute*: This is responsible for the operations to be performed at the execution of the event. Each event class defines an *Execute* method which is called when the event is to occur. With this virtual function, all events are processed in the same way, while the implementation of the method can differ from class to class depending on the characteristics of the event.
- SetSim*: Sets the value of *sim* by defining the SIMULATION object which is to execute the event.
- GetTime*: Returns *time* of the event.

4.1.14 The STABLEEVENT class

This denotes the events that belong to the system objects except CARRIER. These events take only one ENTITY as their subject. (Therefore, there is only one additional attribute to denote this entity.) Examples include generating an entity, inserting an entity into a queue, or servicing an entity.

Attribute

- object*: Denotes the ENTITY which is the subject of the event.

The classes derived from `STABLEEVENT` serve to provide the general behavior of system objects. A list of these classes and the operations performed by their corresponding *Execute* methods are:

- **ENTITYINTOSYSTEM:** This event is invoked when an `ENTITY` is to be generated into the system. The owner of `ENTITYINTOSYSTEM` events is `SOURCE`. The *Execute* method of this event class sets the `ENTITY` generated as *passerobject* of the source. Then, an inter-arrival time is computed from the distribution of the `SOURCE` and the next `ENTITYINTOSYSTEM` event is scheduled. If another system object has been specified as the output node of `SOURCE`, then after querying the status of this next object, `ENTITY` is sent to the output node.
- **ENTITYINTOQUEUE:** This event denotes the arrival of an `ENTITY` to a `QUEUE`—which is the owner of this event class. With the execution of this event, the `ENTITY` (stored in *object*) that is passed to the event is inserted in the `QUEUE` (stored in *owner*). At this moment, there is at least one entity in the queue so the output node of the queue is checked and if it can receive entities, an `ENTITYFROMQUEUE` event is scheduled.
- **ENTITYFROMQUEUE:** Removal of an `ENTITY` from a `QUEUE` is performed by the execution of this event. The *Execute* method updates the contents of the owner queue. Then, the entity that is removed from the queue is sent to the next system object if this output node is able to receive entities.
- **ENTITYINTOSERVICE:** This event denotes the beginning of a service process at a `SERVER` object. By the *Execute* method of an `ENTITYINTOSERVICE` event object, the owner server is invoked to change its state to `BUSY`. Then, the service time is computed from the distribution and an event denoting the end of the service is scheduled.
- **ENTITYFROMSERVICE:** The processing of an entity is terminated by this. The owner of this class is `SERVER`. After stopping the processing of the server, the *Execute* method performs two checks: first to communicate with the output node object to send the entity that has been just

processed, and then to query the input node object to receive the next entity to be processed.

- **ENTITYFROMSYSTEM**: Entities completing their system life are destroyed with the execution of this event. The execution invokes the owner **SINK** object which does the necessary statistical computations.

4.1.15 The MOVINGEVENT class

This denotes the events that are executed by **CARRIER**. To specify the two nodes of a transportation activity, it adds a pair of attributes to the definition of **SIMEVENT**.

Attributes

-*departurepoint*: Stores a pointer to the node from which **CARRIER** has departed.

-*arrivalpoint*: Stores a pointer to the node at which **CARRIER** is arriving.

Two event classes that are derived from **MOVINGEVENT** are:

- **BEGINTRANSPORT**: The beginning of a transportation process by a **CARRIER** is denoted by this event. When the execution of this event begins, the duration of the transportation is inquired from the owner carrier object. Then, the ending time of the process is computed and an event which denotes the stopping of the carrier is scheduled.
- **ENDTRANSPORT**: This event denotes the end of a transportation process. When such an event occurs, appropriate methods of the owner carrier object are called to stop the carrier and update the contents. Depending on the current position of the carrier, the contents can be unloaded or an empty carrier can be loaded. These processes are carried out by the send/receive mechanism that has been designed for the communication of objects.

4.1.16 The EVENTLIST class

This designed to keep linked lists of SIMEVENT objects. Since events are kept in *FELs* in chronological order, EVENTLIST objects maintain such a list. Instances of this class are used in the *FEL* of SIMULATION.

Attributes

-*owner*: Denotes the SIMULATION object which uses the EVENTLIST object as the *FEL* attribute.

-*head*: Points to the first SIMEVENT object in the list. (This is the next event to be processed.)

Methods

-*EventList*: Creates an empty EVENTLIST object.

-*GetHead*: Returns the SIMEVENT object that is pointed by *head*.

-*SetOwner*: Sets the value of *owner* by specifying a SIMULATION object.

-*Schedule*: Inserts a new event into the list while maintaining the chronological order.

-*GetNext*: Returns and removes the SIMEVENT object from the head of the list.

-*Exists*: Checks whether an event is in the list or not.

-*Print*: Prints the list of events that are currently in the list.

4.2 Communication between objects

The principles of OOP do not allow direct access to the attributes of an object by other objects; this is achieved by methods. Methods are functions returning the desired values about an object. The use of virtual functions expedite this task by providing a standardization of the names of the methods.

Methods are mostly used to return the attribute values of the objects. These attributes can be the names, id's, or status variables. Other methods (e.g., those which return an entity from the list of entities) are also used for the transfer of information. For our study, the traveling of objects among system objects of different types is the most important task to be performed.

	CanSend	CanReceive
QUEUE	at least one entity in the queue	TRUE
BUFFER	at least one entity in the buffer	capacity not reached
SOURCE	not used	not used
SERVER	not used	status is IDLE
CARRIER	carrier at unloader and not empty	carrier at loader and not full
SINK	not used	TRUE

Table 4.1. Return conditions for *CanSend* and *CanReceive*

For example, consider a QUEUE which needs to learn whether its output node can receive any entities. The decision to receive an entity is dependent on the class type of the next object and is different for BUFFER, SERVER, and CARRIER. Without any information about the class of the receiver, QUEUE simply calls the *CanReceive* method of the receiver. This virtual call is processed in the correct form with the help of dynamic binding. The result is an indication whether the output system object is able to receive an entity.

Navigation of objects in the system is carried out by the four virtual methods that have been defined in SIMOBJECT. These are *CanSend*, *CanReceive*, *RequestEntity*, and *ReceiveEntity*. The first two methods return TRUE/ FALSE to denote whether the indicated object can send/receive an object while the last two perform the necessary operations for the sending and the receiving of entities. In Table 4.1 we list the conditions that *CanSend* and *CanReceive* return TRUE for different types of system objects.

