THE UNIVERSITY OF CALGARY

Object-Oriented Simulation for Queueing Systems

by

Xiaoming Li

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

April, 1997

Abstract

Many discrete event simulation studies involve the simulation of queueing systems. Two major problems arise in the simulation of queueing systems: first, the modeling of a queueing system can be an extremely complex and error-fraught endeavor; and second, the simulation of the system can be computationally intensive.

This thesis addresses these two problems from the aspects of modeling effectiveness and execution efficiency. An object-oriented (OO) simulation package called QueKit has been developed in order to facilitate the modeling process and retain good efficiency in both sequential and parallel executions. This package provides an OO event-driven modeling framework. It allows the emphasis on modeling the objects that provide services in a queueing system (server architecture) and the emphasis on modeling the objects that need services (client architecture) to be applied seamlessly in a single simulation model. This server-client architecture enables the application programmer to adjust the balance between modeling effectiveness and execution efficiency when developing a simulation model for sequential execution. It is expected to enable more natural parallelism to be exploited from the real system when developing a simulation model for parallel execution. This may result in better modeling effectiveness as well as better execution efficiency than either the server architecture or the client architecture solely applied in a model when running on a parallel machine.

Acknowledgments

I would like to thank my supervisor, Dr. Brian Unger, for his continuous support and guidance during my research work. His patience and concerns gave me a constant encouragement in my research endeavor even from New Zealand via e-mail when he was on his sabbatical leave last year. Dr. Brian Unger made an enormous contribution to this research and thesis. His insights and suggestions made a substantial improvement to the literary quality of this thesis.

Special thanks go to my lovely husband Zhong Chen. Thanks for his accompanying me to the school when I worked on my thesis in many evenings. I would not finish this work without his love and support. I would also like to thank my parents and my brother for their great concerns from a remote place via phone calls and mails.

I'd like to thank my colleagues and friends, Fabian Gomes, Zhong-e Xiao, Steve Franks, Husam Kinawi, and Jya-Jang Tsai for the helpful discussion with them. Especially, both encouragement and criticism from Dr. Fabian Gomes were very valuable to my work.

Thanks to Raimar Thudt for his support of this research, and his many suggestions regarding to the common modeling problems and solutions.

I'd like to thank Darcy Grant, Robert Fridman, Wayne Pearson, Brian Scowcroft, Gerald Vaselenak, and Mark Stadel for their technical assistance.

Dedication

To my father, Shufu Li,
my mother, Qunfang Wu,
and
my husband, Zhong Chen.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Simulation is a methodology for imitating the operation of a real-world system (or process) over time. A simulation involves the generation of an artificial history of the state of a system over time in order to draw inferences concerning the operating characteristics of that system. Simulation is widely used for analysis, performance evaluation, tests of sensitivity and cost effectiveness, forecasting, training, and decision making. In fact, simulation can be used for analyzing any model of arbitrary complexity. The complexity of the model is limited only by the modeler's ability, the power of the method to represent the system under study, and the capacity of a computer to load and run the simulation program. Experiments with the real system are often out of the question due to the high cost or potential danger. Analytical methods are usually limited to relatively simple systems so that they cannot be used for analyzing some complex systems which are mathematically intractable. Simulation can be used for studying any system whose behavior can be described in terms of a model. In contrast to the *continuous simulation* that studies a system in which the system state changes continuously over time, *discrete event simulation* (DES) is the simulation of a system in which the system state changes only at discrete points of time. Discrete event simulation has come to be an important methodology for understanding and predicting the behaviors of many complex systems. *Parallel discrete event simulation* (PDES) refers to the execution of a DES model on a parallel computer.

1.1 Discrete Event Simulation and Parallel Discrete Event Simulation

From the view of a DES modeler, a *system* is a collection of entities that interact with each other over time to accomplish one or more goals. Each *entity* in a system is an element which may require explicit representation in a simulation model. An *attribute* is a property of an entity. For instance, patients, beds, and theaters may be three kinds of enti-

ties in a hospital system. The patient's age, gender, the date of entering the hospital, etc., may be attributes of the patient.

An entity may be *dynamic* in the sense that it may dynamically enter a system, may be dynamically destroyed when it leaves the system, and may move through the system to get service from other entities. An entity may be *static* in the sense that it may reside in a system from the beginning and it provides services for others. Thus in the literature, the former entity is often called a *dynamic entity* because of its transient nature whereas the latter is often called a *static entity* because of its permanent nature [6]. In the hospital system, a patient is a dynamic entity. S/he may require a bed first when s/he enters the hospital, acquire a theater to have an operation if there is one available, return the theater after the operation. Then s/he may stay in the hospital for some time, and finally return the bed and leave the hospital system. Beds and theaters are two kinds of static entities which provide services for patient entities such as staying in the hospital and having an operation respectively. A static entity sometimes is also referred to as a resource which represents one or several identical entities which provide service in the system.

The behavior of a system as it evolves over time is studied by developing a simulation model. *A simulation model* is an abstract representation of a system, usually containing structural, logical or mathematical relationships which describe the system in terms of system state, entities and their attributes, and so on. *System state* is a collection of variables that contain all the information necessary to describe the system at any instant in time.

A DES model assumes the simulated system only changes its state at discrete points of time. The simulation model jumps from one state to another upon the occurrence of each event. An *event* is an instantaneous occurrence that may change the state of a system [3], e.g., an arrival of a new patient in a hospital system is an event. This event may cause the following changes for the system state: the total number of the patients in the system is increased by one; the number of bed resource may be decreased by one if a bed is assigned to the newly arrived patient. An *event notice* is a record of an event in a model. It contains

all the information about the event which is to occur at the current or future time along with any associated data necessary to process the event. At a minimum, the record includes the event type and the event time. The event time corresponds to the actual time in the real system when the corresponding event would occur. An event notice is often simply called an event (which means an event object or event record) in a simulation model. An *activity* is a period of time between two distinct events. One event marks the beginning of the activity and another marks the end of the activity. For example, in the hospital system, an operation may be an activity which starts with the "beginning of operation" event and ends with the "end of operation" event.

During a discrete event simulation, a model changes its state at discrete points of time upon the occurrence of events. The events are scheduled in chronological order on a list called the *event list* (or *future event list*) so that the causality in a simulation is maintained. The *causality* in a simulation is the rule that some changes of the system state must happen before others so that the simulation mimics the modeled system correctly [13]. That is, the order of processing events in a simulation cannot cause a scenario that future affects the past. The violation of this rule will cause the errors in a simulation, and errors of this type are called *causality errors*. It is obvious that causality errors in a simulation will never happen when events are processed in a chronological order in sequential execution. However, this may not be true for a simulation on a parallel computer because events cannot be processed in a chronological order on different processors unless special hardware and/or software mechanisms are provided to ensure the causality is maintained.

With the development of distributed and shared memory multiprocessor systems, great progresses in techniques for efficiently executing DES on distributed and parallel machines has been made for over a decade. Parallel discrete event simulation (PDES) has attracted great interest in recent years because it has exposed a lot of challenges for people to achieve high efficiency of DES on distributed and parallel computers.

In PDES, the logical process view is a dominant modeling framework. With the LP view, a system to be modeled is usually viewed as a physical system which is composed of some number of physical processes that interact at various points in simulated time [15]. A *physical process* (PP) is an element that performs a certain logical function in the real system, e.g., a theater may be a PP in the above hospital system. Each PP in the system is mapped to a *logical process* (LP) in a model. An LP is a software object or a procedure that mimics the behavior of a PP in the modeled system. The LPs in a PDES model are assigned to different processors on a parallel computer to be processed concurrently in order to speed up the simulation. All interactions between PPs are modeled by timestamped messages sent between the corresponding LPs which may reside on different processors. A *message* is a timestamped construct in PDES that carries an event notice passing from one LP to another to model the interactions between two LPs. A message is sometimes also called an *event* or *event message* when it is on an event list. Messages in the LP view are used either to synchronize the actions of two LPs or to pass information from one LP to another.

The synchronization of LPs in a model is concerned with ensuring the causality is maintained in PDES. That is, the correct order of processing events in PDES is ensured in order to yield the same results as the sequential execution in which events are processed in a chronological order. Synchronization algorithms in PDES are broadly fall into two categories: conservative and optimistic. Conservative approaches strictly avoid the possibility of any causality error (i.e., future affects the past) ever occurring. On the other hand, optimistic approaches use detection and recovery mechanism: every LP progresses at its own speed without concern of causing causality error. However, if causality errors are detected, a rollback mechanism is invoked to recover [15].

1.2 Object-Oriented Modeling and Simulation

The object-oriented (OO) approach has been probably associated with discrete event simulation from the very beginning. Many people consider Simula [14], one of the first

simulation languages, as the first real OO programming language. The OO approach has become an important methodology in the software development since 1990s [8]. *OO modeling* is a way of thinking about problems using models composed of software objects that represent real-world objects. A model is built by organizing a collection of these software objects that incorporate both data structure and algorithmically defined behavior. *OO simulation* uses this kind of model to conduct experiments in order to study the dynamics of the modeled system. The key contribution of OO methodology for simulation is the mapping between real-world objects and software objects [13].

OO modeling and simulation are closely related to OO programming (OOP) [34]. OOP is a design and programming discipline that focuses on the objects (i.e., software objects) rather than functions that make up the software system. In OOP, an *object* is a distinguishable component of a program while a *class* is a template for a group of objects that have the same characteristics. An object has a set of attributes (i.e., data) that define its state and a set of interface functions (i.e., methods) for accessing the state. OOP focuses first on identifying objects that make up the software system. The classes and interfaces are then defined and implemented. A program is finally written for creating and manipulating the objects through their interfaces.

Similarly, OO modeling and simulation involves identifying physical processes (PPs) that make up the system to be modeled, and mapping these PPs into object classes. The methods are written for these classes in order to present the PPs' behaviors including interactions. Then all circumstances that can lead to changes in the state of the system are identified and characterized as events. These events are tied to simulation time by means of their scheduled event time. Finally, a program is written for creating and manipulating these objects and events along the progress of the simulation time, and it is executed in order to get the simulation results [22].

OO programming provides convenient facilities for software development using the concepts of encapsulation, inheritance, and polymorphism. *Encapsulation* allows wrap-

ping all data and functions together inside a class and protects them from any unauthorized access. Encapsulation permits a simulation package to keep all data and operations in a safe way so that the detailed information of the package is hidden from the user. It also promotes modular design of a simulation package. *Inheritance* allows a new class (child class) to be defined as an extension or refinement of another class (parent class). This child class is said to be derived from its parent class (or base class). The child class not only can inherit all or some of the features of its parent, but also can add new features of its own. Inheritance provides a facility that allows the simulation objects to be successively refined as a simulation program is developed in a parsimonious way [28]. *Polymorphism* is the ability to overload the meaning of an operator or method that meets the need of a newly defined child class. In a simulation package, this allows one method call to have different meanings to different members in a class hierarchy. Both inheritance and polymorphism promote software reuse by taking advantage of previously defined classes while still providing mechanism for tailoring these classes to specific applications.

1.3 Motivation and Objectives

A great deal of DES and PDES studies involve the modeling of queueing systems. This is because any system that involves arriving demands requiring access to a finite-capacity resource may be characterized as a *queueing system* [20]. That is, any system may be termed a queueing system if it involves entities that need service from a resource which has a limited capacity. Here, entities can refer to people such as customers in a bank system, or refer to objects such as broken machines that need to be fixed. The resource can refer to a person such as a clerk in a bank, or refer to an object such as a printer for printing files. A queueing system may be modeled as a *queueing network* as it usually involves entities moving through a network of queues waiting for service [25]. A *queue* in a queueing system is a collection of associated entities that are waiting for the service from a resource and which are ordered in some logical fashion such as first-come-first-served (FCFS). From retail service systems to telecommunications systems, it is apparent that many real-world systems can be classified as queueing systems. Many of these queueing

systems are so complex that only high fidelity DES is able to capture their dynamic characteristics.

There are two major problems in the simulation of queueing systems. First, the development of a DES model is very difficult when the real system to be modeled is complicated. Second, the necessity for repetitive sample generation for statistical analysis and the testing of numerous alternatives can make DES very computationally intensive. Solutions to the first problem have been addressed over decades in the general DES community through numerous simulation languages and packages. These simulation languages and packages provide modeling frameworks that facilitate the construction of complex models. Solutions to the second problem have also been explored for decades in the PDES community through many techniques on how to speed up simulation execution on parallel and distributed machines. Moreover, object-oriented (OO) techniques are also widely used in both communities in order to facilitate the simulation development process. The focus and objectives are different between the two communities, and little work has been done on the intersection of these two problems [27].

The research presented in this thesis aims to address these problems by systematically studying the above issues and developing an OO modeling and simulation package for simulating queueing systems. This package will be called QueKit in the thesis. There are two goals in the development of QueKit. The first goal is to provide a comprehensible framework for conceptual guidance in the design and development of simulation models for queueing systems. The second goal is to achieve high efficiency both in sequential execution and in parallel execution. Thus the first goal addresses the modeling effectiveness issue and the second goal addresses the execution efficiency issue. Although QueKit has been designed to support parallel execution, the scope of this thesis is limited to the implementation and evaluation of a sequential version.

1.4 Overview of Thesis

The modeling effectiveness issue concerning the development of a simulation software for a queueing system is addressed from three dimensions in this thesis: modeling framework, model architecture, and object orientation. The chosen framework and model architecture in the modeling process will largely affect the execution efficiency in the simulation. Thus the modeling effectiveness issue is discussed with the execution efficiency issue together throughout the entire thesis. The organization of the thesis is as following.

Chapter 2 discusses the modeling effectiveness issue from the first two dimensions: modeling framework and model architecture. It surveys several current modeling frameworks related to queueing system simulation, including event-driven, the process view, and the logical process view. The modeling power and execution efficiency of each modeling framework is discussed and compared to others. Two model architectures, server architecture and client architecture, associated with the modeling of queueing systems are then presented. Their relationships to the above modeling frameworks and object orientations are also discussed. Finally, fourteen problems that commonly occur in the modeling of queueing systems are discussed. Some of them are addressed with simple examples in terms of modeling framework and model architecture.

In Chapter 3, the necessity of providing a simulation package that facilitates the model design and development is discussed. The current approaches to developing an OO simulation package are then presented from the third dimension, i.e., object orientation. Some packages related to queueing system simulation are surveyed with the discussion of the object orientation, modeling power, and execution efficiency of each.

The design and implementation of an OO simulation package - QueKit - are presented in Chapter 4. The package is outlined first. The modeling framework provided by QueKit is discussed with its object orientation. Then the design of the base layer of QueKit is discussed with base classes and their interfaces. The server architecture supported by the base layer is also discussed. The necessity of developing an extended layer is discussed fol-

lowed by the presentation of the extended classes and interfaces. Other two architectures (client architecture and server-client architecture) supported by the extended layer are also presented. Finally, some implementation issues are discussed.

Chapter 5 reviews the fourteen common modeling problems defined in Chapter 2 and presents two benchmark systems used for the preliminary evaluation of both QueKit modeling power and its execution efficiency. Eight models are used for modeling these two systems, and designs of these models are presented. The modeling and simulation results are then discussed in terms of modeling power and execution efficiency.

Chapter 6 sketches the summary and conclusions as well as contributions presented in the thesis, and suggests some future work in this research.

Chapter 2

Modeling Queueing Systems

The major task in the simulation of a system is to come up with a model that captures the dynamic behavior of the system. *Modeling* is to abstract from reality a description of a dynamic system, i.e., to create a model that represents the system [13]. An application programmer usually has to follow a certain modeling framework in order to construct a simulation model. The *modeling framework*, or *conceptual framework*, or *world model view*, is a structure of concepts under which a modeler is guided to represent a system in the form of a model [1]. Specifically, the modeling framework is the way of presenting a model and the way of implementing its event-scheduling mechanism [7]. The chosen modeling framework determines how the modeler must view the system to be modeled and how a model can be constructed.

This chapter discusses the issues of effectively modeling queueing systems from two dimensions: modeling framework and model architecture. The current modeling frameworks related to queueing system simulation are reviewed. Their modeling power and execution efficiency are analyzed. The current approaches to modeling queueing systems are then presented in terms of model architecture. And finally, some common modeling problems for queueing systems are discussed.

2.1 Current Modeling Frameworks

The contemporary modeling frameworks for general discrete event simulation (DES) are *event-driven* [9], *activity-scanning* [29], and the *process view* [14]. The dominant modeling framework for parallel discrete event simulation (PDES) is the *logical process (LP) view* [15].

2.1.1 Concepts and Definitions

The *event-driven view* emphasizes the scheduling of all events. This means that no provision is made for making a state change by tests on model state; if a state change is to occur, it must occur by explicit scheduling of an event. Therefore, when using the event-driven view, a modeler first needs to identify all types of events. Then for each type of event, the modeler writes a event routine that gives a detailed description of the state changes that take place when that type of event occurs. The simulation evolves over time by processing the events, i.e., executing the corresponding event routines, in increasing order of their occurrence time. The simulation terminates when the pre-defined condition(s) is(are) satisfied, e.g., the simulation end time is reached.

The *activity-scanning* view chooses the next event (i.e., the occurrence of a state change in the system) based on both the scheduled time and condition testing. This makes the activity-scanning view best suitable for the simulation models such as animations in which the system states change continuously. Thus it is seldom used in discrete event simulation, and it will not be discussed in the thesis any more.

A *process* is a time-ordered sequence of events and activities that describe the lifetime actions of one or more entities in a system. It is clear that the concept of a process is a level of abstraction higher than event. A model with the *process view* is composed of a set of processes that describe the actions of the active entities in the system being modeled. The active entities are those entities whose behaviors are of interest in the system to be modeled. Unlike the event-driven view, the total history of an active entity can be described by a single (process) routine which also contains the passage of simulated time in the process view. Languages implementing the process view require the modeler to write process routines which are quite different from event routines. Event routines occur in zero time while process routines may contain the passage of simulated time.

Actually, the process view can be split into two types: process interaction and process description (or transaction flow) [6]. In the *process interaction view*, e.g., provided in Sim-

ula [14], each process represents a single active entity. A system is modeled as a collection of processes interacting with each other while cooperating in an action or competing for system resources. The process routines usually require special mechanisms called *co-routines* provided by a simulation language for interrupting and suspending the execution of a routine, and resuming its execution at a later simulated time under the control of an internal event scheduler [6]. For modeling a queueing system, dynamic entities are mostly modeled as processes and static entities are modeled as resources except in a few cases related to the producer/consumer problems [5]. The co-routine mechanism is the characteristics of the process interaction view.

On the other hand, the *process description view*, e.g., provided in GPSS [30], is a special case of the (possibly) more general process interaction view [7]. It provides a way of representing a system's behavior from the viewpoint of dynamic entities (modeled as transactions) moving through the system. A process routine here contains a set of blocks (i.e., constructs provided by a simulation language) which delineates everything that happens to a group of dynamic entities as they move through the system. The interactions among those transactions cooperating for an action or competing for system resources are handled automatically by the simulation language which utilizes the process description view. It is easy to understand and put the emphasis on modeling the dynamic entities in the simulation a queueing system. The programmer doesn't need to be concerned with arranging the transactions to join and to depart from the queues in the model because these queueing operations are automatically handled by the underlying simulation language.

The LP view is the dominant modeling framework used in parallel discrete event simulation (PDES). With the *LP view*, any system being modeled is viewed as a system which consists of some number of physical sub-systems called physical processes (PPs) interacting in some manner. Each PP is mapped to a software object or procedure called logical process (LP) in a simulation model which consists of a collection of LPs. The interactions between PPs are modeled as (event) messages passing between the corresponding LPs.

The LP view is widely used as the framework for model construction in PDES because it often results in the most efficient model execution in parallel. Fujimoto [15] notes that this LP methodology allows application programmers to partition the simulation state variables into a set of disjoint states in different LPs, and ensures that no event accesses the state in more than one LP. This partition permits "minimal" processor synchronization, and thus has become a de facto standard for PDES paradigms [27].

The LP view has a close relationship with the object-oriented (OO) modeling because of the close mapping between PPs and LPs. A model with the LP view is usually used with the event-driven view [16] or with the process interaction view [22]. Thus the LP view is essentially the event-driven view or the process interaction view in parallel simulation with the constraint that no LP can access the state of other LPs except through messages.

2.1.2 Modeling Power Versus Execution Efficiency

The *modeling power* of a modeling methodology or tool, e.g., a modeling framework, or a simulation package, is the measure of how powerful to construct a model with that methodology or tool. It includes two aspects: comprehensibility and flexibility. The comprehensibility is about how easy a model can be constructed with that methodology or tool, and how comprehensible the model will be. The flexibility is about how flexible the methodology or tool can cover a wide range of scenarios for modeling a particular problem. For simplicity, the discussion about the modeling power of a modeling methodology in this section is also applicable to a modeling tool, and so is the discussion about the execution efficiency.

There is not any formal method to measure the modeling power of a methodology. One method is to approximately measure the relative modeling power of different methodologies by comparing the comprehensibility of the models for modeling the same system. For instance, if one model is obviously more comprehensible than another one, then the modeling power of the methodology used in the first model can be considered greater than that used in the second model from the aspect of comprehensibility.

If the above two models are built with programming languages, counting the source code lines in the models may help to measure the comprehensibility of the two models. One model can be considered more comprehensible than another if the number of source code lines in the first model is less than that of the second model. This is because it usually means a model is easier to understand if it contains less source code lines though this is not always true. Thus this method of comparing the number of source code lines between two models is very weak. It can only be a secondary method for comparing the comprehensibility of two models.

Another method is to approximately measure the relative modeling power of different methodologies by comparing their flexibility to cover a broad range of scenarios for modeling the same system. One methodology can be considered more powerful than another from the aspect of flexibility if it can cover a wider range of scenarios for modeling the system.

The *execution efficiency* of a modeling methodology is the measure of how fast a model using that methodology can be executed on a machine. Unlike the modeling power, the execution efficiency of a modeling methodology can be accurately measured. For modeling a specific problem, one methodology can be considered more efficient than another if the execution time of a model using the first methodology is less than that of another model using the second methodology in the same simulation environment including hardware and supportive software.

From the discussion in the previous section, the process view has several advantages over the event-driven view. First of all, the process view provides a more natural modeling framework for modeling queueing systems. This is because the entire experience of a dynamic entity as it flows through a system or the life cycle of a static entity as it processes dynamic entities over time can be clearly described in a single process routine instead of being scattered in several routines. Furthermore, when this approach is implemented in a simulation language, the powerful primitives provided in the language auto-

matically translate certain situations commonly occurring in a simulation model into the corresponding event logic. As a result, a model using this approach may require significantly fewer lines of code than its counterpart one using the event-driven approach.

However, the complex assignment of resources to entities can be difficult to model because the programmer does not have easy ways to access the queues that are attached to the resources. Moreover, for the process description view, the frequent re-scans of the entire suspended event chain cause serious computational inefficiency. This is because the size of the suspended event chain which models those dynamic entities failing to get the resources increases along with the increase of the model size. For the process interaction view, the context switching between multiple processes also causes serious computational inefficiency.

The event-driven view, on the other hand, avoids the above problem of computational inefficiency by placing on the programmers the burden of working out when conditional actions can take place. This is because identifying an event and envisioning the "flow" of a dynamic entity through the system takes imagination and an understanding of the entity's interaction with possibly many events affecting it [6]. The event-driven view is thus more welcome for skillful DES modelers whose major concerns are the computational efficiency.

Therefore, the execution efficiency of the event-driven view is usually better than that of the process view, but the process view is usually easy to use, and the resulting model is easy to understand. Thus the modeling power of the process view is usually greater than that of the event-driven view in the aspect of comprehensibility.

However, it is largely depends on the primitives provided by a simulation package that the process view supported by the package are flexible enough to cover a broad range of scenarios for a specific modeling problem. For some mature languages and packages which support the process view, e.g., Simula and GPSS, their modeling power are greater than that of others which support the event-driven view in both aspects of comprehensibil-

ity and flexibility. This is because the primitives provided in these languages and packages are flexible enough to cover a wide range of scenarios for a specific modeling problem. This may not be always true for other simulation languages or packages. Therefore, the modeling power of the event-driven view may be greater than that of the process view in the aspect of flexibility if the primitives provided in the latter are not enough to support a wide range of scenarios for modeling the same system.