Now for a detailed description of the communication between objects, we study our classical single-queue/single-server example in detail. The phases of an entity during its travel from source to sink is illustrated in the following:

1. The execution of an ENTITYINTOSYSTEM generates the entity. The *Execute* method of this event, after scheduling the next ENTITYINTOSYSTEM, communicates with the output node of SOURCE as in Figure 4.1. The *outlink* of SOURCE has been specified as a QUEUE object by the *SetLinks* function. This object is returned by *GetOutLink* as the *next-object*.
2. Since *CanReceive* always returns TRUE for QUEUE, *ReceiveEntity* is called.

```

void EntityIntoSystem::Execute()
{
    SimObject *next_object;
    double nexttime;

    owner->SetPasser(object);
    nexttime=time+(owner->GetEntityInterval());
    sim->FEL.Schedule(new EntityIntoSystem
        (nexttime,owner,new Entity(nexttime)));
    if ((next_object=(owner->GetOutLink()))!=NULL)
    {
        if (next_object->CanReceive())
            next_object->ReceiveEntity(sim,time,object);
    }
}

```

Figure 4.1. *Execute* method of ENTITYINTOSYSTEM

This method schedules an ENTITYINTOQUEUE event to be executed immediately. The ENTITY to be inserted and QUEUE are kept in this event object. The ENTITYINTOQUEUE event execution handles the necessary operations to indicate the joining of the entity to the queue (Figure 4.2). As seen in the code fragment, upon inserting the entity into the queue, the method checks the next object, in this case a SERVER, to see whether it can send any entities. Assuming the server is IDLE at that moment, an ENTITYFROMQUEUE event is scheduled to be executed immediately.

3. The execution of the ENTITYFROMQUEUE event sends the ENTITY that is removed from the head of the QUEUE to the SERVER for processing (Figure 4.3).
4. The *ReceiveEntity* method of SERVER, when invoked, schedules an ENTITYINTOSERVICE event for processing. The execution of this event in turn invokes SERVER to start service and after retrieving the service time from the DISTRIBUTION object schedules an ENTITYFROMSERVICE event (Figure 4.4).
5. This ENTITYFROMSERVICE event, when executed, invokes the SERVER object to stop the service process. Then it performs two checks. First, it


```
void EntityIntoQueue::Execute()
{
    SimObject *next_object;

    owner->Insert(object,time);
    next_object=owner->GetOutLink();
    if (next_object!=NULL)
        if (next_object->CanReceive())
            sim->FEL.Schedule(new EntityFromQueue
                               (time,owner,NULL));
}
```

Figure 4.2. *Execute* method of ENTITYINTOQUEUE

```
void EntityFromQueue::Execute()
{
    SimObject *next_object;

    owner->SetPasser(owner->Remove(time));
    next_object=owner->GetOutLink();
    if (next_object!=NULL)
        if (next_object->CanReceive())
            next_object->ReceiveEntity
                (sim,time,owner->GetPasser());
}
```

Figure 4.3. *Execute* method of ENTITYFROMQUEUE

```
void EntityIntoService::Execute()
{
    double duration;

    owner->BeginService(object,time);
    duration=(owner->GetDist())->Generate();
    sim->FEL.Schedule(new
        EntityFromService(time+duration,owner,object));
}
```

Figure 4.4. *Execute* method of ENTITYINTOSERVICE

queries the output link to send the ENTITY that has been processed. Second, it communicates with the input link for further entities to process. This is illustrated in Figure 4.5.

The input link is the queue object in our case. Upon invocation of *RequestEntity*, QUEUE schedules an ENTITYFROMQUEUE object if there is at least one entity in the queue waiting for service. The execution continues from step 3. On the other hand, the *outlink* node of the SERVER is the SINK object and with the *ReceiveEntity* method it schedules an ENTITYFROMSYSTEM event.

6. The ENTITYFROMSYSTEM event invokes SINK to collect the statistics of the ENTITY leaving the system. The destruction of the ENTITY is also performed here (Figure 4.6).

The traveling of the entity from source to sink ends with the destruction of the entity. Events belonging to the processing of other entites can also be executed in between. These events are placed in the *FEL* in chronological order and the SIMULATION object processes them one by one.

4.3 User-defined behavior

The previous section has described procedures which have been designed to perform the operations of system objects in a single-queue/single-server system.

```
void EntityFromService::Execute()
{
    SimObject *prev_object,*next_object;

    owner->EndService(object,time);

    next_object=owner->GetOutLink();
    if (next_object!=NULL)
        if (next_object->CanReceive())
            next_object->ReceiveEntity(sim,time,object);

    prev_object=owner->GetInLink();
    if (prev_object!=NULL)
        if (prev_object->CanSend())
            prev_object->RequestEntity(sim,time);
}
```

Figure 4.5. *Execute* method of ENTITYFROMSERVICE

```
void EntityFromSystem::Execute()
{
    owner->TerminateObject(object,time);
    delete object;
}
```

Figure 4.6. *Execute* method of ENTITYFROMSYSTEM

But for most of the cases, the classes provided by SIMLIB may not be adequate. In such circumstances, the programmer should define classes or derive new classes from existing ones. In addition to objects, new event classes may be necessary to represent the behavior of these new objects.

In defining new classes there are some important points to consider. First, the place of the new class in the hierarchy should be taken into consideration. This place should be chosen to reuse most of the existing code and to minimize the amount of additional code. Rewriting of some methods can also be necessary. For example, the overriding of *CanSend*, *CanReceive*, etc. will facilitate the automatic navigation of entities through the system.

There are two ways of adding extra behavior to a derived class. The first is the definition of new methods. Functions to return the value of an attribute that is defined in the subclass are examples of this. The second method is the overriding of the existing methods inherited from the superclass. For example, the programmer can override the *StatPrint* method of the superclass to output the statistical data defined for the subclass.

For user-defined event classes, the definition of the *Execute* method is essential. This method carries out the actions to be performed when the time for the execution of the event comes. The relationship between the owner object and the event is to be considered for ease of implementation.

We now illustrate our ideas with an example. Consider the subsystem in Figure 4.7. Entities generated from the source arrive at the DECISIONNODE. At this point, the contents of the queues are compared and the entity just generated is sent to the queue having fewer number of entities. This helps to balance the load between two branches of the system.

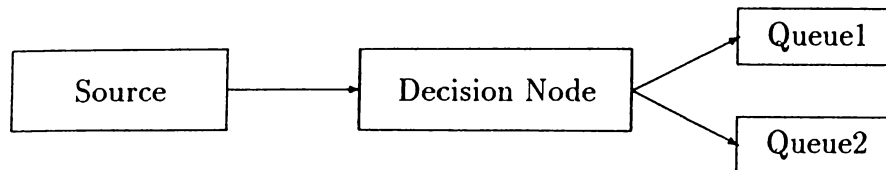


Figure 4.7. A subsystem with a decision node

In this system, the programmer needs to define a new class to represent the decision point. NODE can be used as the superclass of the new class. Calling the new class DECISIONNODE, the class definition of DECISIONNODE can be made (Figure 4.8).

In this definition, the additional attribute *totalentities* holds the number of entities that have passed through the node, and the queue variables represent the output links of the node. The methods *CanSend*, *ReceiveEntity*, and *RequestEntity* are overridden to fit the general structure of entity transfer between objects. There is no need for *CanSend* because the queues at the output of DECISIONNODE do not request entities by themselves.

```
class DecisionNode:public Node
{
protected:
    int totalentities;
    Queue *queue_1,*queue_2;
public:
    DecisionNode();
    int CanReceive();
    void ReceiveEntity(Simulation*,double,Entity*);
    void RequestEntity(Simulation*,double);
};
```

Figure 4.8. A possible definition for DECISIONNODE

Communication methods *CanSend* and *ReceiveEntity* can be written by imitating the same methods of another system object. In fact, *CanReceive* always returns TRUE, because there cannot be a condition to block the passing of entities through the node. The behaviour of *ReceiveEntity* differs slightly. At this point, the contents of the queues are checked and the ENTITY is sent to the resultant QUEUE. The methods for DECISIONNODE are listed in Figure 4.9.

To denote the arrival of an entity into the decision node, we define a new event class called ENTITYFORDECISION (Figure 4.10). This event is scheduled with the invocation of the *ReceiveEntity* method of DECISIONNODE by the SOURCE object.

Upon receiving the entities from the DECISIONNODE, the queues perform their standard actions by inserting the received entities. For the input, the DECISIONNODE is identified as the *outlink* of the SOURCE with a *SetLinks* call. As a result, the entities are transferred from SOURCE to DECISIONNODE with the *ReceiveEntity* method and after the decision, they are sent to the appropriate queue.

```
DecisionNode::DecisionNode(Queue* q1,Queue* q2):Node()
{
    total_entities=0;
    queue_1=q1;
    queue_2=q2;
    insert_to_objlist(this)
}

DecisionNode::CanReceive()
{
    return(TRUE);
}

DecisionNode::ReceiveEntity
    (Simulation* s,double t,Entity* e)
{
    SetPasser(e);
    ++totalentities;
    (s->FEL).Schedule(new EntityForDecision(t,this,e))
}

DecisionNode::RequestEntity(Simulation* s,double t)
{
    int load_1,load_2;

    load_1=queue_1->GetCount();
    load_2=queue_2->GetCount();

    if (load_1<=load_2)
        queue_1->ReceiveEntity(s,t,passerobject);
    else
        queue_2->ReceiveEntity(s,t,passerobject);

    SetPasser(NULL);
}
```

Figure 4.9. Method definitions for DECISIONNODE

```

class EntityForDecision:public StableEvent
{
public:
    EntityForDecision(){};
    EntityForDecision(double,SimObject*,Entity*);
    void Execute();
};

EntityForDecision::EntityForDecision
    (double t,SimObject* o,Entity* e)
{
    time=t;
    type=eEntityForDecision; //enumerated type
    owner=o;
    object=e;
}

void EntityForDecision::Execute()
{
    owner->RequestEntity(sim,time);
}

```

Figure 4.10. Method definitions for event class ENTITYFORDECISION

4.4 System services

For some operations like initializing the system or displaying the status and statistics of the objects, a list of all the system objects is necessary. Such a list, called *ObjList*, is initialized by the library and the created objects are inserted to it at the time of their construction. In Figure 4.9, the constructor of the DECISIONNODE includes a call *insert_to_objlist*. This is used to place the created instance in *ObjList*. The address of the instance, denoted by *this*, is passed as an argument to this function.

At this point, we want to emphasize the notion of *default constructors*. Default constructors are methods for a class requiring no parameters as the values of the attributes of the class and only provide the necessary memory allocation for the instances created by the program. The base class of the system objects, SIMOBJECT, has a constructor named *SimObject()* for this

purpose which sets the value of the *id* attribute to -1. This is to prevent the insertion of unused instances to the system list. Suppose a program declares an array of *QUEUE* objects of size 10 and uses, say, only 6 of these objects depending on the specifications coming from the user at run time. If all of the 10 *QUEUE* objects are inserted in *ObjList*, then after the execution the system will try to output statistical data for the unused 4 objects. Therefore the default constructor of *QUEUE*, inherited from *SimObject()*, sets the *id* values to -1 by the array declaration and objects having their *id* value as -1 are not inserted to *ObjList*. The insertion is performed by the *Assign* method of *QUEUE* while setting the values for the instances. *Assign* is the same as the constructor method except that it does not create the object but sets the values passed as arguments. Multiple calls to this method for changing the attribute values do not cause multiple insertions in the system list because there is a check before the insertion of every object.

The system-maintained list is used mainly for two purposes:

- **Initializing the system**

At the beginning of the execution of a program, there are normally no events to process in the event list of the simulation object. To execute *Step*, the *Init* method of the *SIMULATION* object inserts the initial events. These events are the *ENTITYINTOSYSTEM* instances for the source objects. *Init* invokes the *InitSources* function of the library for this task. This function traces *ObjList* to retrieve the objects with type *oSource* and schedules *ENTITYINTOSYSTEM* events for these objects. After the initialization the execution of events can begin. It is essential for a program to call *InitSources* from the *Init* method of its *SIMULATION* object. The default *Init* of *SIMULATION* performs this task but in case this method is overridden in a subclass of *SIMULATION* class, *Init* should be included in the overridden form. Another point to remember is the insertion of the constructed system objects into *ObjList*. The constructor of every system object class should include an *insert_to_objlist* function call.

- **Output of statistics**

After the execution of *Run*, *Done* is called to output the statistical values maintained by the system objects. This method displays the status and stopping time of the simulation and prints statistical information about

each system object. An example output is presented in Figure 3.9.

Done makes a call to *print_obj_stats* to display the data for each object. Each system class has a *StatPrint* method for this purpose. The *print_obj_stats* function iterates on *ObjList* and calls the *StatPrint* method of each system object in the list. The program should define a *StatPrint* method for each user-defined system object class.

If the user wants to output a subset of system objects or wants to direct the output to a file, this done by overriding *Done* and explicitly calling the *StatPrint* methods of the required objects.

Chapter 5

Three example systems

In this chapter, we describe the three prototype systems we have developed as the experimental components of our study. These systems were first implemented without SIMLIB. While the implementation of the library was in progress, the re-implementation of these systems was used as a testbed. The results provided a comparison of the procedural and object-oriented approaches. The three prototypes were a single-queue/single-server system, a production-line system, and an elevator system.

5.1 The single-queue/single-server example

This is the most basic example of queueing systems and can be described as follows (Figure 5.1):

- Entities arrive at the queue with respect to a statistical distribution.
- The first entity is immediately accepted by the server.
- The server receives entities from the queue and processes them. The duration of service is determined by another statistical distribution.
- Entities are forced to wait in the queue until they are removed by the server for processing.
- An entity leaving the server completes its system life.