In PDES, the simulation model is composed of some LPs which represent the corresponding physical processes (PPs) in the system. These LPs are mapped onto several independent processing elements (PEs) in order to execute the model on a parallel machine. It can be viewed that the global event list in the conventional DES is divided into several event sub-lists in PDES. Each event sub-list is handled by a distinct processor independently. An event (message) moves from a queue of one LP to the event sub-list handled by one processor, and then migrates to a queue of another LP possibly handled by another processor in order to perform an interaction between two LPs.

As discussed in Sec. 2.1.1, the LP view is essentially the event-driven view or the process interaction view with the constraint that no LP can change the state of another LP except through messages. Thus the execution efficiency of the LP view associated with the event-driven view in the sequential execution is not as good as that with the conventional event-driven view because of the constraint of accessing shared variables in the LP view. Similarly, the execution efficiency of the LP view associated with the process interaction view in the sequential execution is also not as good as that with the conventional process interaction view.

However, the close mapping between PPs and LPs in the LP view enables OO techniques to be easily applied in the modeling process using the LP view. This tends to make the LP view easier to use and the resulting model easier to understand. Therefore, the modeling power of the LP view associated with the event-driven view is greater than that of the conventional event-driven view. And similarly, the modeling power of the LP view

associated with the process interaction view is greater than that of the conventional process interaction view.

2.2 Approaches to Modeling Queueing Systems

This section discusses the current two approaches, server architecture and client architecture, to modeling queueing systems from the viewpoint of model architecture. From the concepts mentioned in the previous chapter, entities in a queueing system fall into two categories: (dynamic) entities in the first category need services from other (static) entities while entities in the second category provide these services. For simplicity, entities in the first category are referred to as tokens and entities in the second category are referred to as servers in the rest of this chapter.

Two types of models for simulation design are used in the discussion of the approaches: functional model and declarative model. A functional model represents a modeled system as a directional flow of a signal among transfer functions (boxes) [13]. The modeled system is seen as a set of boxes communicating with messages or signals. For any functional model presented in this thesis, boxes represent service objects (servers, sources, and sinks) while messages or signals represent tokens in a model. A declarative model represents a modeled system by describing its dynamics over time. The event graph is a kind of declarative model [13], and it is used for the discussion of model design in the thesis.

2.2.1 Server Architecture

One way to construct a model for a queueing system is to put the emphasis on the token processes being done at each server. A server functions as an active controller that is in charge of allocating/deallocating resources to a token, scheduling the token's activities, and eventually routing the token to another server. Tokens are passive objects that flow through a network of servers to get services.

For example, a computer system is composed of a CPU and four disks[1]. A fixed number of tasks of two different classes execute in the system. The activities of a task are alternated between the CPU and a disk which is randomly and uniformly chosen from one of the four disks. There are n0 number of class 0 tasks which have a mean CPU execution time of 10 ms (milliseconds), and n1 class 1 tasks which have a mean CPU execution time of 5 ms. CPU execution times for both classes are exponentially distributed. CPU requests of class 1 tasks have preemptive priority over those of class 0 tasks. The latter is preempted and queued, and the CPU is assigned to the class 1 task. The interrupted task resumes its CPU execution when there is no more class 1 task requesting the CPU. The disk requests of both classes are also exponentially distributed with a mean of 30 ms. Tasks completing disk service return to the CPU queue, cycling in this way indefinitely.

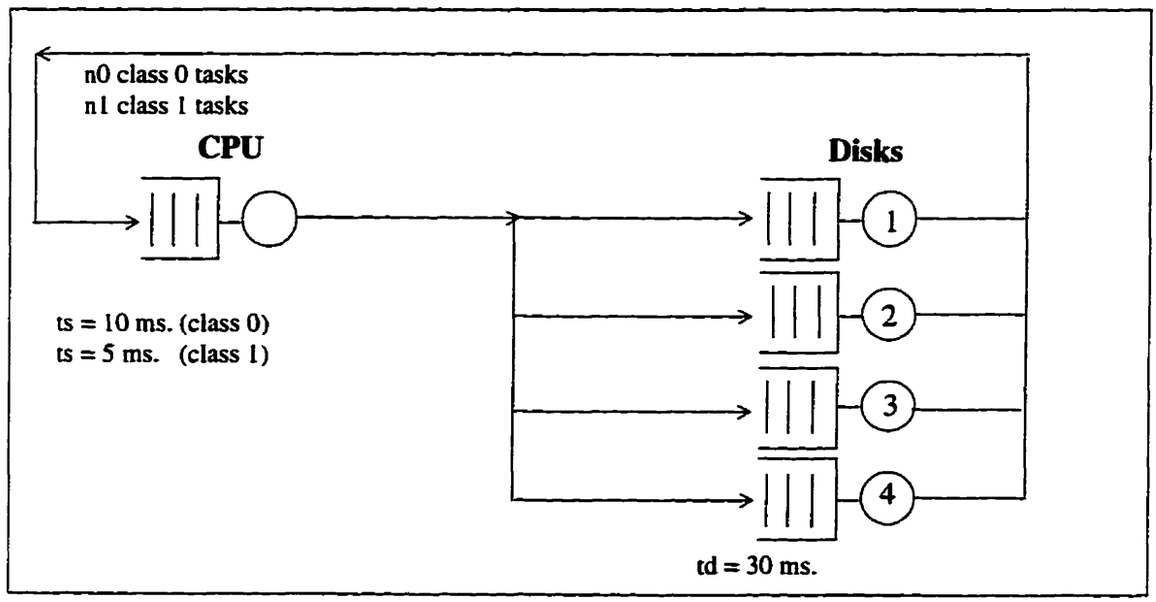


Fig. 2.1 CPU_Disk System

1. This system comes from the M.H.MacDougall's "Central Server Queueing Network" example [23] with a little modification.

The server architecture for modeling this CPU_Disk system is shown with a functional model below. Both CPU and four disks are modeled as servers. Tasks are modeled as tokens.

In the server architecture, servers are active objects in the sense that they control both resource management and tokens' activities. A server decides when and how to allocate resources to tokens and arranges their activities. This kind of passive token versus active server approach is convenient for modeling the scenario in which a master controller in a server takes care of everything: allocating resources to a token, scheduling the completion of a service for the token, and eventually routing (sending) the token to another server. However, the description of the server's behavior will become quite complicated when a token needs multiple resources in an activity and resource allocation strategies are different (e.g., one is based on first-come first-serve (FCFS) and another is based on tokens' priorities). The use of the server architecture in this situation will make the model complicated and hard to understand.

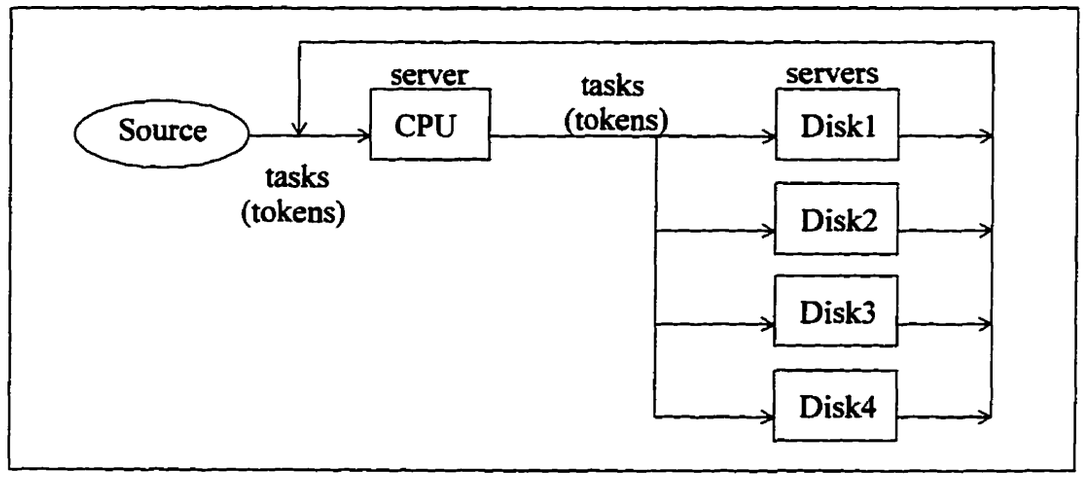


Fig. 2.2 Functional Model for the CPU_Disk System (Server Architecture)

Thus the server architecture is suitable for modeling computer systems and communications systems in which it can be viewed that a master controller in a server controls

everything. In these systems, jobs, processes, and packages can be modeled as tokens; printers, CPUs, and routers can be modeled as servers.

The declarative model of the CPU_Disk system is shown below. A block in the graph indicates an event in which the bold text shows the event type while the rest of the text shows the actions upon the occurrence of the event. A solid arrow represents an activity in the same object while the dashed arrow represents an activity between different objects. An activity may involve scheduling another event, i.e., the event indicated by the head of an arrow which marks the end of the activity. In the following figure, for example, the "start a service for a token" event in a disk will schedule a "finish a service" event to occur in the same disk object at the end of the service activity.

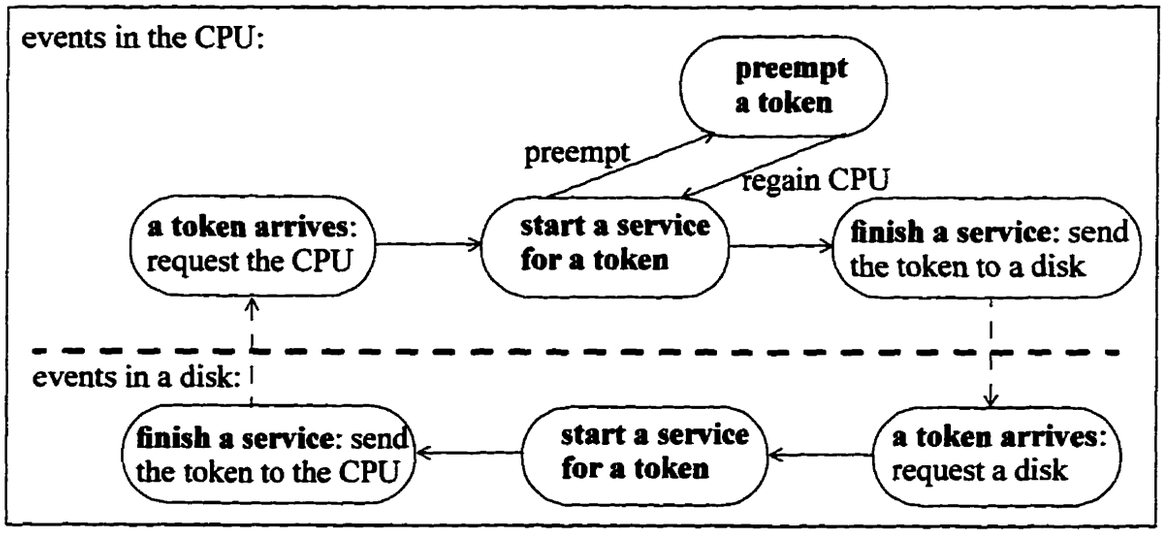


Fig. 2.3 Declarative Model for CPU_Disk System (Server Architecture)

A model with the server architecture is usually implemented with the event-driven view, it may be implemented with the process interaction view as well [31]. When implemented with the event-driven view, each server may change its state upon the following events: the arrival of a token, the completion of a service for a token, and so on. When implemented with the process interaction view, each server can be modeled as a process.

Each process may take tokens from the token pool of its upstream servers where tokens have finished services in those servers, then process them, and put them into the token pool for its downstream server.

2.2.2 Client Architecture

Actually, in most queueing systems, it is often a token itself, not a centralized server controller, which decides where the token should go, when it requests a resource, and how long it keeps the resource. Moreover, the management of different resources may be independent of each other, not controlled by a central controller. This comes to the need for decentralizing the control functionality in a single server object into two parts: one for tokens' activity control and another for resource management.

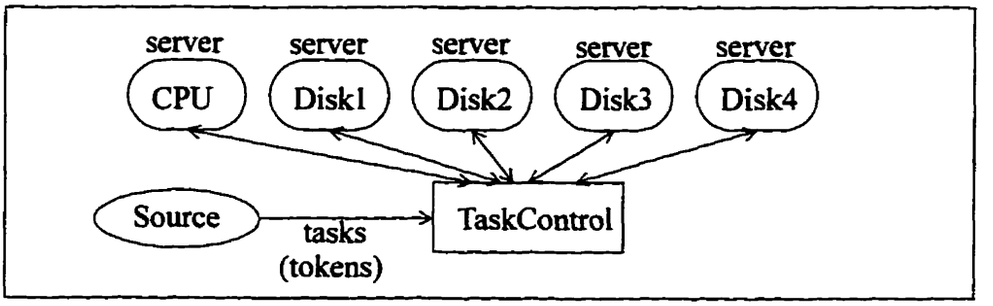


Fig. 2.4 Functional Model for the CPU_Disk System (Client Architecture)

The above figure shows that the client architecture in which the modeling emphasis is shifted from servers to tokens. A server is passive in the sense that it only controls resource allocation/deallocation but has no control on tokens' activities or how long resources will be possessed by tokens. A token decides how long it possesses the resource in the server and when it returns the resources by communications with the servers. The declarative model is shown below.

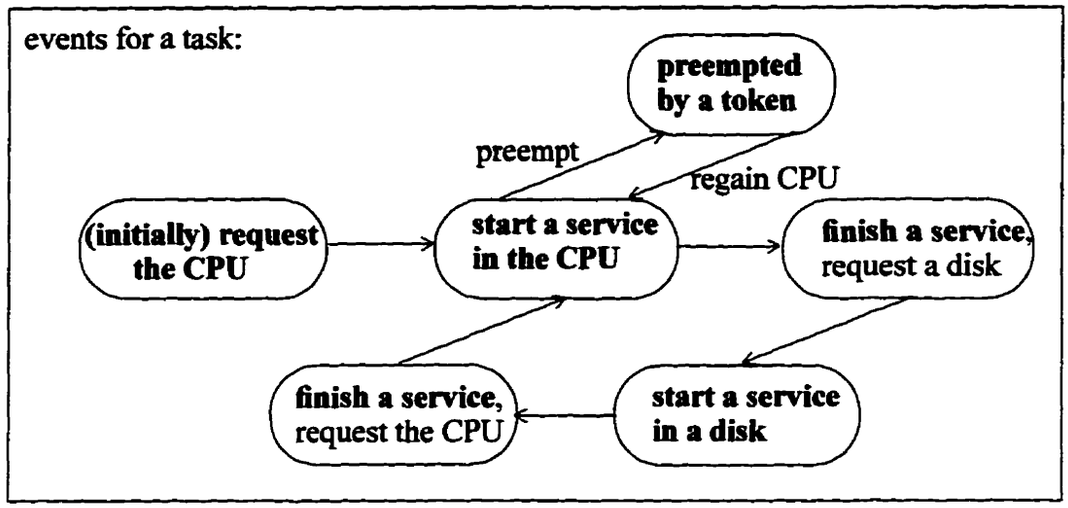


Fig. 2.5 Declarative Model for the CPU_Disk System (Client Architecture)

In contrast to the previous server architecture, the total lifetime activities of a task are described in a single function of for a task instead of being scattered in several functions within CPU and disks (shown in Fig. 2.3). Therefore, a model with the client architecture is suitable for implementing with the process view though it can be implemented with the event-driven view as well. The CPU and disks can be modeled as resources, and tasks can be modeled as transactions in the process description view, or modeled as processes in the process interaction view. Thus the lifetime activities of a task can be described in a single process routine.

2.3 Common Modeling Problems

Although there are numerous modeling problems in the simulation of queueing systems, only fourteen of them that are commonly encountered will be discussed below.

Resource sharing: A server provides service for a group of tokens alternatively, and the service amount of any token can get may depend on the total number of tokens in the group. For instance, the requests from the terminals in a time-sharing system will processed in a round robin fashion in order to guarantee the reasonable response time for the

users. This system can be constructed with the server architecture. The requests can be modeled as tokens, the central server and terminals can be modeled as servers.

Preemption: Preemption happens when an activity of a token is interrupted by another token with a higher priority. A token is interrupted while it is being serviced. It moves back to a wait queue with its service demand possibly reduced. Its position is taken over by another higher priority token in the system. This scenario is called preemption. The former token is called the preempted token and the latter is called the preempting token. For example, a phone call may be interrupted by a higher priority emergency call, and it may continue after the emergency call is finished. The server architecture can be used for modeling this system. The phone can be modeled as a server and the calls can be modeled as tokens.

Breakdown: The ability of providing services of a server is lost for some time due to an internal or external event in a queueing system, and then is regained later. During this down time, any token which is in a service may be interrupted. It may continue its remaining service or restart the interrupted service from the beginning once the broken-down server recovers. Breakdown happens when the status of a server is changed by internal or external events. One example is that a printing job will stop when a printer is out of paper and start again to continue the printing once the paper is fed in. The server architecture can be used for the model construction here. Jobs can be modeled as tokens and the printer can be modeled as a server.

Both preemption and breakdown are related to the activity interruption (or called cancellaton) of tokens. The difference between the two is that the interruption is caused by another token in the former case while it is caused by a change of status in a server in the latter case.

Loss: A token may leave its modeled system because its lifetime has been expired at a certain point before it can finish its journey of activity in the system. For example, a box of decayed fresh fruits from inland may be thrown away before it is loaded on an airplane for

export. The server architecture is suitable for modeling this system. Airplanes can be modeled as servers and the boxes of fruits can be modeled as tokens.

Dropping: A token may be discarded from its modeled system if the length of the waiting line for a server reaches a certain point. An example of this scenario is that low priority cells are discarded in a ATM switch when the number of cells in the corresponding buffer reaches its threshold point, e.g., the threshold point is half of the buffer capacity. The server architecture is preferable for the model construction in modeling this system where ATM cells can be modeled as tokens and the switches can be modeled as servers. ATM cells start travelling from source servers, move through a network of switch servers, and finally reach the destination servers.

Both loss and dropping are situations where a token leaves its modeled system prematurely. The difference between the two is that the time when the token leaves the system is pre-defined or predicted in the first case but not in the second case.

Balking: A token fails to join any queue in a system because of some reasons. A simple example is that a customer decides to leave a bank before s/he joins any wait queue because all the queues seem too long for her/him and s/he has no patience for waiting in any queue.

Migration: A token changes from one wait queue to another while it is waiting for a service. For instance, a customer in a shopping center switches to another wait line because s/he thinks that line is serviced faster.

Reneging: A token leaves the wait line before it reaches the server for service. For example, a person who is waiting for accessing a banking machine leaves the wait line because s/he has no patience to wait any more or s/he has another more important business to do.

Affinity: Among several available servers, some tokens may only choose a specific one to wait for services. A simple example is that some customers only wait for their

favorite barbers to have their hair cut. The model for this system can be constructed with the client architecture in which customers can be modeled as tokens and barbers can be modeled as resources. A customer token can request a resource according to his/her own choice and has his/her hair cut.

Grouping: A server starts a service only when the number of tokens waiting for its service reaches a certain point. Then it takes in some or all of the tokens and starts services for them simultaneously. For instance, a tourist bus begins a tour only when the waiting tourists are enough to fill at least half of its capacity.

Routing: A token is sent by a server to another one after it gets serviced, or a token chooses another server after it finishes the service in the current server. The former is the case that a server routes a token while the latter is the case that a token routes itself in the system. For example, a packet of information is routed by a series of network nodes to the destination in a computer network. Another example is that a person goes shopping by car. S/he needs to buy many items including food, medicine, clothes, and so on. S/he makes the decision, possibly dynamically, on which order to visit a sequence of stores.

Tandem queueing: The transfer of a token from a server to another one starts only when the second server is able to accept the token, i.e., there is a room in the second server for accommodating a token. Otherwise the token has to wait in the first server until a room is available in the second server. One example is that the parts in an assembly line have to wait in the area of the current station before being transferred to the next station when there are rooms available for them over there.

Multiple resources: A token needs several servers simultaneously for an activity. A simple example is that the repairing procedure can only be conducted for a broken machine when a mechanic is not busy and the equipment is available. The client architecture is suitable for the model construction here. Broken machines can be modeled as tokens, mechanics and the equipment can be modeled as servers. A broken machine can request a mechanic and the equipment before the repairing procedure can start.

Entity Transformation: An entity changes its role between a dynamic entity and a static entity in a queueing system. For instance, a machine in a factory system is a static entity when it provides service for refining parts one by one that arrive at the machine. When the machine breaks down, it needs to be fixed by a mechanic who is in charge of the fixing work for all machines in the factory. That is, the machine changes to a dynamic entity which needs service from others when it is broken down. The machine functions as a static entity and starts to work for parts refinement again after it is fixed by the mechanic.

This problem can be modeled with server architecture. A part is modeled as a token which moves through a set of machines in order to finish a series of refinement processes. A machine is modeled as a server that takes the parts from its wait queue, operates on them, and passes them to a machine at the next refinement stage. A mechanic is also modeled as a server. A special token models a breakdown event of a machine. The machine changes its status from available into unavailable when the special token arrives at the machine. Then the special token is sent to the mechanic server by the broken-down machine server. The mechanic server keeps it for some time (modeling the passage of service time), and sends it back to the machine server indicating that the fixing job is done. Upon receiving this special token, the machine server changes its status back to available and starts its service again.

2.4 Summary

This chapter discusses the modeling issues from two dimensions: modeling framework and model architecture.

First of all, the current modeling frameworks including event-driven, the process view, and LP view are reviewed in this chapter. The process view can be split into two type: process interaction and process description (or transaction flow). The analysis of the modeling power versus the execution efficiency of these three frameworks are then discussed. It is concluded that the modeling power of the process view is generally greater than that of the event-driven view whereas the execution efficiency of the event-driven

view is better than that of the process view. The LP view is essentially the event-driven view or the process interaction view with the constraint that no LP can access the state information of another LP except through messages. That is, messages should be used for accessing any shared variables in the LP view. Thus the execution efficiency of a model built with the LP view in a sequential environment is not as good as that with the event-driven view or the process interaction view due to the constraint of accessing shared variables in the LP view. However, the mapping between PPs and LPs in the LP view results in the close relationships between the LP view and the real-world systems. This enables the model that uses the LP view to be easy to build and easy to understand. Hence, the modeling power of the LP view associated with either the event-driven view or the process interaction view is greater than that of the event-driven interaction view or the process view, respectively.

The current two approaches, server architecture and client architecture, to modeling queueing systems are also discussed from the viewpoint of model architecture. The server architecture focuses on modeling servers, i.e., the entities that provide service for others, in a queueing system. It emphasizes how a server allocates/de-allocates resources to a token, how it controls the token's activities, and finally routes the token to another server. A model with the server architecture is usually implemented with the event-driven view, it may be implemented with the process interaction view as well. On the other hand, the client architecture focuses on the modeling of the lifetime activities of a token moving through the system. It puts the emphasis on describing the lifetime activities of a token in a chronological order as it moves through the system. This description is written in a single function in which the token actively requests resources before conducting an activity and returns the resources after the activity. A model with the client architecture is usually implemented with the process description view or process interaction view, it may be implemented with the event-driven view as well.

Finally, fourteen modeling problems that commonly exist in the simulation of queueing systems are addressed with simple examples. Some of them are also discussed in terms of model architectures.

Chapter 3

Packages for Queueing System Simulation

Although some general-purpose languages such as C++ can be used to construct simulation models, they cannot provide the modeling frameworks discussed in Chapter 2 for conceptual guidance. Those modeling frameworks have been provided by many simulation packages based on general-purpose languages. The major advantage of using a simulation package is that it automatically provides most of the features needed in programming a simulation model. It thus results in a significant decrease in programming time (and usually project cost) in a simulation process [21].

There are numerous of simulation packages for modeling queueing systems as many discrete event simulations (DES) involve the simulation of queueing systems. Object-oriented (OO) techniques have been widely used in the design and development of these simulation packages. This chapter surveys some packages related to the simulation of queueing systems and discusses their object orientation and modeling power as well as execution efficiencies.

3.1 Approaches to OO Simulation Package Design

Object-oriented simulation has great intuitive appeal because it is easy to view a real-world system as being composed of objects [18]. OO concepts are applicable to simulation software development at the following levels [2]:

- Abstraction: OO techniques [17] are applied in the analysis of the modeled system from the viewpoint of the real-world concepts. The easy mapping between real-world objects and software objects is emphasized at this layer.
- Design: OO techniques such as encapsulation, inheritance, and polymorphism are applied in the overall software design. The software robustness, extensibility, maintainability, etc., are major concerns at this layer.