The objects of the system are:



Figure 5.1. The single-queue/single-server model

- A source where the entities coming to the system are generated with respect to a statistical distribution of inter-arrival time.
- A queue where entities are forced to wait under a FIFO regime until the server is ready to provide service.
- A server where entities get processed after leaving the queue.
- A sink where entities leave the system.

The system-defined classes have been declared in the SIMLIB header file `simlib.h`. The user-defined class in the development of such an application with SIMLIB is a subclass derived from the `SIMULATION` class. It is necessary to override the *Init* method to declare the connections between the system objects and the *ShouldStop* method to denote the stopping condition of the execution. The simulation stops after 10000 entities have been processed by the server.

Other objects can be directly browsed from SIMLIB by declaring them with the use of their constructors. These objects are `SOURCE`, `QUEUE`, `SERVER`, and `SINK`, and `DISTRIBUTION` objects for the generation of inter-arrival time (for the source) and service time (for the server) distributions. The inter-arrival distribution is a normal distribution with mean 5 time units and variance 2 time units. The service time distribution of the server is again a normal distribution with mean 4 and variance 3. The server is not subject to crashes.

The *Init* method of the derived `QUEUESIMULATION` class is implemented by first calling the inherited method of `SIMULATION` class. This is necessary to force the `SOURCE` object to schedule the first `ENTITYINTOSYSTEM` event. Then the links for the navigation of entities are established by *SetLinks*.

ShouldStop is coded to return `TRUE` if the `SINK` has received 10000 entities. Another check is made on the *SimStatus* attribute for preventing programming

errors like generating an empty event list during the execution.

Finally the main module is implemented by consecutive calls to the methods of the `QUEUESIMULATION` instance `QS`. These are *Init* (for the setup), *Run* (for the execution), and *Done* (for displaying the results).

The code of the implementation is shown in Figure 5.2.

5.2 The production-line example

This is a modified version of the previous example. In this system, the queue-server pairs (with an additional queue at the output node of each server) are linked to each other and the transfer of entities is performed by carriers. The execution of the model is as follows:

- Entities enter the system from the source node and join the first input queue.
- The first server receives entities from this queue and services them.
- After service, the entities are placed in the output queue of the associated server.
- The conveyors carry the entities from the output queues to the input queue of the server in the line. In this way, the entities travel through the servers in the production line.
- A conveyor waits until it is loaded to full capacity before beginning the transportation. After unloading its contents at the input queue of a server, it goes back to the input queue of the preceding server for reloading.
- Entities are transferred to the sink by the last conveyor before they leave the system.

The model, with two servers, is illustrated in Figure 5.3.

The implementation is quite similar to the single-queue/ single-server case. The steps are:

- A subclass of `SIMULATION` is derived as `PLSIMULATION` for defining the initializations prior to the execution and the stopping condition of the run. The *Init* method is replaced by the *Setup* method which takes the

```

#include "simlib.h"

class QueueSimulation : public Simulation
{
public:
    QueueSimulation():Simulation(){};
    void Init();    //override for user-defined events
    int ShouldStop();
};

Distribution dist_arrival(dNormal,5.0,2.0);
Distribution dist_service(dNormal,4.0,3.0);
Source SO("Source",0,dist_arrival);
Queue QU("Queue",0);
Server SE("Server",0,dist_service); //no crashes
Sink SI("Sink",0);
QueueSimulation QS;

void QueueSimulation::Init()
{
    Simulation::Init();
    SO.SetLinks(SO,QU);
    QU.SetLinks(SO,SE);
    SE.SetLinks(QU,SI);
}

int QueueSimulation::ShouldStop()
{
    if ( (SI.total_entities>=10000) || (SimStatus!=SIM_OK) )
        return(TRUE);
    else
        return(FALSE);
}

int main()
{
    QS.Init();
    QS.Run();
    QS.Done();
}

```

Figure 5.2. The code of the single-queue/single-server example

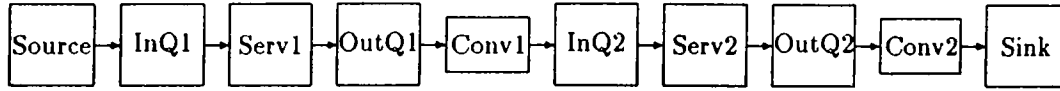


Figure 5.3. The production-line model with two servers

number of servers in the system as a parameter. This parameter can also be received from the user during the execution of the program.

- Next, the system objects and the distributions are declared. There are multiple instances of `QUEUE`, `CARRIER`, and `SERVER` so they are defined as arrays of objects. Their attribute values are not set until the *Assign* method is used.
- *Setup* first calls *Init* (of `SIMULATION`) for triggering the first `ENTITY-INTOSYSTEM` event. The next task is the setting of the values for the attributes of `QUEUE`, `CARRIER`, and `SERVER` which have been defined as arrays. The *Assign* method is used in a loop for this purpose. The final task is the connection of the system objects with *SetLinks* calls. This is also handled with the use of a loop iterating on the indices of the object arrays.
- In our example, *ShouldStop* is designed to stop the execution after 1000 time units.
- In the main body, *Setup*, *Run*, and *Done* methods of `PLSIMULATION` are invoked.

The code of a production-line system simulation with three servers is listed in Figure 5.4.

5.3 The elevator example

The elevator prototype simulates the working of an elevator that meets the requests coming from a number of different floors. A description of the model is as follows:

- Entities arrive at a floor with their destination known. Both the arrival

```

#include "simlib.h"
#include <stdio.h>

#define MAXSYS 10

class PLSimulation : public Simulation
{
public:
    void SetUp(int);
    int ShouldStop();
};

Distribution dist_arrival(dExponential,6.0);
Distribution dist_service(dNormal,5.0,3.0);
Source SO("Source",0,dist_arrival);
Queue QU[2*MAXSYS+1];
Carrier CR[MAXSYS+1];
Server SE[MAXSYS];
Sink SI("Sink",0);
PLSimulation PL;

void PLSimulation::SetUp(int length)
{
    int cnt;
    char name[20];

    Simulation::Init();
    for (cnt=0 ; cnt<length ; ++cnt)
    {
        sprintf(name,"Server-%d",cnt);
        SE[cnt].Assign(name,cnt,dist_service);
    }
    for (cnt=0 ; cnt<(2*length+1) ; ++cnt)
    {
        sprintf(name,"Queue-%d",cnt);
        QU[cnt].Assign(name,cnt);
    }
}