- Implementation: OO programming (OOP) techniques such as encapsulation, inheritance, parameterized typing, etc., which may be related to a specific OOP language, are used for the implementation of the simulation software.

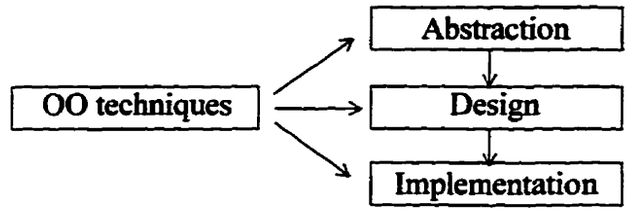


Fig. 3.1 Role of OO Techniques

There are three approaches to developing an OO simulation package. The first one is to develop a data-driven simulator which can provide a set of simulation constructs at the abstract level for model construction. With a data-driven simulator, a model can be built with a set of data that are formatted and provided by the user. The data is fed into the simulator through its interface such as a graphical user interface (GUI). The second approach is to provide those constructs by developing a simulation language as an extension of a general-purpose language, e.g., Simula is an extension of ALGOL language. The last approach is to provide those constructs by creating a library of simulation classes in an OO language such as C++ [32] or Simula [14]. Some packages use the data-driven simulator approach combined with either language extension or library-based approach in order to take a full advantage of OO techniques.

For the data-driven simulator approach, the user would be able to build simulation models by easily taking advantage of the close mapping between the simulation constructs and objects in the system to be modeled. There is no programming involved when using a data-driven simulator. The user can concentrate on the model design, model construction, and data analysis, rather than programming and debugging.

The limitation of this approach is that it lacks the flexibility to handle a wide range of applications. For instance, the need of a new simulation construct for a particular model-

ing problem may result in the user waiting for the simulation developers to add the construct into the simulator. This is because a data-driven simulator fails to take a full advantage of OO techniques such as inheritance, polymorphism, parameterized typing, and so on. That is, the data-driven simulator approach only applies OO techniques at the abstraction level, not yet at the design and implementation levels shown in Fig. 3.1.

On the other hand, both language extension and library-based approaches have the flexibility to cover a larger range of applications than the data-driven simulator. The language extension approach also has the advantage that the compiler or pre-processor can provide strict type checking and code optimization capabilities. However, it has the disadvantage of requiring the user to learn a new language and perhaps an entire new set of programming development tools. The primary advantage of the library-based approach is that the user can continue to use a familiar programming environment, and have full accessibility to the features of the base language.

The packages using either of these two language approaches usually apply OO techniques at the design and implementation levels, but not all of them fully apply OO techniques at the abstraction level. For instance, some packages provide a construct called an event (or a message) to model an entity that needs service from others in a queueing system. To model the passage of service time of the entity in a server is accomplished by sending an event (or a message) to arrive at the server at a later time which indicates the service is completed. Some events (or messages) in a simulation model even don't model any real-world objects, they are there just for the simulation purpose (e.g., an LP queries information in another LP). There are lots of these events (or messages) being sent around in a simulation model, and this usually creates confusion for engineers trying to understand and construct a model.

Therefore, it is important to apply OO techniques at all three levels when developing an OO simulation package because this will result in a package that is easy to use as well

as flexible enough to cover a wide range of applications. Thus the object orientation can be used for the evaluation of the modeling power of a simulation package.

3.2 Packages for Object-Oriented Simulation

Several packages related to simulating queueing systems are discussed in this section. Packages for sequential simulation including Prophesy[1] and HIT [4] are surveyed followed by the discussion of the packages for both sequential and parallel simulation including PROSIT, SimKit, and ATM-TN.

3.2.1 Packages Only for Sequential Simulation

Prophesy is a simulation package intended for queuing analysis and work flow processes. While oriented to simulate computer networks, Prophesy can support simulation of practically any other queuing and work flow problem. The fundamental objects of the Prophesy metaphor are Resources, Procedures, Profiles, and Messages.

Resources: These are permanent entities in a simulation with a finite capacity and capable of receiving simulation messages, processing them, and delivering them to another resource. Each resource object may represent a single resource, or a cluster of cooperating resources, all represented by a single icon in the GUI. A cluster of resources can be made to act upon a single receive queue (tightly coupled resources), or it might define a cluster of independent resource instances acting on separate queues.

Procedures: Procedures are permanent objects called by any Resource. Procedures define a sequence of events that represent the resource's work flow in a chronological order.

Profiles: Profiles are permanent objects associated with a resource. Profiles specify the performance characteristics of the attached resource.

---

1. The information about Prophesy comes from the web site: http://www.csn.net/abstraction/

Messages: Messages are created as dynamic objects while the simulation executes. Messages are created according to the model specifications, and deleted when no longer needed.

Therefore, the modeling framework provided by Prophesy package is event-driven view. The server architecture is used for model constructions, and the Resources function as servers. Thus Prophesy may have good execution efficiency.

Another software tool **HIT** supports model-based performance evaluation of computing and communications systems. Specification of dynamic, discrete-event, stochastic systems is achieved by particular language-based and graphics-based description options.

The concept about queues in HIT is different from others, and thus is interesting. HIT defines a queue as an autonomous object which has four "areas" with possibly limited capacity. These areas are arrival area, entry area, service area, and exit area. The arrival area holds unlimited arriving jobs. The entry area with possibly limited capacity holds the jobs waiting for service. The service area is the place to hold the jobs being serviced and has possibly limited capacity. The exit area holds unlimited jobs with completed service. Accordingly, there are four procedures to control job transitions between these areas [4]. Accept procedure is responsible for accepting the jobs from the arrival area into the entry area of the queue. Schedule is responsible for controlling the job transfer between entry and service areas. Dispatch procedure assigns service time to jobs. Offer procedure selects jobs permitted to leave the queue.

Therefore, the server architecture is provided for the model construction in HIT. Here queues function as servers and jobs function as tokens. HIT may have a great modeling power as its language-based approach using an OOP Simula which provides the process view, as its host language.

3.2.2 Packages for Both Sequential and Parallel Simulation

PROSIT [26] is a sequential and distributed object-oriented workbench for discrete event simulation. It provides an Object-Oriented framework for discrete event simulation. It contains a set of simulation and modeling classes. Simulation classes are those dealing with the simulation phase whereas modeling classes are those used to build models. These two kinds of classes are base classes. To mask the simulation paradigm to the final user of the simulator, a set of library classes are provided for a specific field of application. These classes, gathered in a library, will allow the user to build a model at a higher level of description. With PROSIT, the user builds a model with dedicated class libraries and user defined classes. Without code modification, the simulation can be executed in a sequential or distributed (in both optimistic and conservative variants) way.

PROSIT is devoted to the development of a discrete event simulation system, designed from the ground up with distributed execution in mind [24]. Its design is based on the object-oriented paradigm. C++ is used for both the simulator as well as simulation model construction. This means PROSIT is a package using the library-based approach. The system provides both conservative and optimistic mechanism to synchronize processes [10]. It also provides the suspension (for a simulation time period) and re-activation of execution for processes [11].

Two architectures are chosen for model construction in PROSIT: server architecture and customer architecture. These two architectures correspond to the server architecture and the client architecture mentioned in Chapter 2 respectively where customers are the same as tokens.

With the server architecture, the user describes a model as a set of servers which provide different kinds of service for customers. Customers are received by a server, get serviced, and then are sent to other servers. The "active" objects in the model are the servers. They decide what to do with the customers being processed and where the customers should be sent.

The customer architecture changes the control of execution from the server to the customer. A customer profile (type) represents a set of customers whose behavior is statistically identical [24]. The customers lifetime behavior will be described by a single method that will be activated for each instance of customer of that type. The PROSIT simulator supports migrating a process from one processor to another. The customer process will migrate to the processor where a server resides when it needs to get the service from that server. This migration balances the workload among processors.

A PROSIT simulation can be thought as a collection of concurrently active objects interacting via service calls in simulated time. An activity is an action performed by an active object and corresponds to the member function behave(). An activity has a duration in simulated time. It can halt and be reactivated later. An activity terminates when the corresponding function finishes. Thus a co-routine mechanism similar to that in Simula [14] is provided in PROSIT. An active object executes its main activity, the behave() function, in an autonomous way, independently of, and concurrently with, other active objects [3]. Active objects can also have secondary activities which are attached to other functions. All activities are running concurrently in the simulated time (all activities are running pseudo-concurrently in the sequential version) [3].

There are seven possible states for an active object: initialized, running, sleeping, blocked, suicided (prematurely terminated by itself), killed (prematurely terminated by other objects), and finished. An object enters the initialized state when the active object is created in a C++ constructor, then is automatically managed by the kernel and ready to be activated. Its activation time for process is scheduled at this state. The object enters its running state when its activation time is reached. During its lifetime, the object can either be in a running state (i.e., currently executing or consuming time), in a blocked state (i.e., the main activity is blocked due to a synchronous request), or in a sleeping state (it has put itself in idle-wait state, waiting to be reactivated by another object). The object is considered to be dead when its main activity has terminated. There are three kinds of death: finished (i.e., normal termination of the main activity), suicided (the object terminates itself

prematurely), and killed (the object is terminated by another object using the termination primitive).

PROSIT may have a great modeling power as it uses the process view with either the server architecture or the customer (client) architecture for modeling. However, the inefficiency of the process view may make PROSIT less efficient.

SimKit [16] is a C++ class library that is designed for very fast discrete event simulation. SimKit presents a simple logical process view of simulation enabling both sequential and parallel execution without code changes to application models. The interactions between LPs in a simulation model are represented by messages (events) passing between them. This event-driven logical process view enables efficient scheduling of events via invocation of the corresponding LP's event-processing member function rather than the more costly context-switching required in the process view [16].

The SimKit class library contains only three classes: sk_simulation for simulation control, sk_lp for modeling sub-space behavior and state transitions, and sk_event for modeling the interaction between the logical processes. A simulation model is constructed by deriving LPs from sk_lp class and messages (events) from sk_ev class. The programmer also needs to instantiate a single instance of sk_simulation class in order to invoke the run time simulation kernel.

An execution of a model starts and ends with a single thread of control executing on a single processor [16]. It goes through the following six phases: program initialization, model global initialization, LP initialization, simulation execution, LP termination, and simulation clean-up.

The execution starts from the program initialization phase in which the program main() function is initialized and a single *sk_simulation* object is instantiated. The *sk_simulation* is initialized and all LPs in the model are instantiated in the second phase - SimKit and model global initialization. Allocation of LPs to processors is static and may

be optionally specified by the modeler via the LPs constructor [16]. The second phase ends with passing control to the simulation run time system. In the third phase LP initialization, all LPs' *initialize* member functions are invoked for execution. The initial events are usually scheduled in these LPs' initialization member functions. Then the model execution is controlled by the simulation kernel in stage four - simulation execution. Events are passed to the corresponding LPs by invoking their *process* member functions. The simulation execution stage ends either because the simulation end time is reached or an error occurs. All LPs' *terminate* member functions are executed then in stage five - LP termination. Finally the simulation kernel returns the control back to the main() function of the application in simulation clean-up phase.

The library-based approach is used for the development of SimKit so that OO techniques are used at both design and implementation levels. However, the generality of the LP view makes SimKit good for the application in general discrete event simulation, but not so good for the simulation of queueing systems. This is because SimKit doesn't provide enough high level constructs such as queue for modeling queueing systems.

ATM Traffic and Network (ATM-TN) system is a high fidelity simulator which characterizes ATM network behaviors at cell level. The simulator incorporates three classes of ATM traffic source models: an aggregate ethernet model, an MPEG model and a World Wide Web transactions model. Six classes of ATM switch architectures are modeled including output buffered, shared memory buffered and cross bar switch models, and then multistage switches which can be built from these three basic models [33]. The simulator is built with C++ language and the interfaces provided by SimKit. The event-driven logical process view with OO methodology is used in the construction of the simulator. ATM-TN is a highly efficient simulator dedicated to the simulation of ATM networks, thus it cannot be used for the simulation of general queueing systems.

3.3 Summary

The major advantage of using a simulation package over a general-purpose language is that it automatically provides a modeling framework for the model construction. It also provides most of the features needed in programming a simulation model. It thus results in a significant decrease in programming time (and usually project cost) in a simulation process. The use of OO techniques has the potential for developing a simulation package that is easy to use because it contains close abstraction of the real-world concepts. This chapter discusses the approaches to developing the simulation software from the third dimension, object orientation.

There are three levels for OO techniques applicable in developing a simulation package: abstraction, design, and implementation. There are also three approaches to developing of an OO simulation package: data-driven simulator, language extension, and library-based approach. The data-driven approach are usually successful in applying OO techniques at the abstraction level, but not at the design and the implementaion level. The language extension and the library-based approaches have the potential to apply OO techniques at all three levels.

Packages related to queueing system simulation in the literature usually apply OO techniques at the abstraction level, or at design and implementation level. Some of them such as PROSIT applies OO techniques at all three levels, but they suffer from the inefficiency problem because they use the costly context switching mechanism to provide the process view.

ATM-TN is an efficient data-driven simulator that is dedicated to the modeling and simulation of ATM networks. It thus cannot be used for modeling other queueing systems.

SimKit is a library-based simulation package built with an OOP language. It provides a very simple and efficient LP view for modeling and simulation various DES problems both in sequential execution and in parallel execution. However, the modeling constructs

(or simulation primitives) provided in SimKit are only at the simulation level, not at the application level when they are used for modeling a queueing system. That is, SimKit does not provide enough high level simulation constructs such as queue and server for the simulation of queueing systems. Thus SimKit is a good candidate for developing a package at a higher level for OO modeling and simulation of queueing systems.

Chapter 4

QueKit: An OO Simulation Package for Queueing Systems

The goal of QueKit is to provide an object-oriented (OO) environment for queueing system simulation that facilitates the modeling process while retaining efficiency both in sequential execution and in parallel execution. OO techniques are used for designing QueKit application programer's interfaces (API) to strive for ease of use. The logical process (LP) methodology is used for implementing the QueKit package so that models in QueKit can be executed efficiently both in sequential and in parallel environment.

An OO modeling framework provided by QueKit is presented by discussing QueKit base classes, its server modeling architecture, and its simulation classes in this chapter. An extended layer of QueKit is then outlined with the description of QueKit extended classes and its other two modeling architectures. Finally, some implementation issues are discussed followed by a brief summary.

4.1 Overview of QueKit

The "core" objects in QueKit[1] are Tokens, Sets and Servers. Tokens represent those entities that need services from other entities while Servers represent other entities that provide these services in a queueing system. Tokens flow through a network of Servers to obtain services from those Servers in a QueKit model. Sets are places for keeping Tokens in some logical fashion such as first-come-first-serve (FIFO). A Server has two Sets: a wait Set for holding Tokens that arrive at the Server and are waiting for service, and a service Set for holding Tokens that are currently being serviced.

1. The information comes from the web site: http://www.wnet.ca/telesim/quekit.html.

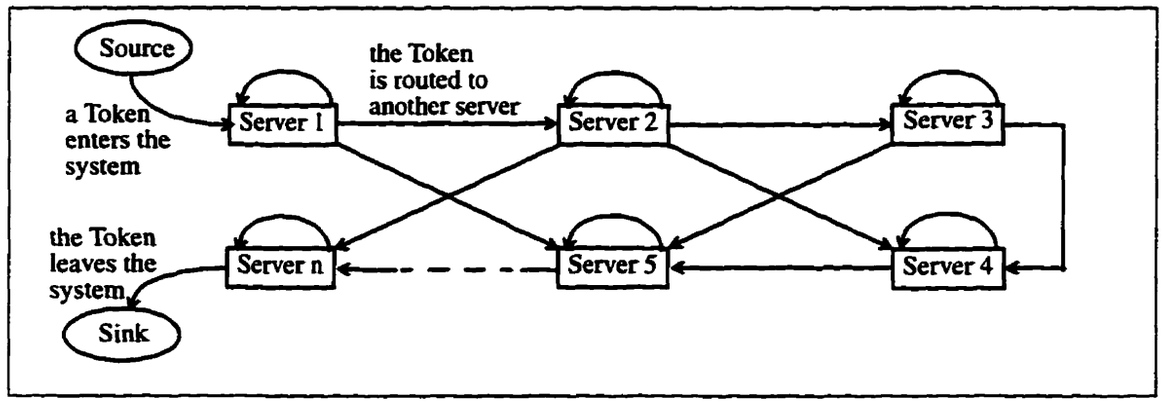


Fig. 4.1 QueKit Overview

The four "core" methods of a Token are *keep (serviceTime)*, *move (destSet)*, *send (destServer, delay)*, and *cancel ()*. The *keep* method is invoked for a Token which resides in a Set to initiate an activity of a wait or service interval. The underlying event which ends such a Token's service or wait interval always occurs within the same Server. The *move* method moves a Token between different Sets in the same Server with no delay. The *send* method moves a Token from one Server to another (possibly the same) Server with a time delay that represents its transit time. The *cancel* method explicitly cancels a Token's current activity, initiated either by *keep* or by *send* method. None of these four methods can be overridden by the user.

In a QueKit model, a Token either resides in a Set, uniquely defined by *currentSet()*, or is in transit between Servers. A Server's *schedule (Token)* method is invoked when a Token arrives at the Server as a result of a previous *send*, or when a previously initiated *keep* to perform a service or complete a wait has elapsed. An invocation of the *schedule* function may transfer Tokens between the wait Set and service Set, may preempt current services, may initiate new Token services, and may send Tokens to other Servers.

The user will typically override a Server's *schedule* method to model different service strategies in a QueKit model. The user may also extend a Token into a class of his/her own

needs which is more powerful and/or more specific than the base Token by adding more methods that a *schedule* method can invoke.

The class Server is derived from class Entity which models a sub-system of the modeled system (it is essentially an LP) and contains no Set. The Source and Sink classes are also derived from Entity. The class Entity has an *initialize ()* method which is expected to create the initial Tokens in a model and schedule their movement. It also has a *terminate ()* method which is expected to do some clean-up work such as collecting the final statistics after a simulation. Class Server, Source, and Sink all inherit these two methods. However, both *initialize* and *terminate* do nothing in the base Entity class. Thus the user may override the *initialize* method of a Server to get the initialized Tokens unless using Source objects in a model. The *terminate* method may be overwritten by the user for the collection of the final statistics.

Class QueSimulation is in charge of the simulation control in QueKit. The user needs to instantiate a single instance of QueSimulation class in order to invoke the run time simulation kernel.

The model execution starts and ends with a single thread of control executing on a single processor in QueKit in the same way as in SimKit [16]. Six phases are involved in a model execution: program initialization, QueKit and model global initialization, Entity initialization, simulation execution, Entity termination, and simulation clean-up.

The execution starts from the program initialization phase in which the program *main* function is initialized and a single QueSimulation object is instantiated. The QueSimulation is initialized and all Entities in the model are instantiated in the second phase - QueKit and model global initialization. Allocation of Entities to processor is static and may be optionally specified by the modeler via the Entity constructors. The second phase ends with passing control to the simulation run time system. In the third phase Entity initialization, all Entities' (including Servers, Sources, etc.) *initialize* member functions are invoked for execution. The initial Tokens are usually created and scheduled for actions in

these Entities' initialization member functions. Then the model execution is controlled by the simulation kernel in stage four - simulation execution. Tokens are activated for actions in the corresponding Entities by invoking Entities' *schedule* member functions. The simulation execution stage ends either duo to the simulation end time is reached or duo to an error occurs. All Entities' *terminate* member functions are executed then in stage five - Entity termination. Finally the simulation kernel returns the control back to the *main* function of the application in simulation clean-up phase.

4.2 Object-Oriented Event-Driven Modeling Framework

The main driving force for developing QueKit is to strive for much greater ease of use than the SimKit [16] package. Specifically, it is to provide an OO environment in simulation of queueing systems for the user at the abstraction level in addition to the design and implementation levels provided in SimKit. This will allow the user to easily map the real-world objects into the software objects without having too much knowledge about simulations. Meanwhile, QueKit also aims to preserve the high efficiency of SimKit as much as possible.

SimKit provides a very simple and efficient logical process (LP) framework for general discrete event simulation [16]. The SimKit API has been used for developing QueKit because of its simplicity and efficiency. The library-based approach of SimKit provides the full accessibility to its base OO language (C++ or Java) so that OO techniques such as inheritance, polymorphism, parameterized typing, etc., can be applied at the design and implementation levels. The library-based approach is also used in the development of QueKit, and QueKit is built at the application level of SimKit. This development strategy enables all OO techniques which are applicable in SimKit applicable in QueKit as well. Moreover, QueKit provides a set of high level model definition primitives specifically for queueing system simulation so that OO techniques are applicable to the model abstraction level. Thus OO techniques are applicable in QueKit at all three levels including abstraction, design, and implementation so that QueKit has greater modeling power than SimKit in the simulation of queueing systems.

In the QueKit environment, entities which need services in a system are modeled as Tokens and entities which provide services for others are modeled as Servers. A QueKit model can be viewed as a collection of objects (Tokens, Servers, etc.) which are the abstract representations of the objects in the system being modeled. The *schedule (Token)* method of a Server describes a series of actions the Server performs whenever an event occurs such as the completion of a service for a Token in the Server.

In a banking machine system, for instance, customers are usually modeled as (event) messages and the banking machine is modeled as an LP in many OO packages providing the LP view. To model the passage of service time when a customer is being serviced by the banking machine, the machine LP sends the customer (event) message arriving to itself after some delay. The model is depicted below. This shows that the LP view does not fully apply OO techniques at the abstraction level. That is, the user has to describe an object's behavior from the simulation domain instead of the problem domain. It thus causes confusion and difficulties for engineers who have not much knowledge about simulations when they build simulation models with the LP view.

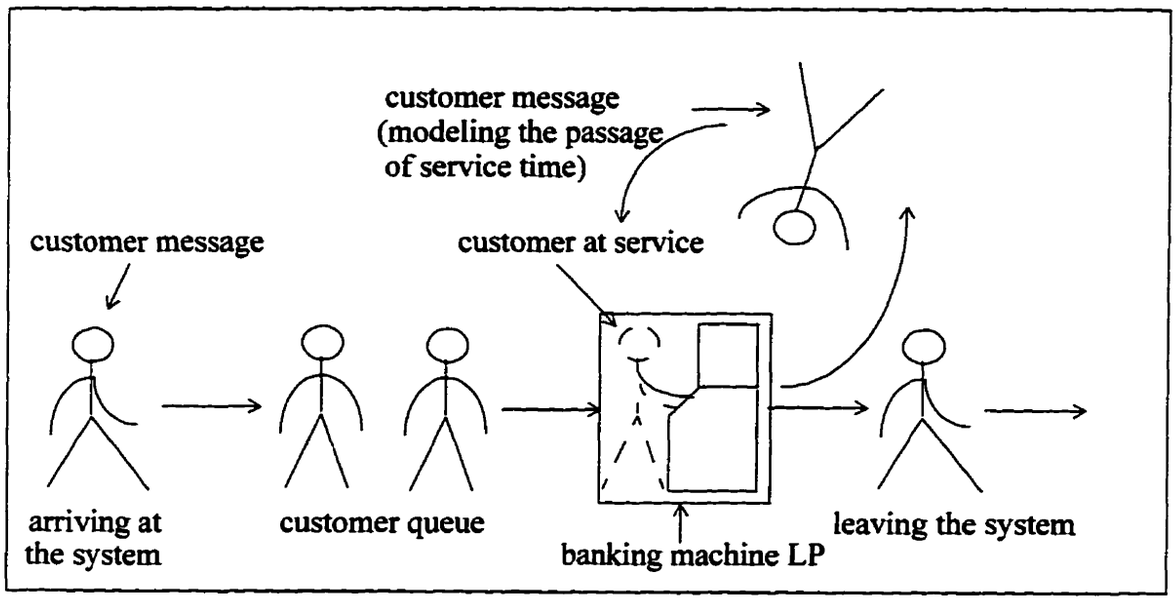


Fig. 4.2 Logical Process Modeling View

In QueKit, however, customers are modeled as Tokens and the banking machine is modeled as a Server. To model the passage of service time when a customer being serviced by the banking machine, *Token::keep (Time)* method will do the job. The customer object just resides inside the service Set of the machine Server until it is activated again by the *Server::schedule (Token)* method after the service time has passed.

There is no event or message object being sent back and forth in any QueKit model. This is different from many current OO simulation packages which provide event-driven or LP view. Events or messages which are necessary in a simulation are transparent from the view of the user in QueKit.