```

Figure 5.4. The code of the production-line example

```

    for (cnt=0 ; cnt<(length+1) ; ++cnt)
    {
        sprintf(name,"Carrier-%d",cnt);
        CR[cnt].Assign(name,cnt,3,20,4);
    }
    SO.SetLinks(SO,QU[0]);
    QU[0].SetLinks(SO,CR[0]);
    for (cnt=0 ; cnt<length ; ++cnt)
    {
        QU[2*cnt+1].SetLinks(CR[cnt],SE[cnt]);
        QU[2*cnt+2].SetLinks(SE[cnt],CR[cnt+1]);
        SE[cnt].SetLinks(QU[2*cnt+1],QU[2*cnt+2]);
        CR[cnt].SetLinks(QU[2*cnt],QU[2*cnt+1]);
    }
    CR[length].SetLinks(QU[2*length],SI);
}

int PLSimulation::ShouldStop()
{
    if ( (SimClock>=1000) || (SimStatus!=SIM_OK) )
        return(TRUE);
    else
        return(FALSE);
}

int main()
{
    PL.SetUp(2);
    PL.Run();
    PL.Done();
}

```

Figure 5.4. The code of the production-line example (cont'd)

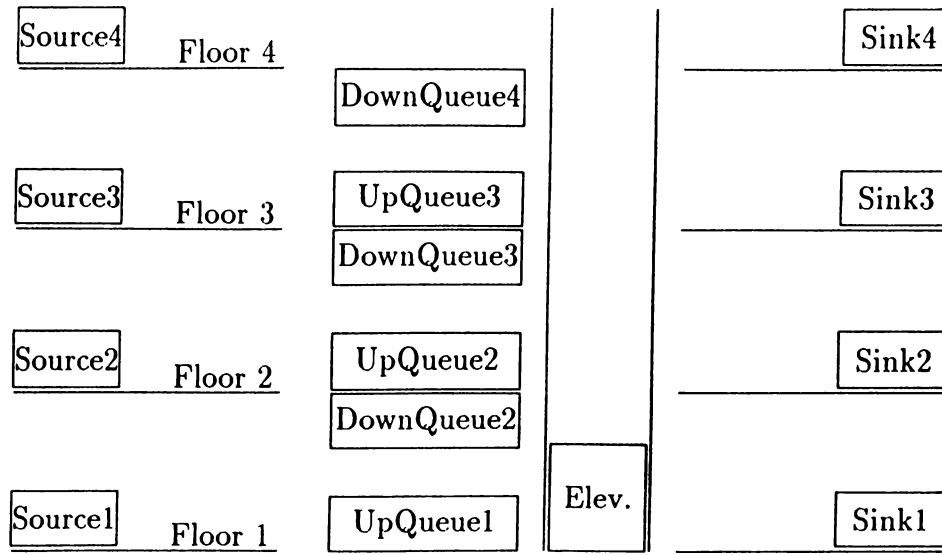


Figure 5.5. Model of the elevator system

floor and destination floor are random.

-There are two queues at each floor, one for entities that are waiting for the elevator to take them UP and one for those waiting to go DOWN. Depending on the values of the arrival and destination floors, an arriving entity joins the proper queue.

-The elevator is initially at the first floor. The first request is answered immediately. Other requests are handled by a decision manager. This decision manager orders the requests for optimum performance and determines the path of the elevator.

-When the elevator arrives at a floor to fulfil a request, it loads the entities that are in the queue associated with the current direction of the elevator.

-The route of the elevator is updated according to the destinations of the loaded entities. Each entity is transferred to its destination and leaves the system. During these operations, other requests coming from the floors are organized by the decision manager.

The system is illustrated in Figure 5.5.

The implementation requires some data structures that are not available in SIMLIB. Below we give a description of these:

- **ELEVENTITY**: The entities traveling between the floors of the system are represented by this class. Derived from **ENTITY**, this class has additional

attributes for the arrival and destination floor values of the entity under consideration.

- **ELEVATOR:** This represents the elevator object that transfers the **ELEV-ENTITY** objects. It inherits from **CARRIER** and has extra attributes for the current floor of the elevator, the time point that the elevator has most recently been at a floor, and the time units required to cover the distance between consecutive floors.
- **DECISIONMAKER:** The overall behavior of the **ELEVATOR** is governed by this. Two lists are kept for requests coming from the entities that wish to travel in **UP** and **DOWN** directions. These requests can be answered in two ways. If the elevator is currently in the same direction with a request, the request is processed immediately and the source floor of the request is taken into the route. Otherwise, the request is kept in the list and processed when the elevator completes the transfers in the current direction. The purpose of **DECISIONMAKER** is to maintain lists of incoming requests and determine the route of the elevator for maximum efficiency.

In addition to the system classes, the programmer needs to define new event classes to represent the state changes. **ENTITYINTOSYSTEM** generates objects of type **ENTITY**. On the other hand, the elevator system needs **ELEVENTITY** objects. Therefore, a new class **NEWENTITYTOELEVATOR** is derived.

The events caused by the movement of **ELEVATOR** necessitate the derivation of class **ELEVENT**, a subclass of **SIMEVENT**. This class has an additional attribute to denote the floor where the event occurs. **ELEVENT** is the parent class of the three other classes that actually represent the events of the system:

-**ELEVSTARTSERVICE:** Denotes the start of service for entities that are waiting at a floor. With the execution of this event, these entities are loaded into the elevator and their destination floors are added to the route.

-**ELEVENDSERVICE:** Denotes the end of service for the entities in the elevator that have destinations as the current floor of the elevator. The execution of this event sends these entities to the **SINK** of the current floor.

-**ELEVATFLOOR:** This denotes that the elevator is at the floor denoted by the *where* attribute of **ELEVENT**. It is used to keep track of the elevator.

Similar to the preceding examples, a class `ELEVSIMULATION` is derived from `SIMULATION` to control the execution. This class has a *SetUp* method which constructs the model with a parameter denoting the number of floors.

The basic execution of the implementation can now be stated:

1. *SetUp* forces the scheduling of the first `NEWENTITYTOELEVATOR` event for each floor.
2. The execution of this event places the generated entity in the proper queue of the floor and a request is sent to `DECISIONMAKER` if necessary.
3. Upon receiving the request, `DECISIONMAKER` examines the status of the elevator and determines whether the request can be answered during the current tour of the elevator. If so, the necessary `ELEVATFLOOR` and `ELEVSTARTSERVICE` events are scheduled. Otherwise, the request is inserted to the list of waiting requests.
4. The execution of a `ELEVSTARTSERVICE` loads the waiting entities from the queues into the `COLLECTION` object of the `ELEVATOR`. Using the destination floors of these entities, the corresponding `ELEVENDSERVICE` events are scheduled. The `ELEVENDSERVICE` events unload the entities when the destination floor is reached. These entities leave the system after the gathering of necessary statistical data.
5. If the elevator runs out of `ELEVEVENT` events, it asks the `DECISIONMAKER` to provide the new route by processing the waiting requests. At this moment, it switches direction to answer these requests.