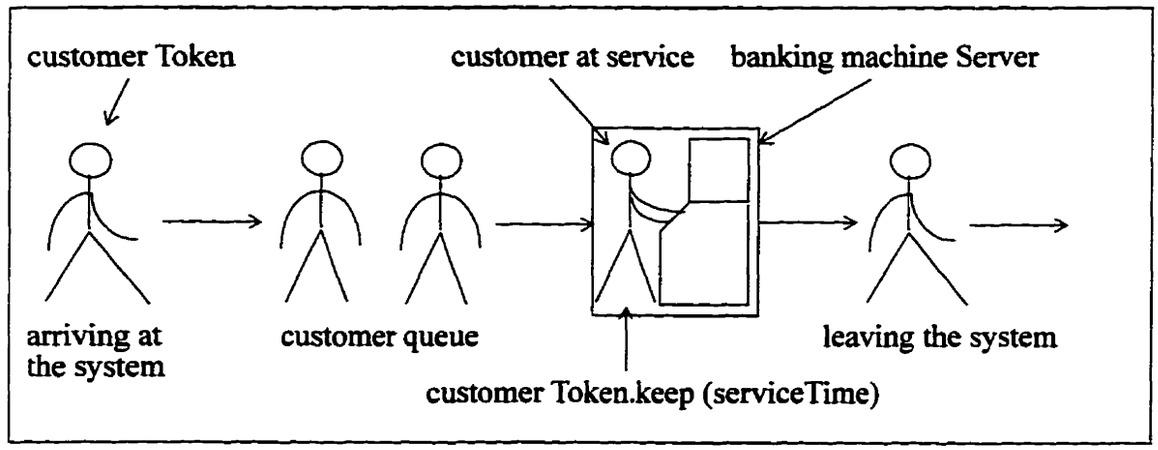


Fig. 4.3 OO Event-Driven Modeling View

4.3 Base Layer

The base layer of QueKit is developed on top of SimKit. It contains base classes for modeling a queueing system and simulation classes for simulation control and statistics as well as trace information collections. The class hierarchy is shown below. These classes support the server modeling architecture.

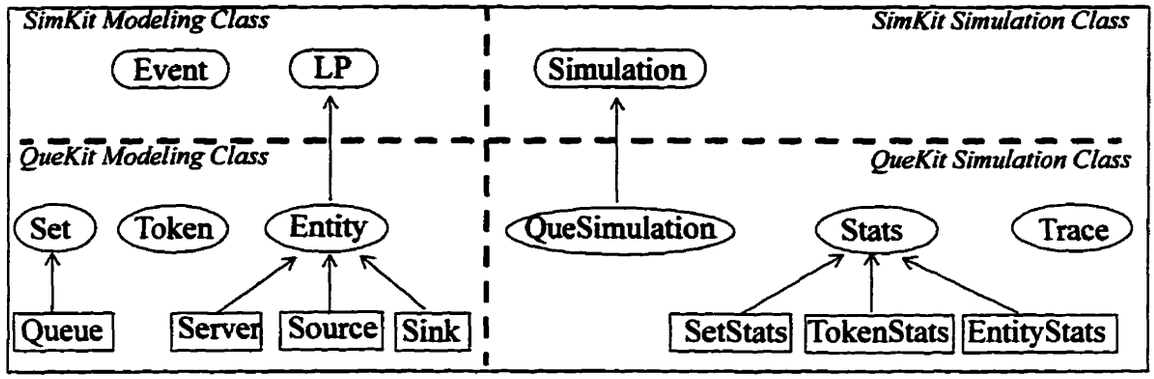


Fig. 4.4 Class Hierarchy of QueKit Base Layer

4.3.1 Base Classes

The basic classes contain Token, Set, Entity, Queue, Server, Source, and Sink.

A Token models an entity that needs services from other entities in a queueing system. A Set models a place for holding Tokens. An Entity models a sub-system in a queueing system, i.e., it is essentially an LP.

Queue is derived from Set and is a special Set provided by QueKit. A Queue models a place for holding Tokens in a priority order beginning from the highest priority.

A Source models an object which injects Tokens into the system in a pattern such as exponential distribution inter-arrival time. A Sink models an object which absorbs Tokens when they exit the system.

Server is derived from Entity and it models a sub-system that provides service for other entities in a queueing system. A basic Server has two Sets: a Queue with infinite capacity as its wait Set for holding the Tokens waiting for services and a Queue with capacity of one as its service Set for holding the Token being serviced.

The functionality of the base *Server::schedule (Token)* method is very simple. It puts the newly arrived Token into its wait Set if the service Set is full. Otherwise it puts the

Token into the service Set, lets it keep for one time unit for modeling the service activity. Then it drops the Token after the completion of the service, and schedules the service for another Token in its wait Set. The basic *Server::schedule (Token)* method intends to present a sample implementation for the user to show how a Server functions as a behavior controller for both Tokens' activities and resource management. It does not intend to provide an implementation that can be used directly by the user. Thus the user often needs to override this basic implementation in order to provide a specific functionality related to his/her application problem.

The following are the base classes and their interfaces.

class Token // **entities that require services from others**

```
Token ()                                  // constructors
Token (String type)
Token (TokenStats stats)
Token (String type, TokenStats stats)

int id ()                                 // unique Token identifier
String type ()                            // Token's type
double priority ()                        // Token's priority
Set currentSet ()                         // Token's current location or null if sent

void keep (double t)                      // scheduling an activity for the Token, i.e.,
                                          // the Token will be activated after time t.
void send (Entity dest, double delay)     // Token sent to destination Entity dest
                                          // and arrives there after time delay
Token move (Set destSet)                  // Token moves from currentSet() to destSet
                                          // in an Entity, returns the dropped Token.
void cancel ()                            // explicitly cancel Token's current activity,
                                          // the previous keep (...) and send (...).
```

class Set // **place for keeping Tokens in an Entity**

```
Set ()                                    // constructors
Set (Entity owner)
Set (SetStats stats)
Set (Entity owner, SetStats stats)
```

```
int id ()                          // unique Set identifier
String type ()                     // Set type
int capacity                       // max # Tokens in Set, can be infinite
int numTokens ()                   // current number of tokens in the Set
Entity ownerEntity ()              // current Entity which owns this Set
boolean full ()                    // true if numTokens () = capacity ()

abstract Token put (Token tk)      // put tk into Set, return the dropped Token
                                   // (could be tk) if Set has a finite capacity.
abstract Token get ()              // get a token out of the Set
abstract Token get (Token token)   // get a specific token out of the Set
abstract Token find (String type)  // find a token with the specific type
abstract Token find (int id)       // find a token with the specific id
```

class Queue extends Set // a Set in which Tokens are kept in a priority order

```
Token put (Token token)            // put a token into the Set
Token get ()                       // get a token out of the Set
Token get (Token token)            // get a specific token out of the Set
Token find (String type)           // find a token with the specific type
Token find (int id)                // find a token with the specific id

Token first ()                     // get first Token in the Queue
Token last ()                      // get last Token in the Queue
Token next (Token tk)              // get the Token next to tk in the Queue
Token prev (Token tk)              // get the Token before to tk in the Queue
```

class Entity extends SimKit.LP // abstract logical processes (LP)

```
Entity ()                          // constructors
Entity (EntityStats stats)

int id ()                          // unique Entity identifier
abstract void schedule (Token tk)  // called on Token entry, exit or the
                                   // completion of any task or service
```

class Server extends Entity // service nodes in a network of servers

```
Server (Set waitSet, Set serveSet)   // constructors
Server (EntityStats stats)
Server (Set waitSet, Set serveSet, EntityStats stats)

Set waitSet ()                     // server's entry and wait area
```

```
                                        // (default is a Queue).
Set serveSet ()                         // server's service area where keeping the
                                        // Tokens being serviced (default is a Queue).
void schedule (Token tk)                // a simple scheduler for allocation resource
                                        // to Tokens, is usually overridden by user

class Source extends Entity             // for injecting Tokens into the system

Source (Class clas)                     // constructors
Source (Class clas, String type)
Source (Class clas, String type, double p1)
Source (Class clas, String type, double p1, double p2)

String type ()                          // Source type: "Determ", "Uniform",
                                        // "EXP", "Normal".
Class template ()                       // the template of the dynamic entities
                                        // which are injected into the system.
double p1 ()                            // parameter 1 for inter-arrival time:
                                        // "Determ" - fixed inter-arrival time;
                                        // "Uniform" - lower bound;
                                        // "EXP" - mean, "Normal" - mean.
double p2 ()                            // "Uniform" - upper bound;
                                        // "Normal" - standard deviation.
double startTime ()                     // start time for functioning
Entity destination ()                   // default destination Entity for the Source
double delay ()                         // delay from Source to destination Entity

void schedule (Token tk)                // injecting a Token into the system and
                                        // scheduling the time for the next Token

class Sink extends Entity               // for absorbing Tokens in the system

int totalNumTokens ()                   // total # of Tokens absorbed so far
void addTotalNumTokens (int n)          // adding n to the totalNumTokens
void schedule (Token tk)                // counting for the totalNumTokens,
                                        // may be overridden by the user
```

4.3.2 Simulation Classes

Simulation classes in QueKit are used for conducting a simulation such as controlling the simulation process, collecting statistics and so on. They include QueSimulaton, Trace, Stats, TokenStats, SetStats, and EntityStats.

QueSimulaton is used for the simulation control. It is derived from class Simulation in SimKit. There is only one instance of QueSimulation can exist in a simulation. A simulation starts when the *run ()* method of this instance is called in the *main* function of the application program. The user often overrides its *initialize ()* method to instantiate all Entities in a simulation. Then the simulation control is handed to the underlying simulation kernel. The *terminate ()* method is called by the kernel when the simulation is ended because the simulation end time is reached or an error is occurred. The *terminate ()* method is often overridden by the user to collect final statistics of a simulation.

Trace class is designed for collecting trace information about Tokens' activities. The information about a Token's movement in the model during a simulation can be collected by setting up the Token's trace attributes. This is done by calling the method *Token::setTracing (attributes)*. There are four attributes for tracing: *EntityEntryTrace, EntityExitTrace, SetEntryTrace, SetExitTrace*. They can be set up simultaneously by bitmap or operation in *Token::setTracing (attributes)*. The information about the Token's activities related to any Resource can be collected by setting other four attributes: *MsgEntityEntryTrace, MsgEntityExitTrace, MsgSetEntryTrace, MsgSetExitTrace*.

Stats is the base class for class TokenStats, SetStats, EntityStats. Any Token can collect some basic statistics by attaching a TokenStats object. Accordingly, SetStats object is for the statistics collection in a Set, and EntityStats object is for the statistics collection in an Entity.

The main statistics collected by a TokenStats object for a Token are the number of Entities it passed, number of wait Sets as well as number of service Sets it entered. It also includes the average time and standard deviation of the time for the Token having stayed in any Entity, any wait Set, and any service Set in the model.

The main statistics collected by a SetStats object for a Set are number of Tokens passed, number of Tokens dropped in the Set, maximum and average Set occupancy, mean and standard deviation of the time for Tokens staying in the Set.

The main statistics collected by an EntityStats object for an Entity are number of Tokens passed, average and standard deviation of a Token staying in the Entity.

class QueSimulation extends Simulation // simulation control

Entity getEntity (int id) // get an Entity with the id
Enum getEntities () // get the Entity list

class Trace // trace class for collecting trace information about Tokens

Trace () // constructors
Trace (String name)

int id () // unique Trace identifier
String fileName () // trace file name

void print (String info) // output a line of information to the trace file

class Stats // base class for collecting statistics

Stats () // constructor

int id () // unique Stats identifier

class TokenStats extends Stats // for collecting stats for Tokens

TokenStats (Token token) // constructor

Token ownerToken () // the Token that the Stats object belongs to
int numWaitSets ()
int numServeSets ()
int numEntities ()
double localWaitTime ()
double localServeTime ()
double globalWaitTime ()
double globalServeTime ()
double meanSetWaitTime ()
double stdvSetWaitTime ()
double meanSetServeTime ()
double stdvSetServeTime ()
double meanEntityWaitTime ()
double stdvEntityWaitTime ()

double meanEntityServeTime ()
double stdvEntityServeTime ()

void reset ()
void resetLocal ()

void entry (Entity e)
void exit (Entity e)
void entry (Set s)
void exit (Set s)

class SetStats extends Stats // for collecting stats for Set

SetStats (Set set) // constructor

Set ownerSet () // the Set that the Stats object belongs to
int maxOccupancy ()
int meanOccupancy ()
int throughput ()
int numDropped ()
double utilization ()
double meanServeTime ()
double stdvServeTime ()

void entry (Token tkn)
void exit (Token tkn)

class EntityStats extends Stats // for collecting stats for Entity

EntityStats (Entity entity) // constructor

Entity ownerEntity () // the Entity that the Stats object belongs to
int throughput ()
double meanServeTime ()
double stdvServeTime ()

void entry (Token tkn)
void exit (Token tkn)

4.3.3 Server Architecture

A model is mainly composed of Tokens and Servers in the server architecture. Tokens model the entities that need services from other entities, and Servers model entities that

provide those services. In the server architecture, Tokens flow through a network of Servers to get services from these Servers. Servers are active objects in the server architecture for both resource management and Tokens' activity control. A Server decides when and how to allocate resources to Tokens and arranges their activities through its *schedule (Token)* method.

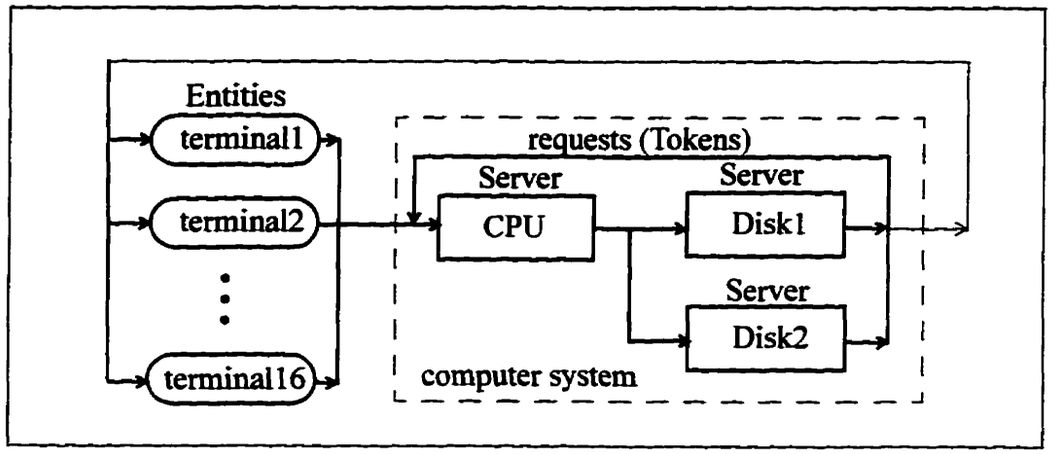


Fig. 4.5 Timesharing System

For instance, in a small timesharing system [19], there are 16 user terminals connected to a minicomputer system with two disks. A user thinks for some time, sends a request to the computer system, and waits for the system response. When the response is received, the user thinks again and then initiates another request. The process of a request may cost several CPU and disk operations alternatively in the computer system. Therefore, this timesharing system can be modeled with the server architecture shown above. The CPU and two disks are modeled as Servers. Terminals (sub-systems) are modeled as Entities and users' requests are modeled as Tokens.

4.4 Extended Layer

The above passive Token vs. active Server approach in the server architecture is convenient for modeling the scenario in which a master scheduler in a Server takes care of

everything: allocating resources to a Token, scheduling the completion of a service for the Token, and eventually routing (sending) the Token to another Server. When a Token needs more than one resource simultaneously in an activity and the allocation strategies of these resources are different, e.g., one is based on first-come first-serve (FCFS) and another is based on Tokens' priorities, the *schedule (Token)* of the Server will become quite complicated and lack clarity.

For example, in a hospital system [6], there are limited beds in the hospital ward for accepting patients. A patient is admitted to the ward if there is a bed available, otherwise s/he has to wait for a bed. S/he stays in the ward for some time for the treatment. S/he may need an operation after the treatment. There is only one theatre in the hospital. The patient has to stay in the ward for a while after the operation and then returns his/her bed and is discharged from the hospital. The system is illustrated below.

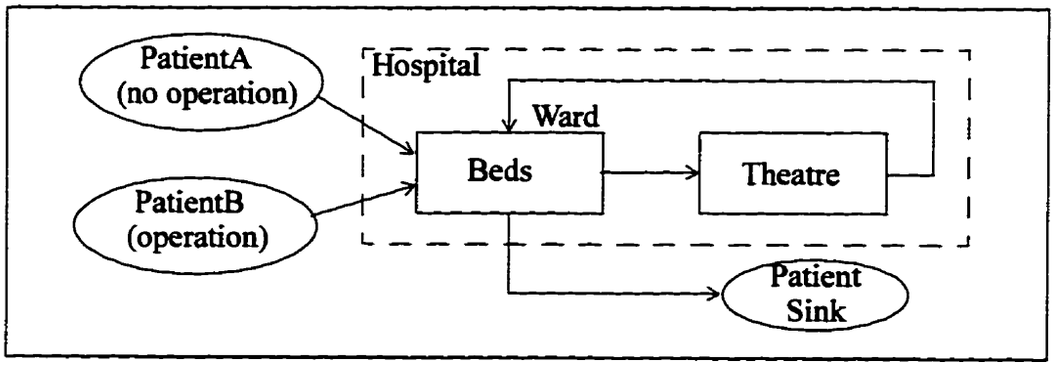


Fig. 4.6 Hospital System Diagram

With QueKit server architecture, patients can be modeled as Tokens in the hospital system. However, the theatre and the ward cannot be modeled as different Servers because a patient who is in an operation still reserves his/her bed in the ward. Thus the ward and the theatre have to be modeled as a single Server shown as following.

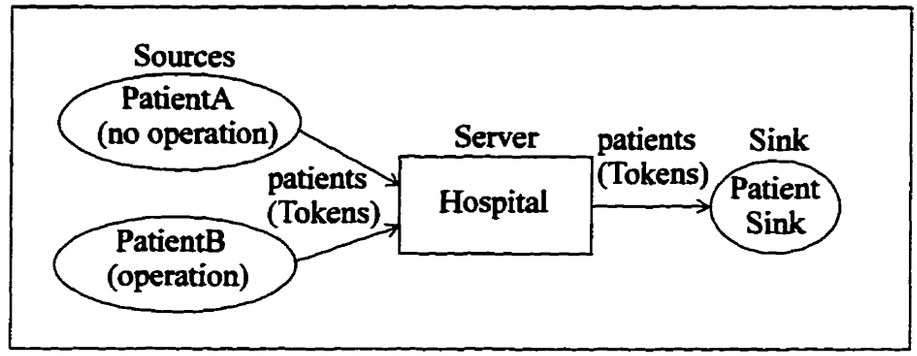


Fig. 4.7 Hospital System (Server Architecture)

There are drawbacks for modeling the hospital system in above server architecture. Since the management of the ward and the theatre in the hospital may be relatively independent of each other. With the server architecture, all patients's activities and resource allocation/deallocation of beds and the theatre are forced to be described in a single function (Hospital Server's *schedule* method). The results are: (1) the Hospital Server's *schedule* method is relatively complicated; (2) the natural parallel activities in the system are forced to be totally serialized in a simulation even if the model is run on a parallel machine.

The extended layer of QueKit will solve this problem by separating the functionality of a Server into two parts, one for the resource management and another for the Tokens' activity management. This layer contains the following classes that are derived from QueKit base classes.

4.4.1 Extended Classes and Interfaces

Extended classes include Job, Client, and Resource. Job is derived from Token. A Job models an active entity that request services from other entities in a queueing system. Client is derived from Entity. A Client models a behavior controller for a set of Jobs that behave in the same way. The *schedule* method of a Client describes how these Jobs get resources and conduct their activities in a chronological order during their lifetime or

period of their lifetime. A base Client has a host Set for holding the Jobs which behave in the same way while they are in the host Set.

Resource is derived from Server and it models a sub-system for a single sort of entity which provides service for other entities. A Resource is a specialized Server and it is only responsible for handling the resource allocation/deallocation requests from Jobs.

The base functionality of the *Resource::schedule (Token)* method is dedicated to the allocation/deallocation of the resource. Upon receiving a Message (derived from Token) for resource request, it sends the Message back to its host Entity when the resource is enough, otherwise it puts the Message into its wait Set. Upon receiving a Message for resource return, it recovers its resource amount and drops the Message. The user can override this base functionality in order to provide more complicated one for his/her application problem.

class Job extends Token **// active entities that require service from others**

```
// requesting a specific amount resource from res with a specific priority,
// default amount = 1.0, default priority = 0.
void requestRes (Resource res)
void requestRes (Resource res, double amount)
void requestRes (Resource res, double amount, double priority)

// preempting a specific amount resource from res with a specific priority,
// default amount = 1.0, default priority = 0.
void preemptRes (Resource res)
void preemptRes (Resource res, double amount)
void preemptRes (Resource res, double amount, double priority)

// returning a specific amount resource to res, default amount = 1.0.
void freeRes (Resource res)
void freeRes (Resource res, double amount)
```

class Client extends Entity **// activity controller for Jobs**

```
Set hostSet ()          // host Set for all Jobs in the Client
void end ()             // mark the end of the lifetime of Client,
                        // and destroy it.
```

class Resource extends Server // sub-system for resource allocation/deallocation

```
double capacity ()          // maximum amount of resource
double amount ()            // current amount of resource

void schedule (Token msg)   // allocating/deallocating resource to the
                            // Token associated to msg
```

4.4.2 Client Architecture

A model is mainly composed of Jobs, Resources, and Clients in the client architecture. Client class is derived from Entity class as described above. A Client class provides a behavior control profile which represents the lifetime activities of a group of Jobs whose behaviors are statistically identical. The behavior of the group of Jobs is described by the *schedule* method of a Client class. A typical *schedule* method is the description of a series of actions about Jobs' lifetime activities in a chronological order. Resource class is also derived from Entity class. A Resource object functions as a passive resource controller in the sense that it only controls resource allocation/deallocation but has no control on Jobs' activities and how long resource will be possessed by Jobs. Thus the *schedule* method of a Resource object is dedicated to resource allocation/deallocation. In a Client, a Job decides how long it possesses the resource by its *keep (Time)* method and returns the resource by its *freeRes (Resource)* method later.

The following model of the above hospital system depicts this kind of distributed control in which the management of an independent resource is handled in a Resource object and the lifetime activities of the same kind of Jobs are handled in a single Client object. The solid lines are the routes of Jobs and the dashed lines indicate the communications between Clients and Resources.

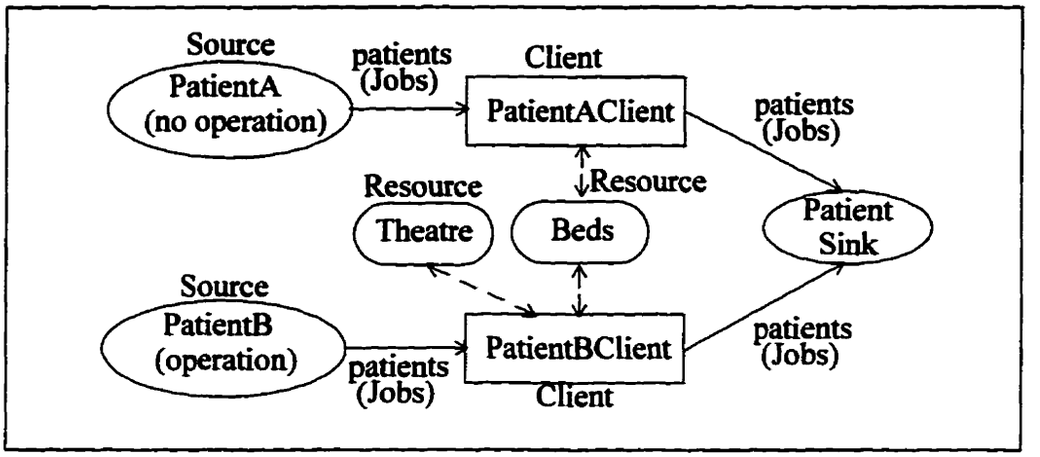


Fig. 4.8 Hospital System (Client Architecture)

The above model is easier to build than the model shown in Fig. 4.7 because the complexity of the *schedule* method of the single Hospital Server in Fig. 4.7 is distributed in four objects (PatientAClient, PatientBClient, Beds, and Theatre). Furthermore, the above model exploits more natural parallelism of the system than the model shown in Fig. 4.7 because the four objects can be dispatched into at most four different processors and executed in parallel.

Nevertheless, the execution of a model built with the client architecture is not as efficient as that with the server architecture in sequential environment. This is because at least two Messages (two events involved) for a resource allocation and one Message (one event involved) for a resource deallocation are needed in a Client while only one invocation of *schedule (Token)* in a Server (one event involved) can allocate and deallocate several resources.

Therefore, there are trade-offs between the clarity of model presentation and the execution efficiency of model when choosing an architecture for model construction. When the allocation of resources is simple, i.e., the assumption of a master controller for resource allocation/deallocation is close to the situation in reality, the server architecture is preferred. This is because it guarantees good execution efficiency while preserving good

clarity for the model. On the other hand, when the allocation of resources is complex, e.g., the allocation/deallocation of those resources are independent of each other with different strategies such as FIFO and priority, the client architecture is preferred. In this case, the functionality of the complex *schedule* method of a Server is decomposed into three parts: (1) a simple *schedule* method in a Client which chronologically describes the lifetime activity of Jobs in a straightforward way; (2) a dedicated resource controller in a Resource for handling the allocation/deallocation requests from Jobs; (3) communications between Clients and Resources via Messages. (2) and (3) are mostly provided by the QueKit package. Thus the work of the application programmer spent on model construction and debugging will be greatly reduced and the model will be more readable so that the productivity of the software development is increased from the view point of the software engineering.

Hence, part of the model execution efficiency seems to be scarified for good model clarity in the client architecture. However, the decomposition of the functionality of a centralized Server results the model inheriting more natural parallelism from the modeled system. This is because a Job's activity is independent of resource allocation/deallocation to another Job so that they can be handled simultaneously in a parallel environment. Thus the model execution efficiency for a model built with the client architecture may be equivalent to or better than that of a model built with the server architecture for modeling the same system when running in a parallel environment.

4.4.3 Server-Client Architecture

Using one of the above architectures solely in a model construction may not be enough to exploit all potential parallelism of a real system being modeled while preserving good model clarity. The server-client architecture allows both server and client architectures to be applied seamlessly in constructing a single QueKit model.

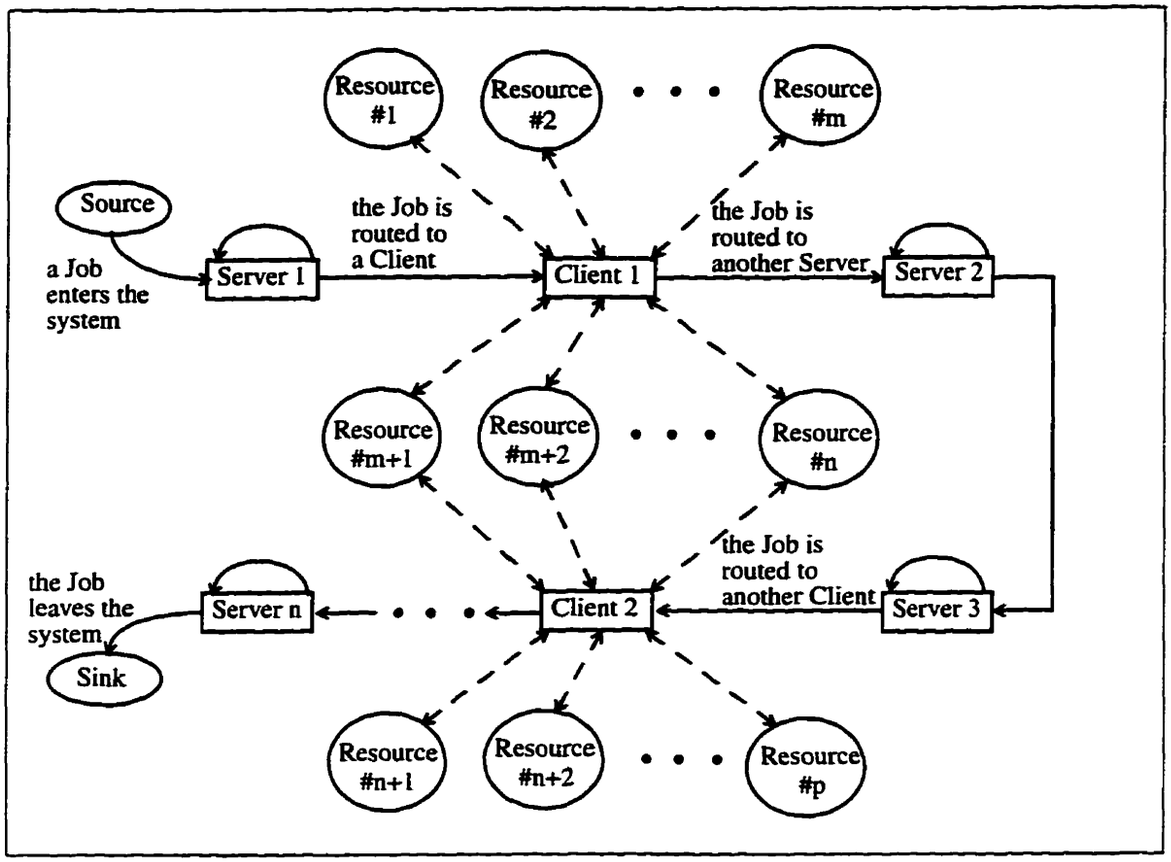


Fig. 4.9 Server-Client Architecture

In the server-client architecture, the lifetime activity of a Job is divided into several periods in a chronological order according to the way how resources are handled in the real system and the clarity of describing the handling of these resources in the *schedule* methods. Each activity period is described in a *schedule* method of a Server or a Client. Jobs flow through a network of Servers and Clients to get services and complete their life-time journey. As depicted in Fig. 4.9, a Job may start from a Server, travels through a series of Servers to get services. Then it is sent to a Client, experiences a sequence of activities inside the Client, and is finally sent to another Server. This pattern may repeat until the Job finishes its lifetime journey and is absorbed in a Server or a Client.

4.5 Implementation Issues

The QueKit package is built upon the SimKit package. Therefore, any primitive in QueKit is implemented at the application level of SimKit. The pseudo-code for implementing three methods of Token class and two methods of Job class are presented in this section.

Both simulated delay (*Token::keep (Time)*) and event cancellation (*Token::cancel()*) are associated with *process (Event)* method in Entity class. This method is inherited from Entity's parent class LP in SimKit for handling events. It is defined as a final method so that it cannot be overridden by the application programmer. In this method, the *Entity::schedule (Token)* function is invoked only when the processed event is effective, i.e., its associated Token has not cancelled the event. This is done by comparing the event identification number of the processed event and the event identification number recorded in the associated Token. The processed event is effective only when these two numbers are the same. Otherwise, the processed event has been cancelled either implicitly by multiple *Token::keep (Time)* or explicitly by *Token::cancel()*.

```
public void process (Event ev) {
    QEvent Qev = (QEvent)ev;
    // ignores any canceled event, only activates the Token which is associated
    // with an effective event (i.e., there is a Token currently associated with it)
    if (Qev.token().id() == Qev.id())
            schedule (Qev.token());
}
```

Fig. 4.10 Pseudo-Code for Entity::process (Event)

4.5.1 Simulated Delay

Since the OO event-driven view is used in QueKit, the event object is transparent to the user, thus any activation of a Token by *Entity::schedule (Token)* is associated with a

process of an event object inside an Entity. The simulated delay (*Token::keep (Time)*) is usually for modeling the passage of a specific amount of time. For the three base classes in QueKit, i.e., Token, Entity, and Set, only Token can be associated with the simulated delay primitive. This is because *Entity::keep (Time)* can only be implemented in the process view as the context switching is necessary for controlling the passivation and activation of a process to model the passage of time. And *Set::keep (Time)* makes no sense here as a Set is only a place for holding Tokens.

Token::keep (Time) models a Token to be engaged into an activity for a specific amount of time in an Entity, e.g., the Token is being serviced in a Server. A Token must reside in a Set of an Entity when it is engaged in an activity. An Entity can have multiple Tokens being engaged in their activities simultaneously, and each of which is associated to an effective event. A Token can have only one outstanding *keep* effective. Although a Token can issue multiple *keep*s, the latest *keep* will make its previous *keep* ineffective.

```
void Token::keep (Time keepTime) {
   if ((keepTime < 0) or (currentSet == null)) {
         error handling
   } else {
         QEvent ev = new QEvent (this);// associate the event with this Token
         eventId = ev.id();// record the effective event id
         // schedule the event arriving at its owner Entity after time keepTime
         ev.send_and_delete (ownerEntity(), currTime() + keepTime);
   }
}
```

Fig. 4.11 Pseudo-Code for Token::keep (Time)

4.5.2 Token Movement

There are two types of movement for a Token in a QueKit model. One is modeling a Token moving from one Entity to another. For instance, a Token is routed to another Server after it finished the service in the current Server. This is done by invoking *Token::send (Entity, Time)*. It causes the Token to be sent to a specific Entity and reach that Entity after the specific time of delay.

```
void Token::send (Entity dest, Time delay) {
   if ((delay < 0) or (dest == null)) {
         error handling
   } else {
         remove the Token from its current Set if it is in a Set
         QEvent ev = new QEvent (this);// associate the event with this Token
         eventId = ev.id();       // record the effective event id
         // schedule the event arriving at its destination Entity after time delay
         ev.send_and_delete (dest, currTime() + delay);
   }
}
```

Fig. 4.12 Pseudo-Code for Token::send (Entity, Time)

Another type of movement is modeling a Token moving back and forth between different Sets in the same entity. *Token::move (Set)* causes a Token to be moved into a specific Set in an Entity. This type of movement does not involve any delay as the first type, i.e., a Token reaches the destination Set immediately after it leaves the current Set. If any non-zero delay should be modeled in the movement between different Sets in the same Entity, or the movement is between different Sets in different Entities, *Token::send (Entity, Time)* should be used instead of *Token::move (Set)*.

```
Token Token::move (Set set) {
    Token token = null;
    if ((set == null) or (set.ownerEntity() != currentSet.ownerEntity())) {
            error handling
    } else {
            if (currentSet != null)
                    currentSet.get (this);// remove Token from its current Set
                    token = set.put (this);// put Token into the Set set
    }
    return token;
}
```

Fig. 4.13 Pseudo-Code for Token::move (Set)

4.5.3 Event Cancellation

The traditional approach to event cancellation is to remove the event to be cancelled directly from the event list so that the cancelled event will never be processed in the future. This mechanism is very simple and efficient, but it makes some software packages like SimPack [12] unable to be used in a parallel system based on Time Warp paradigm.

```
void Token::cancel (Token token) {
    if ((token.currentSet == null) || (token.currentSet != null) &&        \
                    (currentActiveEntity == token.ownerEntity()))
            // make the Token not associating to any event (-1 is a invalid event Id)
            token.eventId = -1;
}
```

Fig. 4.14 Pseudo-Code for Token::cancel (Token)

The approach used in QueKit is to make the event cancellation transparent for both underlying simulation kernel as well as the application programmer. That is, the event

cancellation is handled inside QueKit in a simple way: QueKit simply ignores the cancelled events whenever these events are encountered (shown in Fig. 4.11). A Token's activity can only be canceled by its owner Entity, i.e., the Entity which scheduled the event by invoking *Token::keep (Time)* or *Token::send (Entity, delay)* before.

4.5.4 Resource Allocation/Deallocation

A Resource functions as a shared variable in a QueKit model. Thus any access to a Resource should be via messages in an LP view.

The following methods of a Job are associated to the resource allocation/deallocation: *requestRes (Resource), requestRes (Resource, amount), requestRes (Resource, amount, priority), preemptRes (Resource), preemptRes (Resource, amount), preemptRes (Resource, amount, priority), freeRes (Resource), freeRes (Resource, amount)*. Messages are used for the communications between Resources and other Entities for resource allocation/deallocation. However, messages are also transparent from the user.

Job::requestRes (Resource, amount, priority) causes a Message sent to the specific Resource for requesting the specific amount of resource at a specific priority. The default amount is one unit and default priority is zero if they are not specified. A message sent from a Job to a Resource functions as a representative of the Job competing for the resource in the Resource object.

```
void Job::requestRes (Resource res, double amount, double priority) {
    Message msg;
    if ((amount <= 0) and (res == null)) {
            error handling
    } else {
            // create a Message associated with this Job and its owner Entity,
            // set its type to "QK_Request" and let it carry the requested amount
            msg = new Message ("QK_Request", this, ownerEntity, amount);
            msg.setPriority (priority);
```

```
            msg.send (res, currTime());// send the Message to the resource Entity
        }
    }
```

Fig. 4.15 Pseudo-Code for Job::requestRes (Resource, amount, priority)

Job::preemptRes (Resource, amount, priority) causes a Message sent to the specific Resource for preempting the specific amount of resource at a specific priority.

```
void Job::preemptRes (Resource res, double amount, double priority) {
    Message msg;
    if ((amount <= 0) and (res == null)) {
            error handling
    } else {
            // create a Message associated with this Job and its owner Entity,
            // set its type to "QK_Preempt" and let it carry the requested amount
            msg = new Message ("QK_Preempt", this, ownerEntity, amount);
            msg.setPriority (priority);
            msg.send (res, currTime());// send the Message to the resource Entity
    }
}
```

Fig. 4.16 Pseudo-Code for Job::preemptRes (Resource, amount, priority)

Job::freeRes (Resource, amount) causes a Message sent to the specific Resource for returning the specific amount of resource. The default amount is one unit if it is not specified.

```
void Job::freeRes (Resource res, double amount) {
    if ((amount <= 0) and (res == null)) {
            error handling
    } else {
```

```
            // create a Message associated with this Job and its owner Entity,
            // set its type to "QK_Return" and let it carry the requested amount
            Mesage msg = new Message ("QK_Return", this, ownerEntity, amount);
            msg.send (res, currTime());// send the Message to the resource Entity
        }
    }
```

Fig. 4.17 Pseudo-Code for Job::freeRes (Resource, amount)

4.6 Summary

QueKit aims to provide an OO design and implementation environment for queueing system simulation to facilitate the modeling process. QueKit also support efficient sequential execution, as well as the potential for parallel execution. The library-based approach is used for the development of QueKit so that OO techniques can be applied at abstraction, design, and implementation level. The modeling framework provided in QueKit is OO event-driven. Dynamic entities are modeled as Tokens or Jobs and static entities are modeled as Servers or Resources. The LP view is used for the implementation of QueKit so that it can be executed in a parallel environment either optimistically or conservatively. Three architectures are provided by QueKit for the model construction: server architecture, client architecture, server-client architecture.

Chapter 5

Preliminary Evaluation of QueKit

Fourteen modeling problems mentioned in Chapter 2 will be revisited here for the preliminary evaluation of the QueKit abstraction. Moreover, two benchmark systems will be described in detail for a preliminary evaluation of the QueKit abstraction, as well as, its performance. Different QueKit architectures are also used for the model constructions in order to compare the modeling power and execution efficiency among these architectures. Furthermore, both systems are also modeled with the SimKit application programmer's interface (API) for comparison.

5.1 Revisiting the Common Modeling Problems

All the common modeling problems listed in Chapter 2 are revisited with examples in this section in order to show how QueKit can be used to solve these problems. Event graphs are used as declarative models to describe the dynamics of these problems.

Resource Sharing: This problem has been addressed with an example in Sec. 4.3.3 (Fig. 4.5 Timesharing System). The server architecture is used there. Actually the client architecture can also be used for modeling this system except the execution efficiency will not be as good as that of the server architecture. With the client architecture, the *CPU* and two *disks* are modeled as Resources. Sixteen *terminals* are modeled as Clients and user *requests* are modeled as Jobs. The following declarative model indicates the life cycle of a terminal Client. It actually describes the *schedule* method of a terminal client.

A block in a declarative model represents an event in which the bold text shows the event type while the rest of the text shows the actions upon the occurrence of the event. More details about actions in an event block are described with the QueKit application programmer's interface (API) in a declarative model. As in Chapter 2, a solid arrow in any declarative model represents an activity within an object while a dashed arrow represents

an activity involving different objects. One event block marks the beginning of the activity at the tail of the arrow, and another event block marks the end of the activity at the head of the arrow.

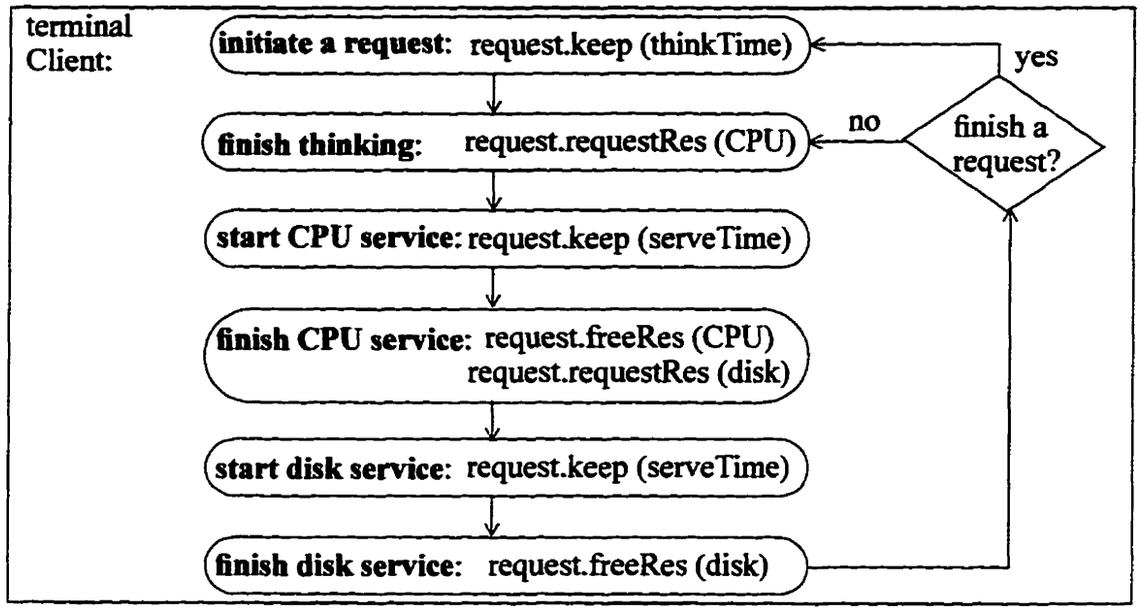


Fig. 5.1 Declarative Model for the Timesharing System (QueKit Client Architecture)

A request Job in the above figure is kept for "thinkTime" to model the user's thinking time in the "initiate a request" event. It then requests to access the CPU when the thinking activity, which is represented by the first top-down arrow, is finished. The request initiates a CPU activity, which is represented by the second top-down arrow, once it gets the CPU. It releases the CPU and requests a disk when the CPU activity is finished. A "start disk service" event occurs once the request gets the disk. A "finish disk service" event occurs when the disk service activity is finished. The request Job will be back to request the CPU again if its process is not completed, otherwise it will be back to model another user's thinking activity again.

Preemption: This problem will be addressed with the CPU_Disk system in Sec. 5.2.

Grouping and Loss: There is a truck with capacity of 50 boxes in a transportation system to transport goods from city A to city B. It takes 3 hours for the truck with goods travelling from City A to B and 2.5 hours for the empty truck travelling back to City A from City B. Suppose the truck starts from City A. Assuming that a box of goods arrives at the city A at an inter-arrival time exponentially distributed with mean of 0.1 hours. The truck has to wait until it is fully loaded, and then begin its trip from city A. It takes 30 minutes to load/unload the truck in city A/B. Goods may decay anytime and decayed goods are thrown away. The lifetime is exponentially distributed with mean of 10 hours.

The server architecture is suitable for modeling this system. A *box of goods* is modeled as a Token, the *truck* is modeled as a Server with a service Set of capacity of 50. The *storage place in city A* (StorageA) is also modeled as a Server while the *storage place in city B* (StorageB) is modeled as an Entity because it is simple. A special signal can be used to model the arrival of the empty truck at City A. This signal also informs City A that the truck is ready for loading. The declarative model is shown below. It shows the *schedule* member functions for Entity StorageA, Server StorageB, and Server truck respectively.

The lifetime limit of a box of goods can be recorded inside the boxes Token when it arrives at city A. The StorageA Server will throw away the decayed goods whenever the truck is ready for loading and the number of boxes is enough for loading the truck. The actual loading activity will begin when there are still enough boxes for loading after cleanup of all of the decayed goods. The StorageB Entity can collect the following five statistics by testing the time period when the lifetime of a box of goods expires: 1) number of boxes decayed before loading; 2) number of boxes decayed during the loading process in City A; 3) number of boxes decayed during the transportation from City A to City B; 4) number of boxes decayed during the unloading process in City B; 5) number of boxes transferred in good shape. A box of goods is decayed before loading if its lifetime limit is shorter than its loading time. A box of goods is decayed in transit in city A/B if its lifetime limit falls in loading/unloading activity. A box of goods is decayed during the transporta-

tion if its lifetime limit falls in the activity of the trip from city A to B. Otherwise, the box of goods is transferred to city B in good shape.

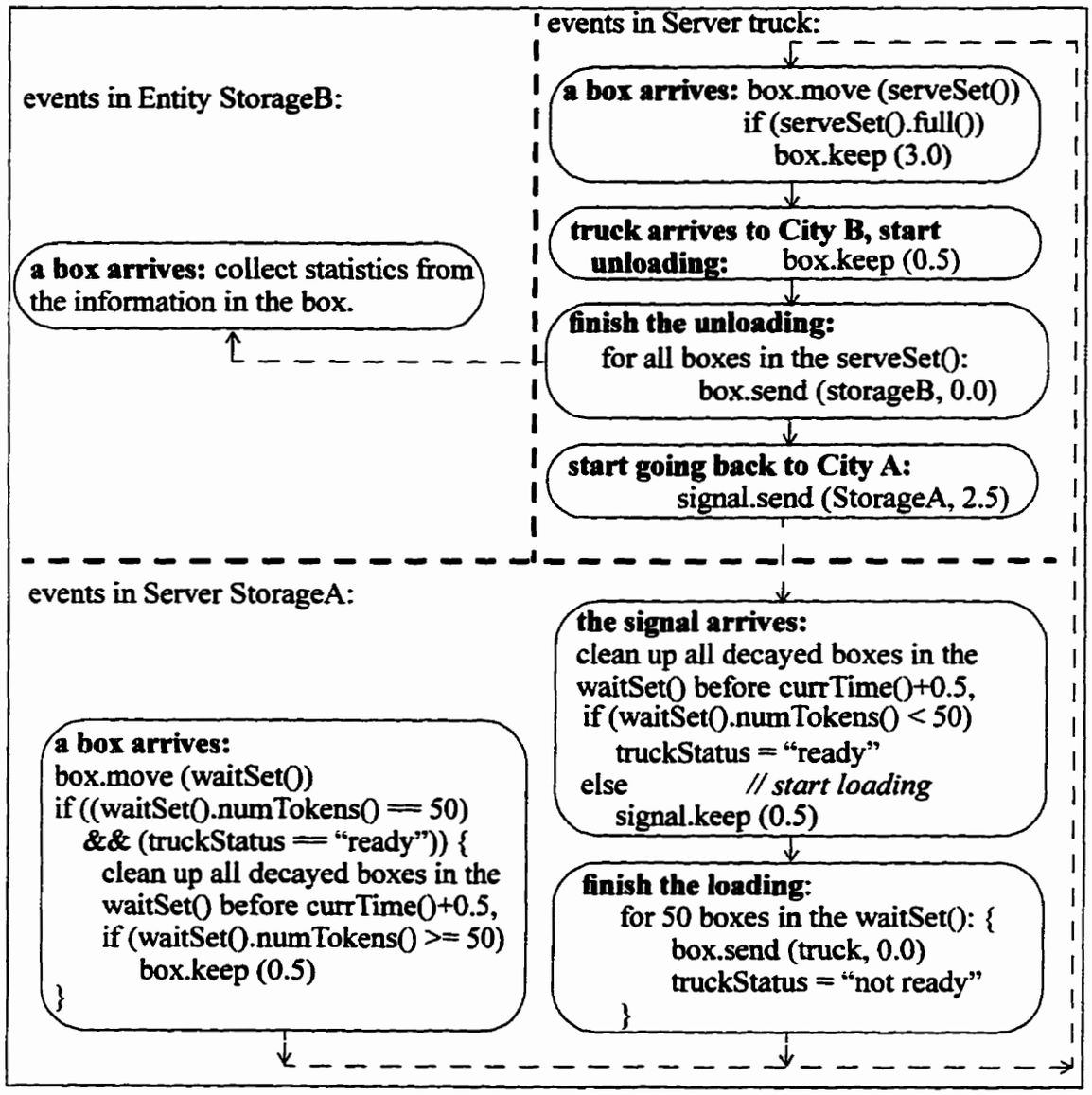


Fig. 5.2 Declarative Model for the Transportation System

Routing and Dropping: An ATM network can be used for studying these two problems. Suppose the simplified output buffer switches are used in this network. The routing

of an ATM cell in a switch is done by looking up the virtual path identifier (VPI) table. A cell will be discarded (dropped) by the switch if the output buffer is full. The server architecture is suitable for modeling this system. An ATM cell is modeled as a Token. A switch is modeled as an object in which each input port is modeled as an Entity and each output port is modeled as a Server. The declarative model is shown below.