The code is rather long (about 1300 lines) to include here. So we only present the *Execute* methods of `ELEVENDSERVICE` and `NEWENTITYTOELEVATOR` to give an idea of the extra behavior defined by the programmer (Figures 5.6 and 5.7).

```

void ElevEndService::Execute()
{
    Entity* temp;
    Collection TC;

    ((Elevator*)owner)->currentfloor=where;
    ((Elevator*)owner)->lasttime=time;

    while ((temp=((Elevator*)owner)->
                GetContents()->Remove())!=NULL)
    {
        if (((ElevEntity*)temp)->targetfloor==where)
            sim->FEL.Schedule(new EntityFromSystem
                (time,&Sinks[where],temp));
                // terminate the entities who have
                // reached their destination
        else
            TC.Insert(temp); // store others in a temporary list
    }

                // now restore the contents
    while ((temp=TC.Remove())!=NULL)
        ((Elevator*)owner)->GetContents()->Insert(temp);

    if (nonextfloor()) //get the new route
        DM.ask();
}

```

Figure 5.6. The *Execute* method of ELEVENDSERVICE

```

void NewEntityToElevator::Execute()
{
    double nexttime;
    int so,ta;
    ElevEntity* elevobject;

    owner->SetPasser(object);
    elevobject=((ElevEntity*)object);
    if (Elev.GetStatus()==IDLE)
        Elev.lasttime=time;

        // analyze the floors and generate
        // the request
    if ( (elevobject->sourcefloor) > (elevobject->targetfloor) )
    {
        sim->FEL.schedule
            (new EntityIntoQueue
             (time,&DownQueues[owner->GetId()],object));
        if (DownQueues[owner->GetId()].IsEmpty())
            DM.request(time,elevobject->sourcefloor,DOWN);
    }
    else
    {
        sim->FEL.schedule(new EntityIntoQueue
            (time,&UpQueues[owner->GetId()],object));
        if (UpQueues[owner->GetId()].IsEmpty())
            DM.request(time,elevobject->sourcefloor,UP);
    }

        // schedule the next event with
        // random floors
    nexttime=time+(owner->GetEntityInterval());
    so=owner->GetId();
    ta=(int)(myrand()*floorcount);
    while (so==ta)
        ta=(int)(myrand()*floorcount);
    sim->FEL.schedule(new NewEntityToElevator
        (nexttime,owner,new ElevEntity(nexttime,so,ta)));
}

```

Figure 5.7. The *Execute* method of NEWENTITYTOELEVATOR

Chapter 6

Conclusion

OOP, relying on the concept of “objects,” provides the system designer with techniques emphasizing the abstraction of program modules. These modules are mapped to data structures that encapsulate attributes and behavior of system objects.

OOP is useful in the development of simulation software. A model of any complexity can be divided into components which can be represented as objects with clearly defined interfaces. Based on this idea, a library that provides the general data structures of a simulation program allows the development of reusable code. Such a library significantly reduces coding time.

The research presented in this thesis has investigated the use of the object-oriented paradigm in the area of simulation code development. A prototype library, SIMLIB, has been designed and coded. SIMLIB provides a set of system objects that can be used to represent the physical components of a model, and a set of event objects to describe the state changes in the system. Using SIMLIB, one can define new system objects and event classes with a small amount of code. With proper declarations for a particular system, the simulation process executes the events one-by-one in chronological order to reflect the state of the system.

The classes defined in SIMLIB have been devised from three “classical” discrete event simulation examples developed prior to the design of the library. These examples have been studied to extract the basic execution principles

and data structures of a simulation application. Later these systems have been re-implemented with SIMLIB, for an overall evaluation of our approach.

The essence of a class library lies in the definition of its classes and their methods. In the design of the overall system, an important point is the communication between the objects of an application. A clear protocol defined in terms of messages (methods) is used for inter-object communication.

The development of simulation software with SIMLIB consists mainly of the declaration of system objects and events of the model at hand. Whether system-defined or user-defined, these objects provide the mapping of the model to code. Physical links for the navigation of entities through the system are also defined. Execution of the model is handled by control classes. Once the components of the model are defined, other tasks like initializing the system with entity-generation events and statistical data collection are performed by the built-in functions of the library.

A comparison of the amount of code written for the three prototypes is given in Table 6.1. Clearly, the single-queue/single-server and the production line examples have been reduced to very small programs with the use of SIMLIB. Basically, these systems do not need extra data structures and all the components of their models can be represented by the classes browsed from SIMLIB. This argument does not entirely hold for the elevator example as this requires a number of user-defined classes for the representation of the system. A great portion of the code for this example is the implementation of the decision maker module which is responsible for keeping track of the requests and routing the elevator.

	Straightforward implementation	SIMLIB implementation	Reduction (%)
single-queue/single-server	170	40	0.76
production-line	1280	70	0.95
elevator	1300	700	0.46

Table 6.1. Comparison of code length (number of lines) of the three prototypes implemented with and without SIMLIB

The results indicate that SIMLIB does reduce the amount of code for systems that employ standard system objects like queues, servers, etc. but cannot achieve the same effect for more complex models. In fact, SIMLIB is a very small library supporting a minimal set of simulation objects. The main idea is the representation of a model with objects mapped to the components of the model. Once the general principles for the execution of a model are provided by this philosophy, a more adequate set of objects can be defined and included in SIMLIB.

Although we have chosen to focus on the development of classical network classes, it is clear that a wide range of classes could be designed to facilitate the development of simulation models for specific system types. For example, it would be relatively easy to design classes to represent the components of a communication system. We again emphasize the fact that SIMLIB allows the programmer to choose model components from an expanding library of already defined classes or construct his/her own model components, possibly derived from the library classes. The OOP philosophy encourages an evolutionary model building process in which code is reused and continuously upgraded.

SIMLIB promises a simulation methodology worthy of serious consideration for the future of computer simulation. The library is compatible with one of the most popular languages available today, C, and a language with growing popularity, C++.

Finally, we state three potential research areas related to our study:

- **Visual modeling:** OOP is seen to be a promising software methodology with its facilities to allow the simulation programmer to build applications in a graphical environment. The framework, when equipped with the user friendly interface and animations, might enhance the quality of a program while reducing the efforts for model building. Basic results of the simulation can be extracted by animation and after the execution output reports provide a convenient way of representing the data as useful information. Such environments have also facilities for the management of the large number of classes presented in them.
- **Integration of different techniques:** Various methods have been used

for the computer representation of a simulation model. There is an increasing use of AI techniques for modeling knowledge representation. Problem solving methods allow the application of different AI techniques for solving modeling and simulation tasks. For example, consider a system which keeps all the models developed by the users for future use. This *model base* may be designed to reduce the model development time with its capabilities for storing and retrieving the models. To select a model for an application, a rule-based system which analyzes the stored models can help.

- **Parallel simulation:** Object-oriented simulations can be executed in parallel on a multiprocessor which allows greater processing and memory resources to be applied to a given problem in a cost-effective manner. Object-oriented simulations are naturally suited to running on multiprocessors because the components of a model represent logically distinct threads of execution on a separate processor. Simulation class libraries can be enhanced by features that support parallel simulation like distributed I/O facilities, performance analysis tools, and run-time mapping of entities to processors.

Appendix A

Class declarations in SIMLIB

SIMLIB library can be found in `/home/usr3/isikli/tez/simlib/source`. The library comprises four source files: `sim_simul.c`, `sim_collect.c`, `sim_object.c`, and `sim_event.c`.

The three prototypes described in Chapter 5 are also in this directory. The filenames are `qs.c` for the single-queue/single-server example, `pl.c` for the production-line example, and `elev.c` for the elevator example.

The classes provided by SIMLIB are declared in `simlib.h` which should be included by every application that uses the library. In the following, we present the listing of this header file.

```
// simlib.h, April 93

#include "fstream.h"

// constant declarations

#define TRUE 1
#define FALSE 0

#define IDLE 0
#define BUSY 1

#define SIM_OK 0
#define SIM_FEL_EMPTY 1

#define eEntityIntoSystem 0
#define eEntityIntoQueue 1
#define eEntityFromQueue 2
```

```

#define eEntityIntoService 3
#define eEntityFromService 4
#define eEntityFromSystem 5
#define eBeginTransport 6
#define eEndTransport 7

// distribution and object type enumerations

enum Dtype {dConstant,dNormal,dExponential};
enum Otype {oNode,oSink,oQueue,oServer,oBuffer,oSource,oCarrier};

void insert_to_objlist(SimObject*);
void print_obj_list(ostream&);
void InitSources(Simulation*);
void print_obj_stats(ostream&,double);
double myrand();

class SimEvent;
class Simulation;
class Distribution;
class Node;
class EventList;
class Collection;

class Entity
{
public:
    double entrytime;
    double totalwaittime;
    double lastwait;
    Entity* next;

    Entity();
    Entity(double);
    void BeginWaiting(double);
    double EndWaiting(double);
    virtual void Print();
};

class SimObject
{
protected:
    Otype type;
    char* name;
    int status;
    int id;
    SimObject *next;
    SimObject *inlink;
    SimObject* outlink;

    friend SimEvent;
    friend StableEvent;
    friend MovingEvent;

```

```

public:
    SimObject();
    int GetStatus();
    void SetStatus(int);
    int GetId();
    Otype GetType();
    void SetNext(SimObject*);
    SimObject* GetNext();
    char* GetName();
    SimObject* GetInLink();
    SimObject* GetOutLink();
    virtual void SetLinks(SimObject&,SimObject&);
    virtual void IncTotalWait(double){};
    virtual void SetPasser(Entity*){};
    virtual Entity* GetPasser(){return(NULL);}
    virtual double GetEntityInterval(){return(0.0);}
    virtual Distribution* GetDist(){return(NULL);}
    virtual void Insert(Entity*,double){};
    virtual Entity* Remove(double){return(NULL);}
    virtual void Insert(Entity*){};
    virtual Entity* Remove(){return(NULL);}
    virtual void BeginService(Entity*,double){};
    virtual void EndService(Entity*,double){};
    virtual void TerminateObject(Entity*,double){};
    virtual void ReceiveEntity(Simulation*,double,Entity*){};
    virtual void RequestEntity(Simulation*,double){};
    virtual int CanSend(){return(0);}
    virtual int CanReceive(){return(0);}
    void Print(ostream&);
    virtual void StatPrint(ostream&,double){};
    virtual double Move(double,SimObject*,SimObject*){return(0.0);}
    virtual void Stop(double,SimObject*){};
};

class Carrier:public SimObject
{
protected:
    Collection* contents;
    int capacity;
    int entitiescarried;
    double distance;
    double speed;
    double totalwaittime;
    double lastwait;
    SimObject* from;
    SimObject* to;

public:
    Carrier(){};
    Carrier(char*,int,int,double,double);
    void Assign(char*,int,int,double,double);
    Collection* GetContents();
    int CanReceive();
    int CanSend();

```

```

    void ReceiveEntity(Simulation*,double,Entity*);
    void RequestEntity(Simulation*,double);
    void BeginWaiting(double);
    void EndWaiting(double);
    double Move(double,SimObject*,SimObject*);
    void Stop(double,SimObject*);
    void SetLinks(SimObject&,SimObject&);
    void StatPrint(ostream&,double);
};

class Node:public SimObject
{
protected:
    Entity* passerobject;

public:
    Node();
    void SetPasser(Entity*);
    Entity* GetPasser();
};

class Collection:public Node
{
protected:
    Entity* head;
    int count;
    int totalentries;
    int maxentries;
    double totalwait;

public:
    Collection();
    void Insert(Entity*,double);
    Entity* Remove(double);
    void Insert(Entity*);
    Entity* Remove();
    int IsEmpty();
    int GetCount();
    void Print();
    double GetTotalWait();
    void BeginWaiting(double);
    void EndWaiting(double);
    void IncTotalWait(double);
    void UpdateTotalWait(double);
};

class Queue:public Collection
{
public:
    Queue(){};
    Queue(char*,int);
    void Assign(char*,int);
    Entity* Remove(double);
    void ReceiveEntity(Simulation*,double,Entity*);
    void RequestEntity(Simulation*,double);
};

```

```

        int CanSend();
        int CanReceive();
        void StatPrint(ostream&,double);
};

class Buffer:public Queue
{
    int buffersize;

public:
    Buffer(int);
    void Assign(int);
    void Insert(Entity*,double);
};

class Distribution
{
    Dtype type;
    double mean;
    double deviation;

public:
    Distribution();
    Distribution(Dtype,double);
    Distribution(Dtype,double,double);
    double Generate();
    void Print(ostream& s);
};

class Source:public Node
{
    Distribution *distgeneration;
    int totalentities;

public:
    Source(){};
    Source(char*,int,Distribution&);
    void Assign(char*,int,Distribution&);
    Distribution* GetDist();
    double GetEntityInterval();
    void SetPasser(Entity*);
    void StatPrint(ostream&,double);
};

class Server:public Node
{
    int totalentities;
    double totalidletime;
    double lastwait;

public:
    Distribution *distservice;
    Distribution *distrepair;
    double probcrash;
};