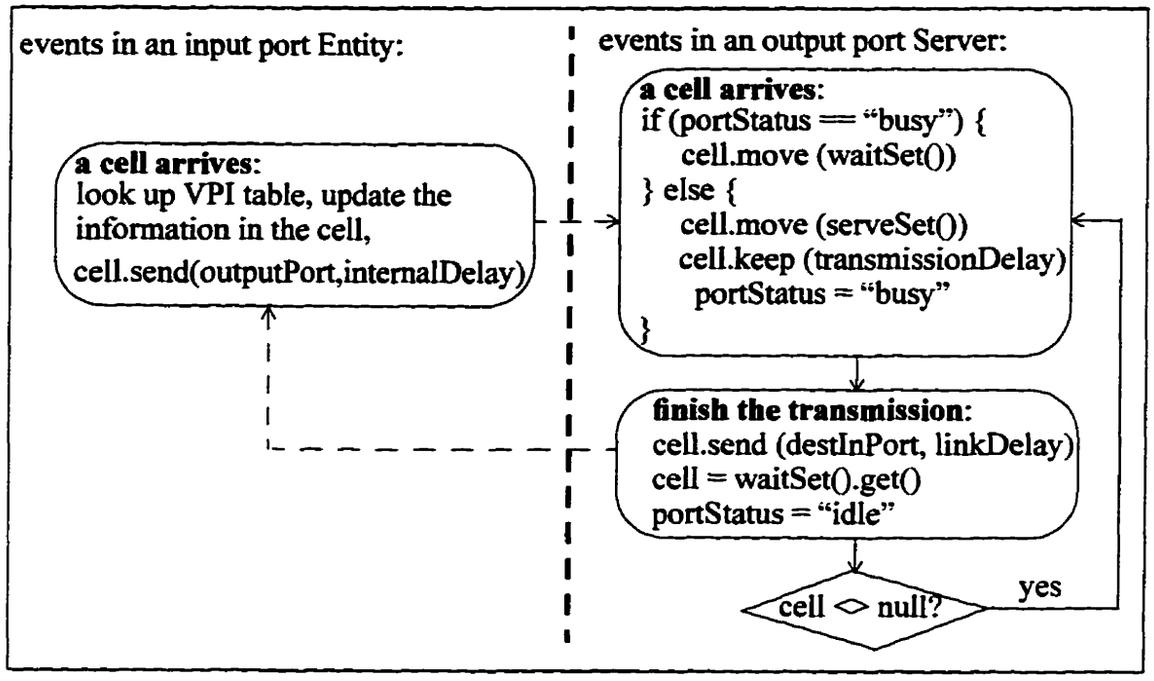


Fig. 5.3 Declarative Model for the ATM Network

Tandem Queueing[1]: Faulty units are sent for repairing to a special section in a factory. The repairing is carried out in two stages - first the unit is stripped down, and then it is rebuilt.

Each operation has its own work station. Work station A (stripping) can work on two units at a time while work station B (rebuilding) on one unit at a time. But storage is limited, and at most 4 units can be queued in front of station A, and 2 in front of station B. If

1. This system comes from G.M.Birtwistle's book [5] with a little modification.

4 units are already queued in front of station B, a newly arrived faulty unit is subcontracted. When a strip job is completed, the unit is automatically moved to the area in front of station B when there is a room over there. Otherwise, station A is blocked until a space is freed.

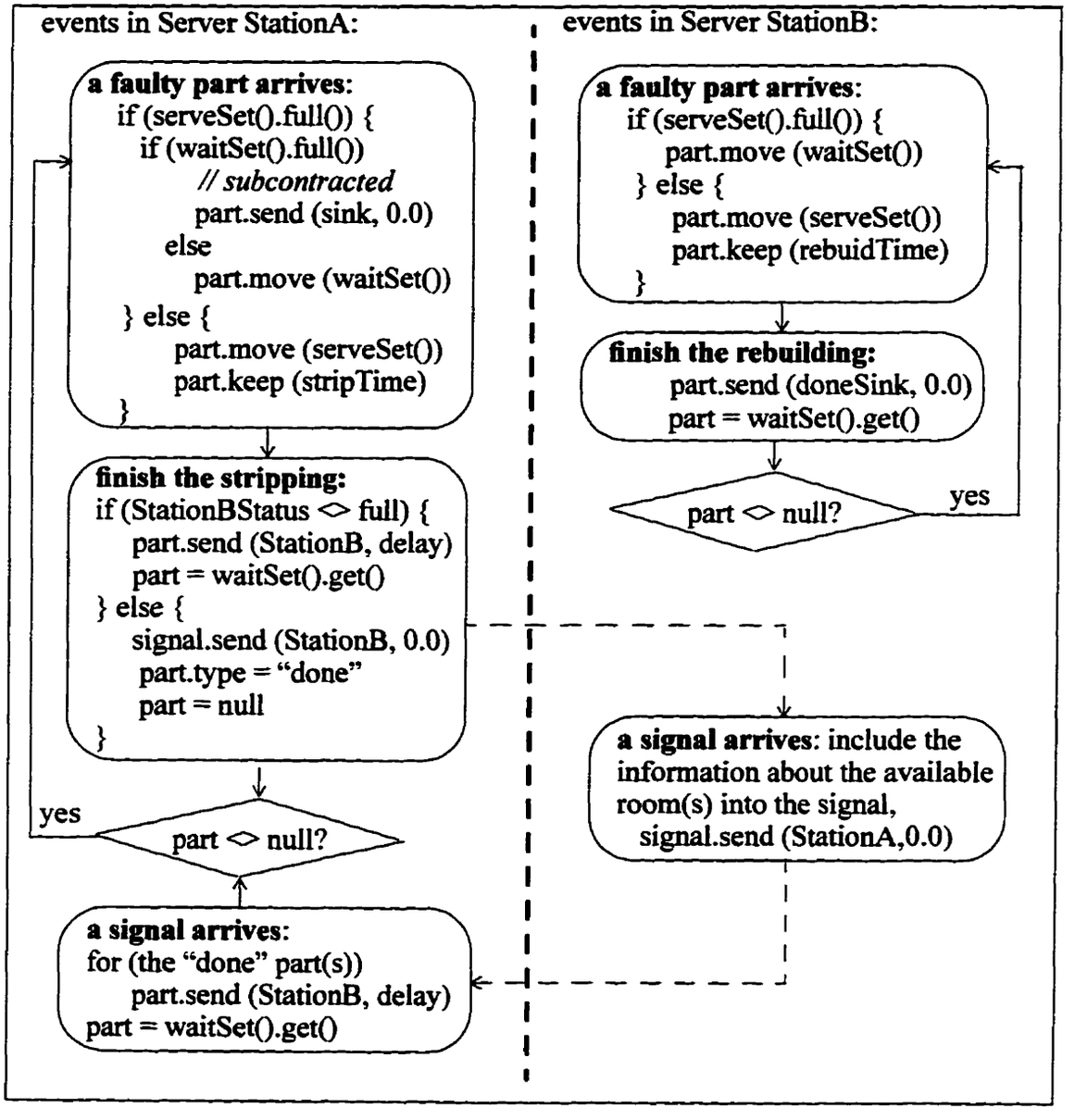


Fig. 5.4 Declarative Model for Factory System

The server architecture is suitable for modeling this system. The faulty units are modeled as Tokens, two repair stations are modeled as Servers. The station A Server has a waitSet with capacity of 4 and a serveSet with capacity of 2. The station B has a waitSet with capacity of 2 and a serveSet with capacity of one. A special signal Token is used for modeling the communications between the two station Servers. The declarative model is shown in Fig. 5.4.

Affinity: Customers arrive at a barber shop at an inter-arrival time exponentially distributed with a mean of one minute. There are four barbers providing services for the customers. The incoming customers usually choose the shortest waiting list to wait for service from a barber. However, about 10% customers favor Barber #2 because s/he is the most skillful one in the shop. These special customers are determined to wait for service from him/her no matter how long his/her waiting list is.

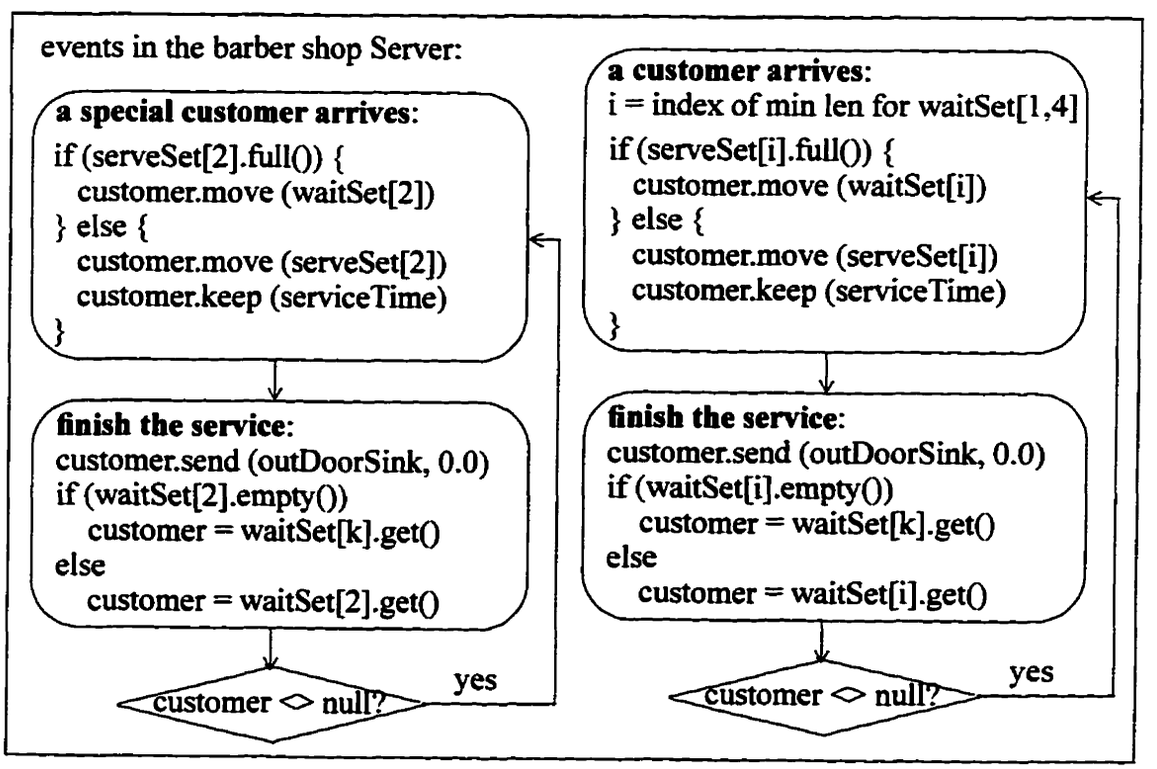


Fig. 5.5 Declarative Model for the Barber Shop System

The server architecture is suitable for modeling this system. Customers are modeled as Tokens and the barber shop is modeled as a Server here. There are four Queues as the waiting Sets with infinite capacity in the Server. Each Queue models a list of customers waiting for a specific barber. There are also other four Queues as the service Set with capacity of one in the Server. Four Tokens in these queues model the four customer who are in the services. The declarative model is shown in Fig. 5.5.

Balking, Migration, and Reneging: There are two checkout counters in a grocery story. Suppose customers always choose the shortest line to wait for checking out. If the number of people in the shortest line is more than 6, there is 10% probability for a newly arrived customer giving up the checking out and leaving. Also suppose there is 30% probability for a customer leaving her/his waiting line and giving up the checkout if s/he has waited in the line for a long time (maxWaitTime) when neither of the waiting lines is not empty. Otherwise s/he will migrate to the empty waiting line when the maxWaitTime is passed.

The server architecture is used here. Customers are modeled as Tokens, the checkout system is modeled as a Server. In the Server, there are two Queues with infinite capacity as the waitSets for modeling two waiting queues for the two checkout counters respectively. There are also two Queues (checkout[1] and checkout[2]) with the capacity of one for modeling two checkout counters respectively. The declarative model is shown below.

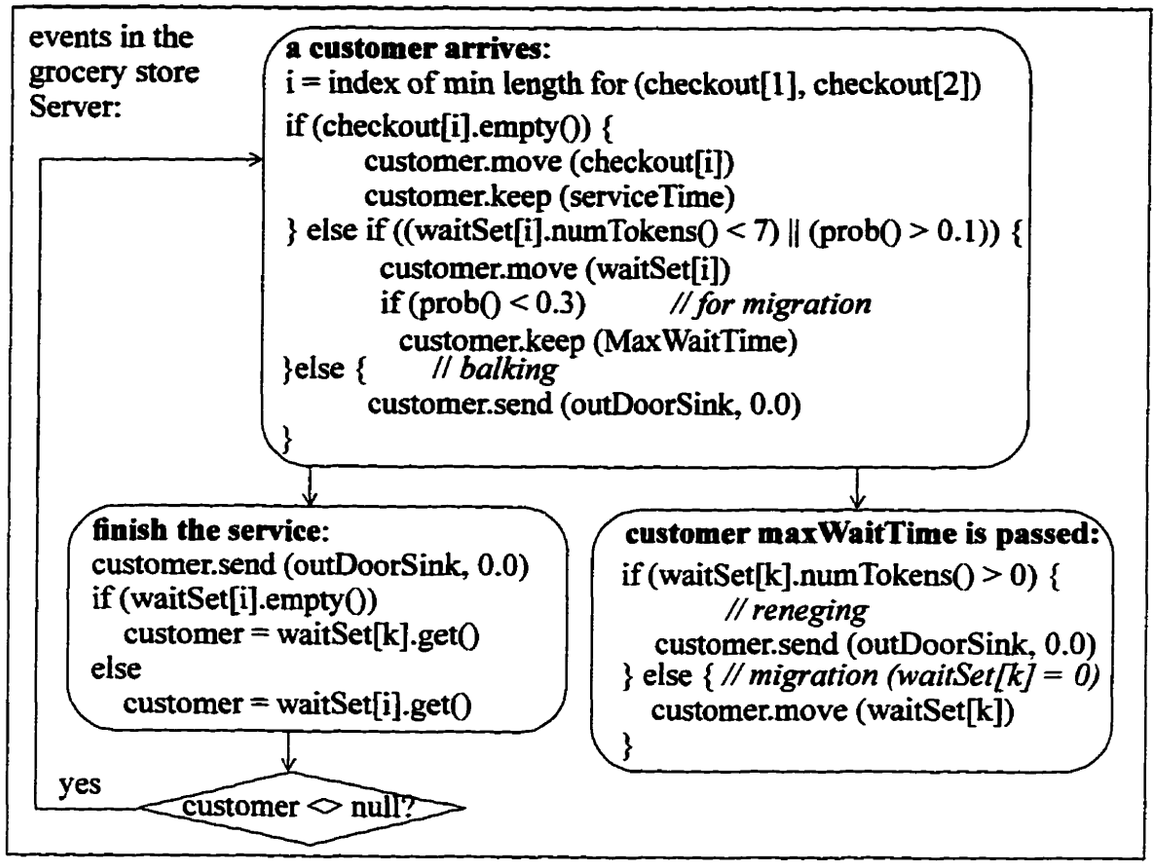


Fig. 5.6 Declarative Model for the Grocery Store System

Multiple Resources: The server architecture is suitable for modeling this kind of problems when the multiple resources required in an activity are actually managed by a controller. For example, a computing job needs the CPU, some memory, and some disk space. The operating system controls the management of these resources. Thus the jobs can be modeled as Tokens and the operating system can be modeled as a Server that is in charge of the management of the CPU, memory, and disk resources. On the other hand, the client architecture is suitable for modeling those problems in which the resources are independently managed. A harbor system will address this problem in Sec. 5.2.

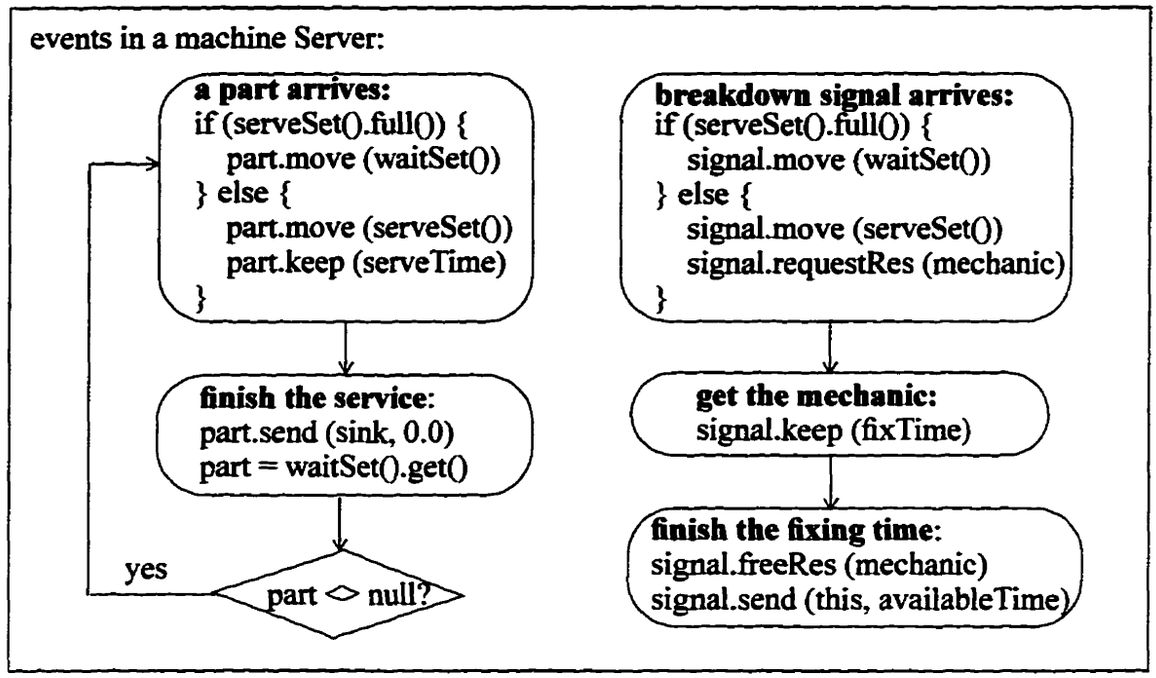


Fig. 5.7 Declarative Model for the Factory System

BreakDown and Entity Transformation: The server-client architecture is used for modeling the factory system mentioned in Sec. 2.3. Parts are modeled as Tokens, machines are modeled as Servers and the mechanic is modeled as a Resource. A machine Server has a wait Set with infinite capacity for holding parts waiting for service, and a service Set with capacity of one for holding the part being serviced. Suppose a machine is able to complete a service before a breakdown event happens. A breakdown event for a machine Server is modeled as a Job which randomly and periodically arrives at the machine Server to cause it to be broken down. It will restore the machine to the normal status after it is fixed by the mechanic. The breakdown event Job functions as a special signal that has a higher priority than parts arriving at a machine Server. The declarative model is shown above.

In summary, QueKit can model all the above problems that commonly exist in the simulation of queueing systems in a way that is very close to the dynamic behaviors of the

modeled systems. The user has to identify the objects that require services from others and the objects that provide these services. Then the former objects can be modeled as Tokens or Jobs and the latter objects can be modeled as Servers or Resources depending on what architecture is chosen. The Client class may be used for describing the lifetime activities or periods of lifetime activities for Jobs if the client architecture or the server-client architecture is applied. Finally, the *schedule* member functions of these Servers and Clients are written. They describe how the Servers handle the services for the incoming Tokens and route them to other Servers after services, and how the Clients describe the Jobs' activities in a chronological order to model the Jobs flow through the modeled system. These Tokens, Jobs, Servers, Clients, and Resources are gathered together with a single QueSimulation object to form a complete model that is ready to execute after proper parameters are set up for a simulation run.

Among the three QueKit architectures, the server architecture is flexible enough to model all of above fourteen problem scenarios. It will only make a model more complicated and result in an unnatural description of the modeled system when multiple independent resources are involved in a single activity. This is because it forces these independent resources to be managed under a single Server controller which does not exist in reality. On the other hand, the client architecture is suitable for modeling systems that involve multiple independent resources.

5.2 Benchmark Models

The CPU_Disk system and the Harbor system are presented in this section. These two systems are chosen for evaluation in detail because the CPU_Disk system covers preemption while the Harbor system covers requesting multiple independent resources in an activity. Preemption is related to event cancellation. The client architecture is suitable to be applied in the modeling of a system that requires multiple independent resources in an activity. Therefore, all the basic classes (Token, Set, Entity, Queue, Server, Source, and Sink) and extended classes (Client and Resource) will be used in modeling these two sys-

tems. Both functional and declarative models are used for the design of the simulation models in this section.

5.2.1 CPU_Disk System

The model description is in Sec. 2.2.1 (refer to Fig. 2.1). Four models are built for this system. One model is built with the SimKit API for comparison. Three models are built with the QueKit API: first one is built with the server architecture; second one is built with the server-client architecture (source code of is at Appendix C); last one is built with the client architecture. The same statistics are collected in all four models. Although any parameter for n0 or n1 can be used for the experiments, in this study, n0 is six and n1 is two in the above four models.

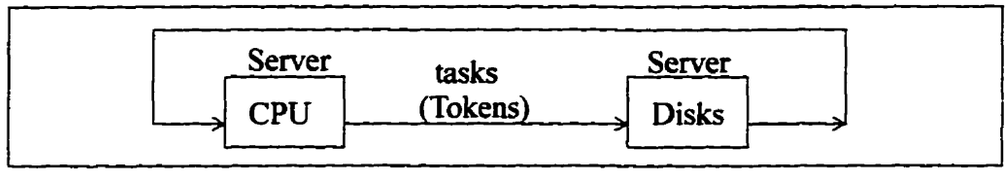


Fig. 5.8 Functional Model for the CPU_Disk System (Server Architecture)

In the *server architecture*, tasks are modeled as Tokens, the CPU is modeled as a Server. Tasks of class 0 have a priority 0 while tasks of class 1 have a priority 1. The CPU has a Queue with infinite capacity as the waitSet which holds the tasks waiting for services, and has a priority queue with capacity of one as the serveSet which holds the task being serviced. The four disks are modeled as a single Server. It has four priority queues with infinite capacity as the waitSets which hold the tasks waiting for services for the four disks respectively. It has also four priority queues with capacity of one as the serveSets which holds the tasks being serviced for the four disks respectively. In the CPU scheduling, a preempted task of class 0 will increase its priority from 0 to 0.5, record its remaining service time, and move back to the waitSet of the CPU. This will guarantee that it is placed ahead of any other task of class 0 and after any task of class 1. The preempted task will restore its priority to 0 when it finishes its service in the CPU. The functional model is

depicted below. The solid lines are the routes of the tokens. The declarative model was shown in Fig. 2.3.

In the *client architecture*, tasks are modeled as Jobs. The CPU is modeled as a Resource with capacity of one. Each of the four disks is also modeled as a Resource with capacity of one. The tasks with priority of one can preempt the CPU while the tasks with priority of zero can only request the CPU. The preemption is automatically handled in the CPU Resource. The functional model is depicted in Fig. 5.9. The dashed lines indicate the communications between TaskClient and Resources. The declarative model was shown in Fig. 2.5.

For the model with QueKit server architecture or SimKit API, there are two kinds of events for the CPU/Disk scheduling: a task requests for service from CPU/Disk and the completion of a service in CPU/Disk. For the model with QueKit client architecture, there are five kinds of event for the tasks' activities: 1) requesting/returning the CPU; 2) completing a service in the CPU; 3) releasing the CPU and requesting one of the disks; 4) completing a service in the disk; 5) releasing the disk and going back to 1).

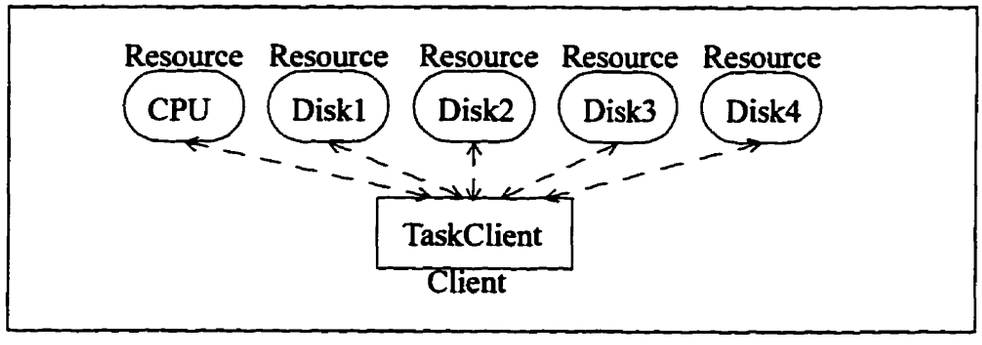


Fig. 5.9 Functional Model for the CPU_Disk System (Client Architecture)

In the *server-client architecture*, tasks are still modeled as Jobs and four disks are modeled as Resources as that in the above client architecture. The CPU, however, is modeled as a Server as that in the server architecture. The life cycle of a task is divided into two periods: one starts from the time waiting for a CPU service and another starts from the time waiting for a disk service. The first period is modeled with the server architecture and the second period is modeled with the client architecture. The following are the functional model and declarative model.

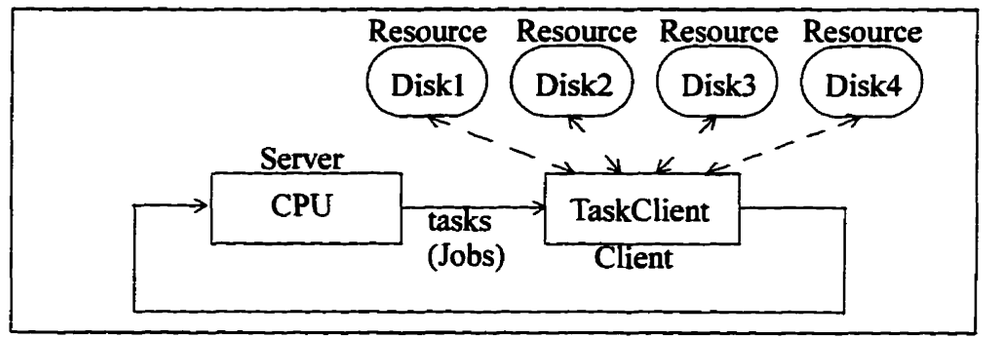


Fig. 5.10 Functional Model for the CPU_Disk System (Server-Client Architecture)

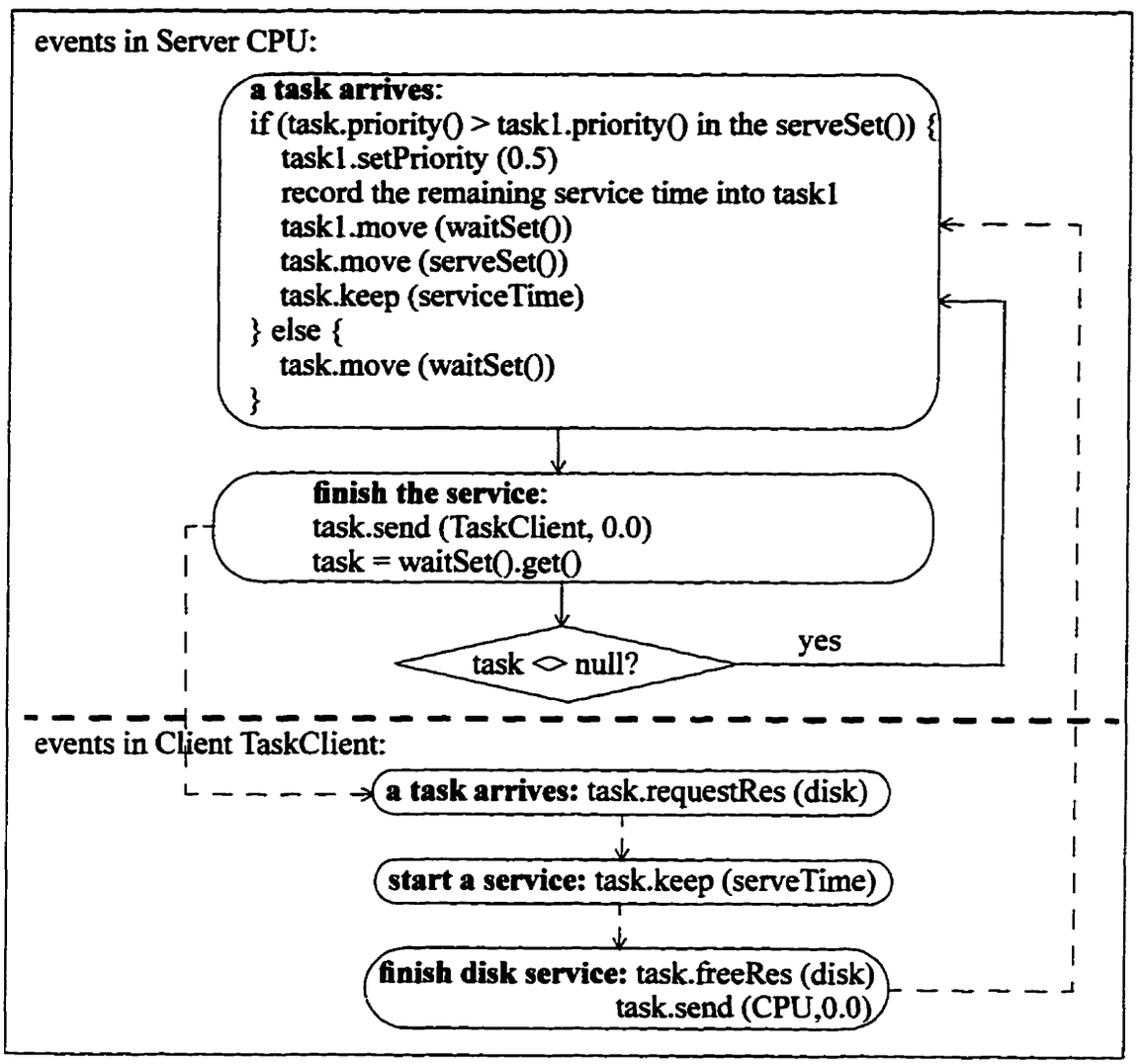


Fig. 5.11 Declarative Model for the CPU_Disk System (Server-Client Architecture)

The CPU_Disk System is simple and thus suitable to use all three QueKit architectures for modeling. Therefore, there are not much differences of modeling power among these three architectures for modeling this system. However, the execution efficiency will be different for using these architectures. The results will be discussed in Sec. 5.3.

5.2.2 Harbor System

There are two jetties and three tugs in a harbor for servicing incoming boats[1]. Boats arrive at the harbor at an inter-arrival time exponentially distributed with the mean of 18 time units. They must pass an inspector for security check before each of them can request for a jetty for docking. The time for the inspector to check a boat is also exponentially distributed with the mean of 3 time units. When a jetty is available, a boat may dock and start to unload. When this activity is completed, the boat leaves the jetty and sails away. Two tugs are required for docking and only one tug is required for leaving. Assume that tug maneuvers take 2 time units, and unloading takes 14 time units.

Four simulation models are built for this system. One model is built with SimKit API for comparisons. Three models are built with the QueKit API, one with the server architecture, one with the server-client architecture (source code is at Appendix D), and another with the client architecture. Same statistics are collected in the four models.

In the *server architecture*, boats are modeled as Tokens. The inspector is modeled as a Server having a priority queue as the waitSet with infinite capacity and another priority queue as the serveSet with capacity of one. The harbor is modeled as a Server with a service Set for holding the boats being serviced and two wait Sets, one for holding the boats waiting for the tug and another for holding the boats waiting for jetties. The tug and jetties are modeled as two integer variables in the harbor Server representing the resources. The functional and declarative models are shown below.

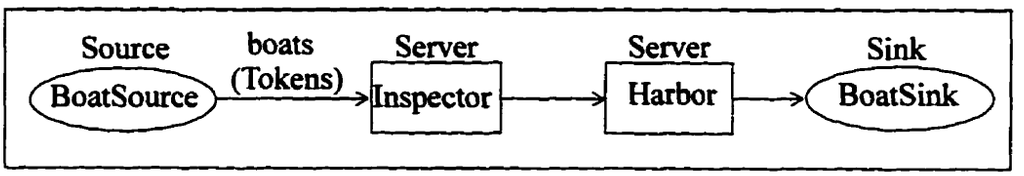


Fig. 5.12 Functional Model for the Harbor System (Server Architecture)

1. This system comes from G.M.Birtwistle's "Port System" example [5] with a little modification.

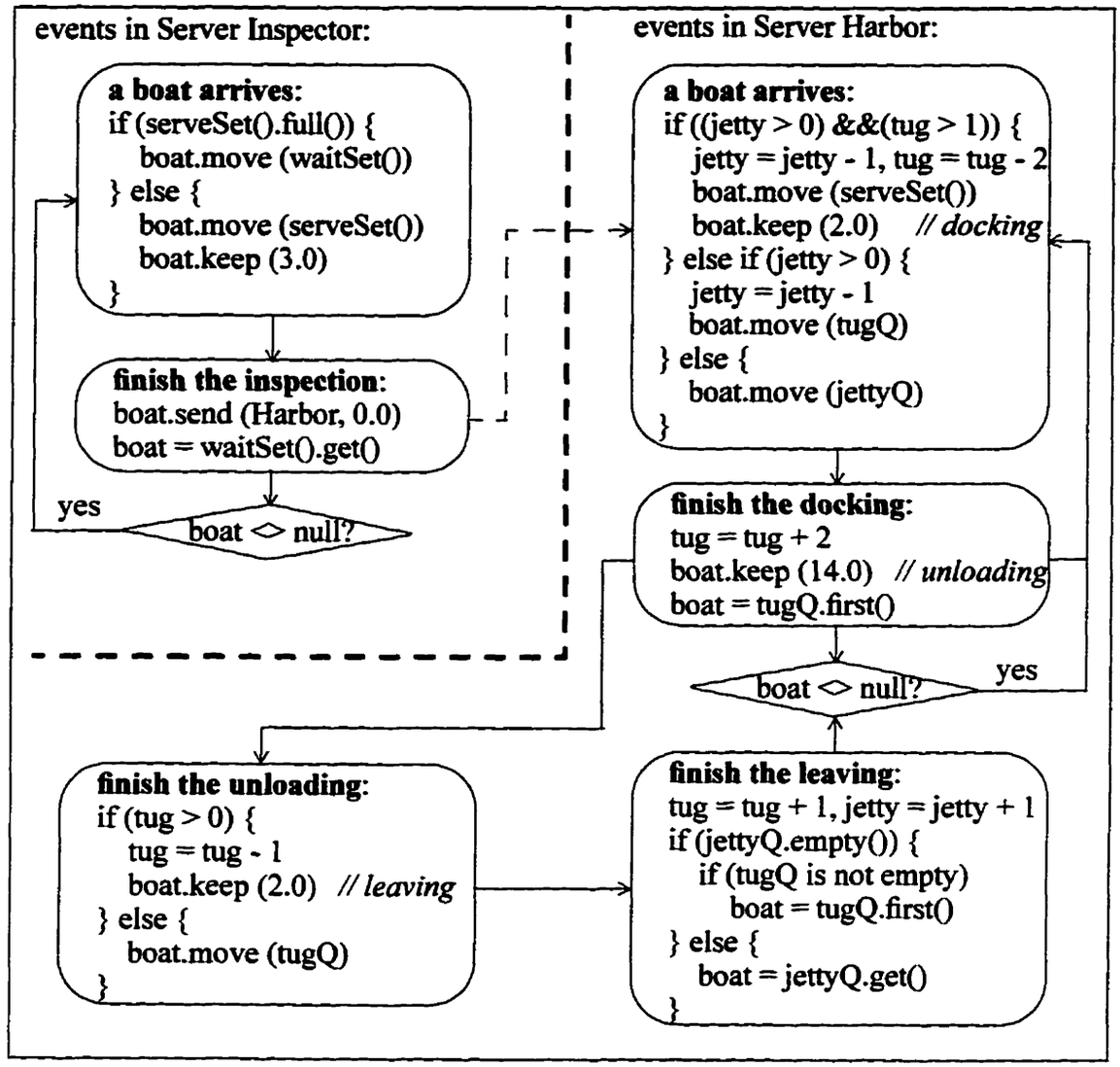


Fig. 5.13 Declarative Model for the Harbor System (Server Architecture)

In the *client architecture*, boats are modeled as Jobs. The inspector is modeled as a Resource with capacity of one. Jetties are modeled as a Resource with capacity of two, and tugs are modeled as a Resource with capacity of three. The lifetime activities of a boat are described in a BoatClient object. In the functional model shown below, the solid lines are the routes of the boats, and the dashed lines indicate the communications between BoarClient and Resources.

5.14 F

Fig 5.15

In t
modeled

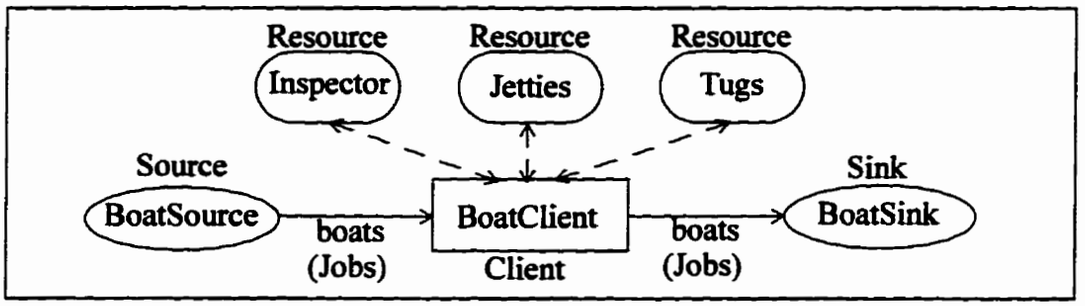


Fig. 5.14 Functional Model for the Harbor System (Client Architecture)

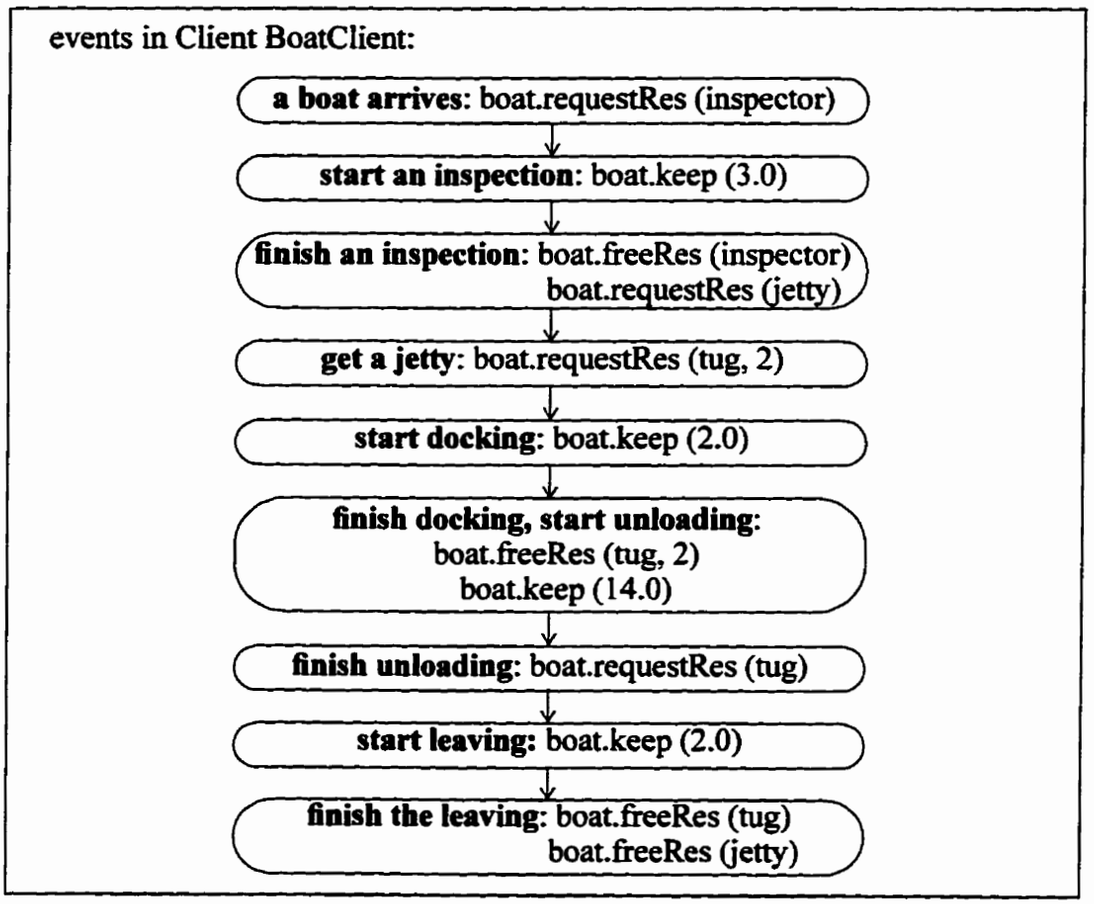


Fig. 5.15 Declarative Model for the Harbor System (Client Architecture)

In the *server-client architecture*, boats are again modeled as Jobs. The inspector is modeled as a Server having a queue with infinite capacity as the wait Set and a queue with

capacity of one as the service Set. Jetties and tugs are still modeled as Resources with capacity of two and three respectively.

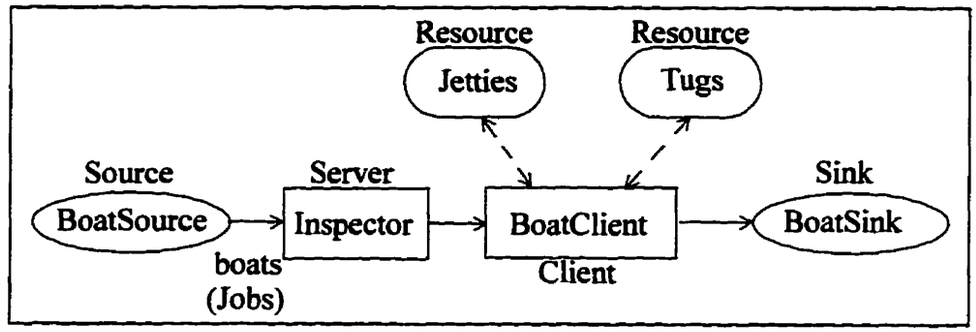


Fig. 5.16 Functional Model for the Harbor System (Server-Client Architecture)

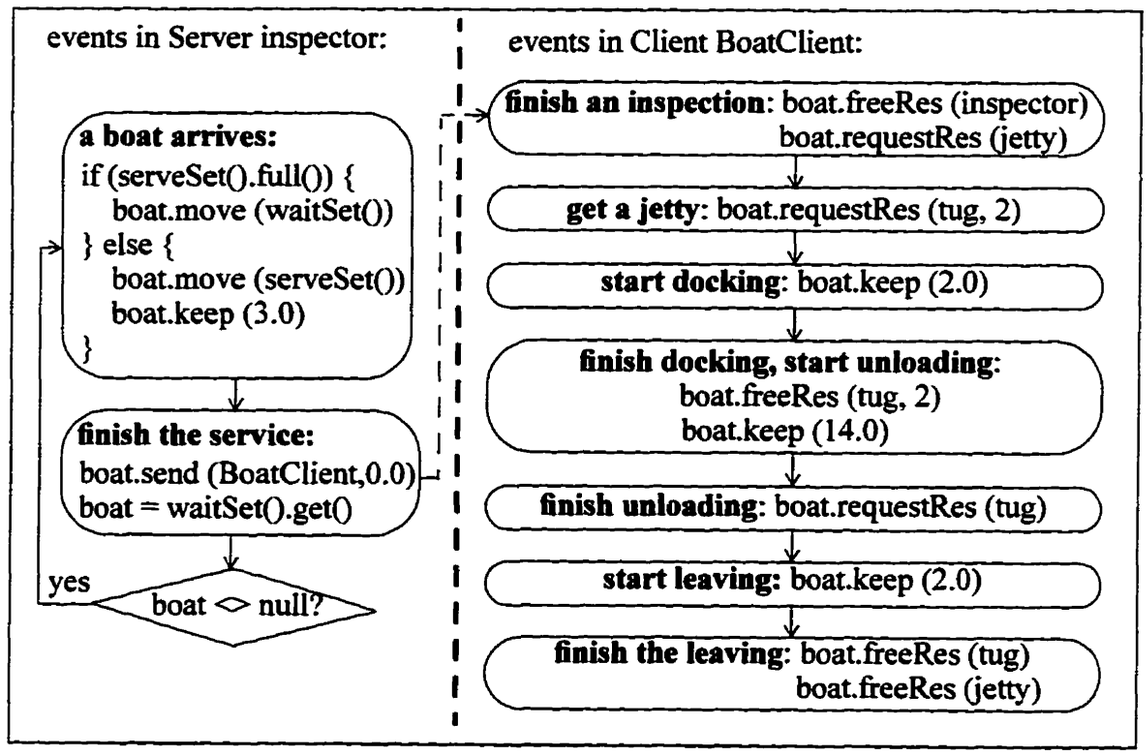


Fig. 5.17 Declarative Model for the Harbor System (Server-Client Architecture)

The lifetime of a boat is divided into two periods: one starts from waiting for the inspection, another starts from requesting a jetty after the inspection. The first activity period of a boat is modeled in the server architecture in which the Inspector server controls the services for boats and routes them to the BoatClient. The second activity period of a boat is described in a chronological order by the BoatClient object below. The functional model and the declarative model are shown above.

The above three declarative models show the dynamics of the three simulation models built with three QueKit architectures respectively. Subjectively, the model with the client architecture shown in Fig. 5.15 is easier to understand than the model with the server architecture shown in Fig. 5.13. This is because the former describes the lifetime activities of a boat in a chronological order that is very close to the sequence of activities that occur during a boat's lifetime within the harbor system. The latter, however, describes these activities in two Servers in which the event logic is different from the problem description. This subjective view of the client architecture is close to Birtwistle's presentation of the "process view" of simulation which is often considered as one of the most natural ways to construct a model. Therefore, the modeling power of the client architecture is greater than that of the server architecture in the aspect of comprehensibility when multiple resources are required during a single activity. For the same reason, the modeling power of the server-client architecture is between that of the server architecture and the client architecture.