```

```

    Server(){};
    Server(char*,int,Distribution&,Distribution&,double);
    Server(char*,int,Distribution&);
    void Assign(char*,int,Distribution&,Distribution&,double);
    void Assign(char*,int,Distribution&);
    void BeginService(Entity*,double);
    void EndService(Entity*,double);
    int IsIdle();
    Distribution* GetDist();
    int CanReceive();
    void ReceiveEntity(Simulation*,double,Entity*);
    void StatPrint(ostream&,double);
};

class Sink:public Node
{
public:
    int totalentities;
    double totalwaittime;
    double totalsystemtime;

    Sink(){};
    Sink(char*,int);
    void Assign(char*,int);
    void TerminateObject(Entity*,double);
    int CanReceive();
    void ReceiveEntity(Simulation*,double,Entity*);
    void StatPrint(ostream&,double);
};

class SimEvent
{
public:
    double time;
    int type;
    SimObject* owner;
    SimEvent* next;
    Simulation* sim;

    friend EventList;
    friend Simulation;

    virtual void Print(ostream&){};
    virtual void Execute(){};
    virtual void SetSim(Simulation*);
    double GetTime();
};

class StableEvent:public SimEvent
{
protected:
    Entity *object;
    friend EventList;
};

```

```

public:
    void Print(ostream&);
};

class MovingEvent:public SimEvent
{
protected:
    SimObject *departurepoint;
    SimObject *arrivalpoint;

public:
    void Print(ostream&);
};

class BeginTransport:public MovingEvent
{
public:
    BeginTransport(double,SimObject*,SimObject*,SimObject*);
    void Execute();
};

class EndTransport:public MovingEvent
{
public:
    EndTransport(double,SimObject*,SimObject*,SimObject*);
    void Execute();
};

class EntityIntoSystem:public StableEvent
{
public:
    EntityIntoSystem(double,SimObject*,Entity*);
    void Execute();
};

class EntityIntoQueue:public StableEvent
{
public:
    EntityIntoQueue(double,SimObject*,Entity*);
    void Execute();
};

class EntityFromQueue:public StableEvent
{
public:
    EntityFromQueue(double,SimObject*,Entity*);
    void Execute();
};

class EntityIntoService:public StableEvent
{
public:
    EntityIntoService(double,SimObject*,Entity*);

```



```

        void Execute();
};

class EntityFromService:public StableEvent
{
public:
    EntityFromService(double,SimObject*,Entity*);
    void Execute();
};

class EntityFromSystem:public StableEvent
{
public:
    EntityFromSystem(double,SimObject*,Entity*);
    void Execute();
};

class EventList
{
    Simulation* owner;
    SimEvent* head;

public:
    EventList();
    SimEvent* GetHead();
    void SetOwner(Simulation*);
    void Schedule(SimEvent*);
    SimEvent* GetNext();
    void Print(ostream&);
    int Exists(SimEvent*);
};

class Simulation
{
public:
    double SimClock;
    EventList FEL;
    int SimStatus;

    friend SimEvent;

    Simulation();
    virtual void Init();
    virtual void Step();
    virtual void Run();
    virtual void Done();
    virtual int ShouldStop();
};

```

References

- [1] J. Banks and J.S. Carson. *Discrete-Event System Simulation*. Prentice-Hall, 1984.
- [2] R. Belanger. MODSIM II, a modular, object-oriented language. In O. Balci, R.E. Sadowski, and R.E. Nance, editors, *Proceedings of the 1990 Winter Simulation Conference*, pages 118–122, 1990.
- [3] J. Bezivin. Some experiments in object-oriented simulation. In *OOPSLA Proceedings*, pages 394–405, 1987.
- [4] E.L. Blair and S. Selvaraj. DISC++: a C++ based library for object-oriented simulation. In E.A. MacNair, K.J. Musselman, and P. Heidelberger, editors, *Proceedings of the 1989 Winter Simulation Conference*, pages 301–307, 1989.
- [5] B. Cox. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley, 1986.
- [6] D.L. Eldredge, J.D. McGregor, and M.K. Summers. Applying the object-oriented paradigm to discrete-event simulations using the C++ language. *Simulation*, 54(2):83–91, 1990.
- [7] C.B. Basnet et al. Experiences in developing an object-oriented modeling environment for manufacturing systems. In O. Balci, R.E. Sadowski, and R. E. Nance, editors, *Proceedings of the 1990 Winter Simulation Conference*, pages 477–481, 1990.
- [8] A. Goldberg and D. Robson. *Smalltalk-80: The Language*. Addison-Wesley, 1989.

- [9] A.R. Kaylan. A simulation environment based on object-oriented technology. In A.R. Kaylan and T. Ören, editors, *Proceedings of Advances in Simulation '92 Symposium*, pages 163–174, 1992.
- [10] G. Lomow and D. Baezner. A tutorial introduction to object-oriented simulation and SIM++. In O. Balci, R.E. Sadowski, and R.E. Nance, editors, *Proceedings of the 1990 Winter Simulation Conference*, pages 149–153, 1990.
- [11] J.M. Mellichamp and A.F.A. Wahab. An expert system for FMS design. *Simulation*, 48(5):201–209, 1987.
- [12] B. Meyer. Reusability: The case for object-oriented design. *IEEE Software*, 4(2):50–64, 1987.
- [13] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.
- [14] J.G. Moser. Integration of artificial intelligence and simulation in a comprehensive decision-support system. *Simulation*, 47(6):223–229, 1986.
- [15] R. O’Keefe. Simulation and expert systems—A taxonomy and some examples. *Simulation*, 46(1):10–16, 1986.
- [16] T.I. Ören and B.P. Zeigler. Artificial intelligence in modeling and simulation: Directions to explore. *Simulation*, 48(4):131–134, 1987.
- [17] C.D. Pegden. *Introduction to SIMAN*. Systems Modeling Corporation, 1982.
- [18] R.I. Phelps. Artificial intelligence—An overview of similarities with OR. *Journal of the Operations Research Society*, 37(1):13–20, 1987.
- [19] M. Pidd, editor. *Computer Modeling for Discrete Simulation*. John Wiley, 1989.
- [20] A.B.B. Pritsker. *The GASP Simulation Language*. John Wiley, 1974.
- [21] A.B.B. Pritsker and C.D. Pegden. *Introduction to Simulation and SLAM II*. John Wiley, 1986.
- [22] E.C. Russell. *Building Simulation Models with SIMSCRIPT II.5*. CACI Products Company, 1983.

- [23] T.J. Schriber. *Simulation Using GPSS*. John Wiley, 1974.
- [24] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.
- [25] B.P. Zeigler. Hierarchical, modular discrete-event modeling in an object-oriented environment. *Simulation*, 49(5):219–230, 1987.