5.3 Modeling and Simulation Results

The simulation models are constructed with the QueKit API (Java version). Thus the simulation experiments are executed sequentially. The results for the CPU_Disk system and the Harbor system are illustrated in the following graphs and tables. An overhead (%) is the measure of a QueKit model execution time comparing to the execution time of the corresponding SimKit model. It is calculated with the following formula for each QueKit architecture:

Overhead (%) = 100 • (QueKit Run Time - SimKit Run Time) / SimKit Run Time (EQ 1)

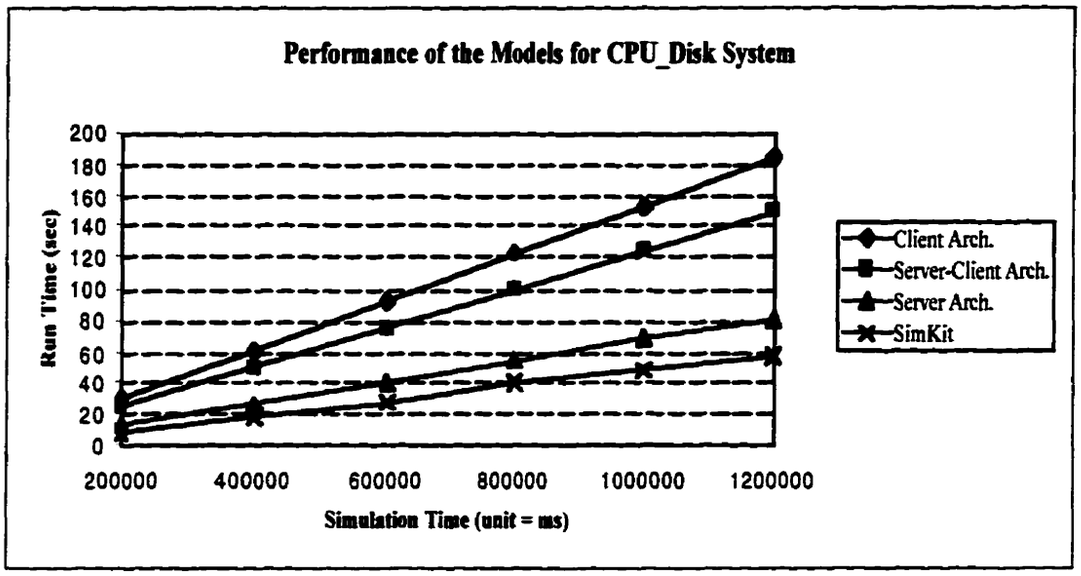


Fig. 5.18 Performance of the Models for CPU_Disk System

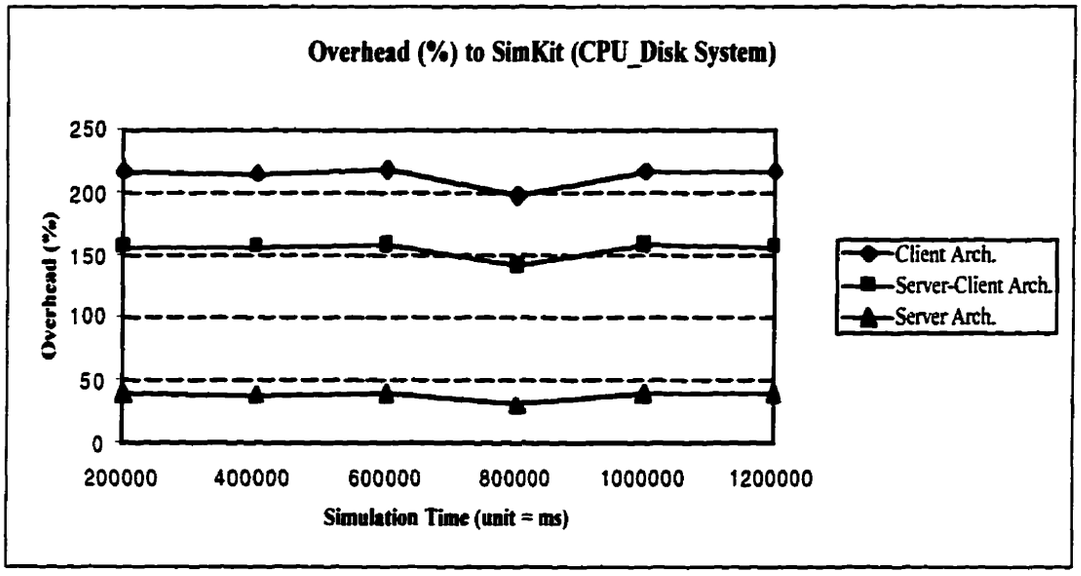


Fig. 5.19 Model Overhead (%) to SimKit Model (CPU_Disk System)

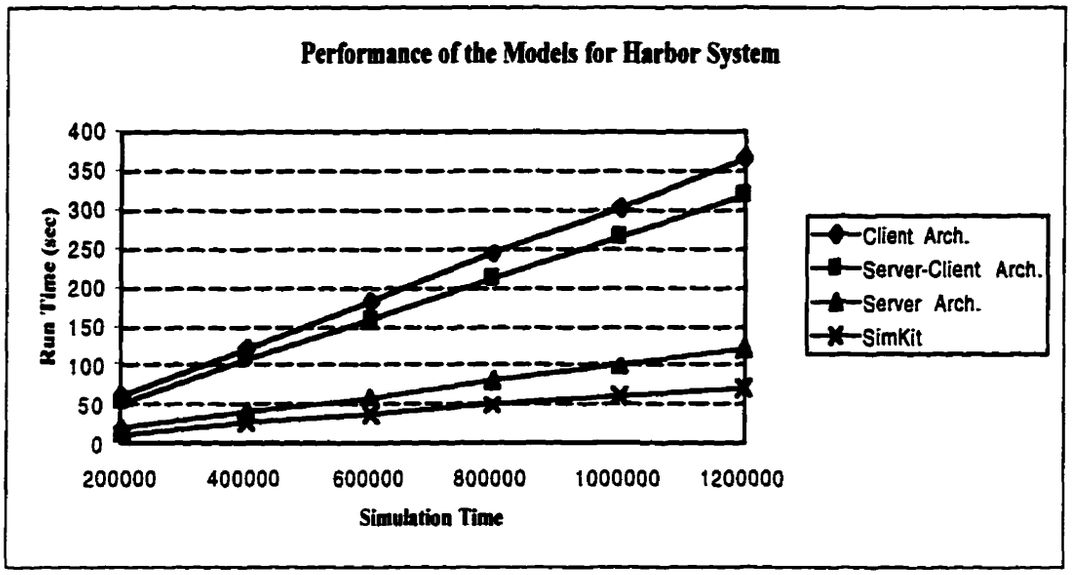


Fig. 5.20 Performance of the Models for Harbor System

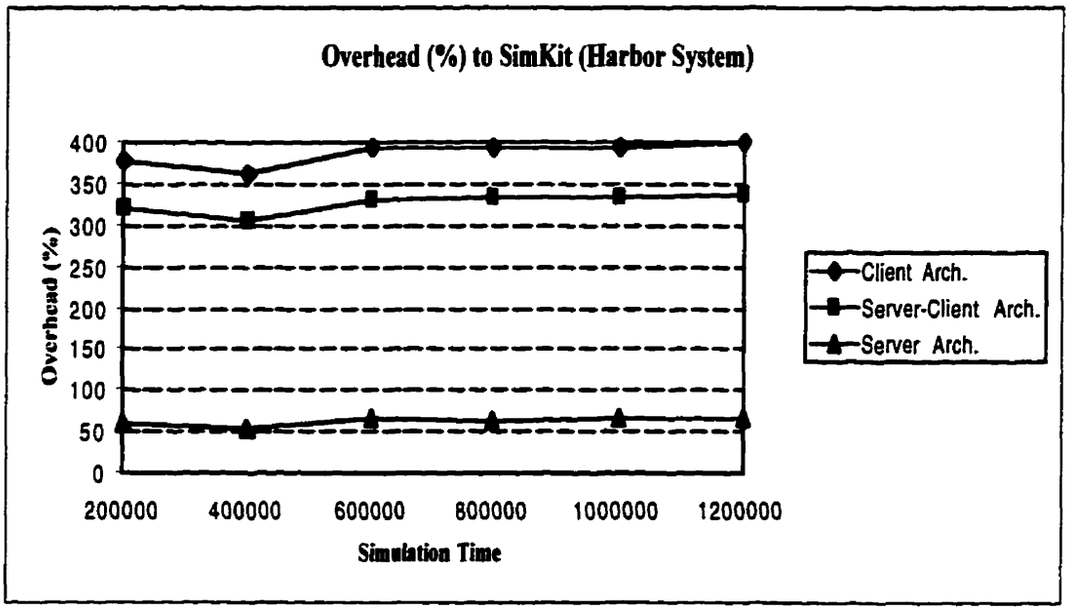


Fig. 5.21 Model Overhead (%) to SimKit Model (Harbor System)

The above four figures show that the execution efficiency of QueKit server architecture is not as good as that of SimKit, but it is much better than that of QueKit client archi-

tecture. The execution efficiency of QueKit server-client architecture is somewhere between that of QueKit server architecture and QueKit client architecture. The position depends on the distribution of the server architecture portion and the client architecture portion applied in a model. From the declarative model Fig. 5.11 (QueKit server-client architecture for the CPU_Disk system), 3/5 events for a life cycle of a Job belong to the client architecture portion. However, 7/9 events for a life cycle of a Job belong to the client architecture portion from Fig. 5.17 (QueKit server-client architecture for the harbor system). This observation is consistent with the simulation results shown in the following graphs. The execution efficiency of the server-client architecture is much closer to that of the client architecture for the harbor system than that for the CPU_Disk system.

TABLE 1. Simulation Results for the CPU_Disk System

Simulation Time (unit = ms)	# of Events in the SimKit Model	# of Events in the QueKit Model (Server Arch.)	# of Events in the QueKit Model (Server-Client Arch.)	# of Events in the QueKit Model (Client Arch.)
200000	45936	45936	79298	93317
400000	92013	92013	158854	186907
600000	137755	137755	237906	279722
800000	183028	183028	316067	371688
1000000	228978	228978	395443	464974
1200000	276345	276345	477206	561212

TABLE 2. Simulation Results for the Harbor System

Simulation Time (unit = ms)	# of Events in the SimKit Model	# of Events in the QueKit Model (Server Arch.)	# of Events in the QueKit Model (Server-Client Arch.)	# of Events in the QueKit Model (Client Arch.)
200000	90114	90114	191490	214020
400000	179566	179566	381576	426468
600000	266971	266971	567311	634055
800000	355839	355839	756152	845114
1000000	443353	443353	942124	1052962
1200000	532351	532351	1131240	1264330

From the above two tables, the number of events in the QueKit models with the server architecture is the same as that in the corresponding SimKit models. Thus the overhead of

the model with QueKit server architecture is only caused by the following two factors: 1) making events transparent to the user in QueKit; and 2) statistics and trace collection in QueKit because there is still overhead for testing whether statistics and trace are needed or not even if they are not needed in a simulation at all. The overhead caused by these two factors are unavoidable for any QueKit model. It is at the average of 39% for the model of CPU_Disk system and at the average of 62% for the model of the harbor system shown in the tables below.

TABLE 3. Modeling and Simulation Results for the CPU_Disk System

Model Architecture	Lines of Source Code	Average Overhead (%)
SimKit Model	240	0
QueKit Model (Server Arch.)	147	39
QueKit Model (Server-Client Arch.)	131	155
QueKit Model (Client Arch.)	122	214

TABLE 4. Modeling and Simulation Results the Harbor System

Model Architecture	Lines of Source Code	Average Overhead (%)
SimKit Model	196	0
QueKit Model (Server Arch.)	114	62
QueKit Model (Server-Client Arch.)	97	327
QueKit Model (Client Arch.)	85	387

For the models with the client architecture or the server-client architecture in QueKit, the number of events are more than one and half times but less than three times of the number of events in the corresponding SimKit models. This is the third factor that causes the overhead for the client and server-client architectures in addition to the two factors mentioned above. For the CPU_Disk system, the overhead caused by this third factor is 116% on average for the server-client architecture and 175% on average for the client architecture. For the harbor system, this part of overhead is 265% on average for the server-client architecture and 325% on average for the client architecture. This overhead is expected to be minimized by optimization techniques in the future.

As mentioned in Chapter 2, the comparison of the modeling power between different approaches comes from the comparison of the comprehensibility and flexibility of the

models using these approaches. The more comprehensible and more flexible a model are, the greater the modeling power of the approach used in the model will be. Thus the modeling power of different approaches is discussed in terms of the model comprehensibility and flexibility below.

For modeling the CPU_Disk system, the simulation model with QueKit server architecture is more comprehensible than the corresponding SimKit model. This is because the simulation event objects are totally transparent in the QueKit model whereas they are so pervasive in the SimKit model that they block the sight of the modeler from clearly viewing the model. For the same reason, the simulation model with QueKit server architecture is more comprehensible than that with SimKit API for modeling the harbor system.

For the models built in QueKit, the model with QueKit client architecture is more comprehensible than the model with QueKit server architecture for modeling the CPU_Disk system. This is because the lifetime activities of a task are described in a single *schedule* function of TaskClient class in the model with QueKit client architecture instead of being scattered in two *schedule* functions of CPU and Disk classes.

It may not be so convincing from modeling the CPU_Disk system that a model with QueKit client architecture is more comprehensible than that with QueKit server architecture because both architectures are suitable for modeling this system. However, the difference of comprehensibility between different QueKit architectures are more clear in modeling the harbor system. This is because the harbor system is suitable for applying the client architecture or the server-client architecture since multiple resources are involved in a single activity in the system. The model with QueKit client architecture Fig. 5.15 or QueKit server-client architecture Fig. 5.17 is much more comprehensible than the model with QueKit server architecture Fig. 5.13. This is because the activities of a boat can be clearly described in a chronological order in the client architecture, and this is very close to the scenario in the real system.

It is also mentioned in Chapter 2 that the comparison of the source code lines in different models using different approaches to modeling the same problem can help the comparison of the comprehensibility of these approaches for that problem. The less the source code lines are in a model, the more comprehensible the approach used in the model will be. In TABLE 3 and TABLE 4, only the actual lines of effective source code are counted, i.e., comment lines, blank lines, and continued lines from the previous line in the source file are not included over there. The information about the source code lines can be seen in the Appendix C (QueKit server-client architecture for the CPU_Disk system) and Appendix D (QueKit server-client architecture for the harbor system) at the end of the thesis.

Therefore, both TABLE 3 and TABLE 4 shows that models built with QueKit client architecture have the greatest comprehensibility and models built with SimKit have the least comprehensibility. Models built with QueKit server-client architecture are less comprehensible than those built with QueKit client architecture whereas they are more comprehensible than those built with QueKit server architecture.

Furthermore, these two tables also show that there are trade-offs between the comprehensibility and the execution efficiency for a specific model. The more comprehensible a model is, the worse the model execution efficiency will be, and vice versa. This is actually an advantage of QueKit. A model with modest complexity can be built with any of the three architectures in QueKit. The server architecture is preferred for an experienced user when the execution efficiency is the major concern in a simulation. The model is mostly built with the server-client architecture. The user can adjust the balance between the model execution efficiency and model comprehensibility by adjusting the distribution of the portion for applying the server architecture and the client architecture in the model.

From the aspect of flexibility, QueKit server architecture has the same flexibility as SimKit for modeling various queueing systems. This is because it uses a simple scheme to make events transparent from the application programmer while still preserving other functionality of SimKit so that it can provide an OO event-driven modeling framework.

Therefore, QueKit has a greater modeling power than SimKit in the simulation of queueing systems. However, QueKit client architecture is not as flexible as QueKit server architecture because sub-classes cannot be derived from the Resource class. Although many Resource classes can be provided within QueKit, there will always be situations when a new Resource class type is needed. Thus the client architecture can not be flexible enough to cover all of resource management paradigms in queueing systems.

5.4 Summary

QueKit has greater modeling power for queueing systems than SimKit. This is because it provides an OO event-driven modeling framework rather than the LP modeling framework in SimKit. That is, the modeling classes provided by QueKit are at the higher level and more specific to modeling queueing systems, and thus more close to the real world objects. This results in the model built in QueKit more comprehensible. Moreover, the simple scheme to make events transparent from the application programmer enables QueKit to preserve all the functionality of SimKit except the direct manipulation of events. It results in QueKit having the same flexibility as SimKit to cover a wide range of scenarios in the simulation of various queueing systems. However, the model execution efficiency in QueKit is not as good as that of SimKit duo to this event hiding scheme.

Among the three modeling architectures in QueKit, a model built with the client architecture will be more comprehensible than the one built with the server architecture. However, the server architecture is more flexible than the client architecture in the modeling a wide range of scenarios in a queueing system. Moreover, the execution efficiency of the server architecture is better than that of the client architecture. Thus the server architecture is preferred for an experienced user when the execution efficiency is the major concern in a simulation. The client architecture is preferred for a less experienced user when the model comprehensibility is more important in a simulation. Nevertheless, a user has to come to the server architecture if the client architecture is not flexible enough to model a specific queueing scenario in an application.

The OO event-driven modeling framework provided in QueKit enables the server architecture and client architecture to be applied seamlessly in a single simulation model. This results in the server-client architecture. The execution efficiency of the server-client architecture is somewhere between the above two architectures according to the distribution of the server architecture portion and the client architecture portion applied in a model. Therefore, there are trade-offs between the model comprehensibility and the execution efficiency when choosing an architecture for queueing system simulation in QueKit. The greater the model comprehensibility is, the less the model execution efficiency will be, and vice versa.

The overall execution overhead of QueKit compared to SimKit is caused by three factors. Making events transparent to the user is the first factor that causes overhead in QueKit. The second factor is the generality in the statistics and trace collection in QueKit because there is still overhead for testing whether statistics and trace collection are needed even if they are not needed as all. The last factor is that there are more simulation events in a model with QueKit client architecture, or server-client architecture, than that in the corresponding model with QueKit server architecture.

The first two factors cause the overhead in a model with QueKit server architecture. All three factors cause the overhead in a model with QueKit client architecture or server-client architecture.

Chapter 6

Conclusion

Many discrete event simulation (DES) studies involve the modeling of a real-world queueing system. There are two major problems in the simulation of queueing systems: it is very difficult to come up with a simulation model when the system to be modeled becomes large and complex; and a simulation of such complex systems can be very computationally intensive.

This chapter summarizes the results and the contribution of the thesis in the course of solving these two problems, and concludes with topics for future work.

6.1 Summary and Conclusions

The above two problems have been addressed by the general discrete event simulation (DES) community and parallel discrete event simulation (PDES) community for decades. Numerous simulation packages have been developed in order to provide the modeling frameworks that facilitate the construction of complex models. A lot of techniques on how to speed up simulation execution on parallel and distributed machines have also been explored. Furthermore, object-oriented (OO) techniques have also been widely used in these two communities in order to facilitate the simulation development process. The focus and objectives are different between the two communities, and little work has been done on the intersection of these problems. This is the focus of the thesis.

The above problems are discussed from two aspects in this thesis: modeling effectiveness and execution efficiency. The modeling effectiveness issue is addressed from three dimensions: modeling framework, model architecture, and object orientation. Since the chosen framework and model architecture in the modeling process will largely affect the execution efficiency in the simulation, the modeling effectiveness issue is discussed with the execution efficiency issue together throughout the entire thesis.

An application programmer usually has to follow a modeling framework to construct a model. The modeling framework can provide conceptual guidance for the modeler in the simulation process. The current modeling frameworks for queueing systems are event-driven, the process view, and the logical process (LP) view. The process view can be split into two types: process interaction and process description (or transaction flow).

The modeling power of a modeling framework is concerned with the comprehensibility of the resulting model and the flexibility to cover a wide range of modeling scenarios. The following two conclusions can be drawn from the analysis of the above three frameworks.

(1) The modeling power of the process view is generally greater than that of the event-driven view whereas the execution efficiency of the event-driven view is better than that of the process view.

(2) The LP view is less efficient in its access to variables that are shared by more than one LP. This follows the constraint that state information of an LP cannot be accessed from outside of that LP except through exchanging (event) messages.

(3) The LP view offers part of the modeling power of the process view. This is accomplished by making all state information private within the LPs of a model, and by associating all events with some LPs. This is in contrast to the event-driven view where all events relate only to the overall system and all state information is global.

From the viewpoint of model architecture, there are two approaches to modeling queueing systems: server architecture and client architecture. The server architecture emphasizes the modeling of (static) entities (modeled as servers) that provide services for others while the client architecture emphasizes the modeling of (dynamic) entities (modeled as tokens) that need services from others in a queueing system. The following are the two conclusions from the study of model architectures.

(1) The server architecture focuses on how a server manages the resource allocation/ deallocation to/from tokens, and how the server controls the activities of these tokens, and how it routes the tokens to other servers after service. A server is active in the server architecture in the sense that it functions as a dominant controller which controls everything within its area. A token is passive in the sense that it is passively processed by servers. This kind of passive token vs. active server architecture is suitable for modeling a queueing systems such as computer systems and communications networks.

(2) The client architecture focuses on when a token requests a resource, how it conducts an activity, and when it returns the resource and moves on to another activity. That is, the client architecture emphasizes modeling the lifetime activities of a token moving through the system in a chronological order. A token is active in the client architecture in the sense that it can actively request/return resources and controls its own activities. A server is passive in the sense that it only functions as a resource controller which manages the resource allocation/deallocation, but it has no control over the activity of any token. This kind of active token vs. passive server architecture is suitable for a queueing system in which a token needs more than one resource that are independently managed.

The use of OO techniques makes a simulation package easy to use and easy to maintain because it supports the close correspondence between a model and real-world system. There are three levels for the OO concepts applicable to the development of simulation packages: abstraction, design, and implementation. There are also three approaches to developing an OO simulation package: data-driven simulator, language extension, and library-based approaches. The data-driven approach is usually successful in applying OO techniques at the abstraction level, but not at the design and implementaion levels. The language extension and library-based approaches have the potential to apply OO techniques at all three levels.

For the packages related to queueing system simulation in the literature, data-driven simulators usually apply OO techniques at the abstraction level. Whereas, the language

extension [illegible]
impleme[illegible]
three lev[illegible]
switching [illegible]

Simk[illegible]
very sim[illegible] efficien[illegible]
problems [illegible]
construct[illegible] simulation[illegible]
not at the application lev[illegible]
Kit was [illegible] as [illegible]
modeling and simulatio[illegible]

An [illegible] simulati[illegible]
defined in this thesis. Qu[illegible]
ment for queueing system[illegible]
good execution efficienc[illegible]
top of SimKit. The [illegible]
that OO techniques can [illegible]
modelin[illegible] framework pr[illegible]
services [illegible] others [illegible]
that prov[illegible] these servic[illegible]
impleme[illegible] of QueK[illegible]
QueKit o[illegible] of SimKit [illegible]
ronment [illegible] op[illegible]

Thre[illegible]
tecture, c[illegible] architec[illegible]
by the modeling and sim[illegible]

extension or the library-based packages usually apply OO techniques at the design and implementation levels. Some of them, such as PROSIT, may apply OO techniques at all three levels, but they suffer from inefficiency problems because they use a costly context switching mechanism in order to provide the process view.

SimKit is a library-based simulation package built with an OO language. It provides a very simple and efficient LP event-driven view for modeling and simulating various DES problems both in sequential execution and in parallel execution. However, the modeling constructs (or simulation primitives) provided in SimKit are only at the simulation level, not at the application level when they are used for modeling a queueing system. Thus SimKit was chosen as the environment for developing a package at a higher level for OO modeling and simulation of queueing systems.

An OO simulation package for queueing system simulation called QueKit has been defined in this thesis. QueKit aims to provide an OO design and implementation environment for queueing system simulation that facilitates the modeling process while retaining good execution efficiency in both sequential and parallel executions. QueKit is built on top of SimKit. The library-based approach is also used for the development of QueKit so that OO techniques can be applied at abstraction, design, and implementation levels. The modeling framework provided in QueKit is an OO event-driven view. Entities that need services from others in a queueing system are modeled as Tokens or Jobs while entities that provide these services are modeled as Servers or Resources. Although the current implementation of QueKit only supports sequential execution, the strategy of developing QueKit on top of SimKit enables any QueKit model to be executed within a parallel environment including optimistic and conservative approaches.

Three architectures are provided by QueKit for the model construction: server architecture, client architecture, server-client architecture. The server architecture is supported by the modeling and simulation constructs in the base QueKit layer. Both client architec-

ture and server-client architecture are supported by the modeling and simulation constructs in the extended QueKit layer.

A model with QueKit server architecture is mainly composed of Tokens and Servers. Tokens flow through a network of Servers to obtain service and finally leave the system. A Server has two Sets, one is the service Set that holds the Tokens being serviced, and another is the wait Set that holds the Tokens waiting for services. A Server decides the strategy for servicing Tokens through its *schedule (Token)* method. This kind of passive Token vs. active Server architecture is suitable for modeling those systems such as communications networks which can be viewed that servers controls everything.

A model with QueKit client architecture is mainly composed of Jobs, Clients, and Resources. A Jobs models an active entity that can request services from others. A Resource models a passive entity that provides those services. The entire lifetime activities of a Job is described in a Client *schedule* method. A Job decides when and where to request the service from Resources, and how long to keep the resource(s). A Resource is passive in the sense that it only responds to the requests from Jobs for resource allocation/ deallocation, and it has no control on the Jobs' activities like a Server.

The server-client architecture allows both QueKit server architecture and QueKit client architecture to be applied seamlessly in a single simulation model. It is unique in QueKit. It has some advantages that will be presented below.

For modeling the same queueing system, it is easier to build a model in QueKit and the resulting model will be more comprehensible that any model built in SimKit. This is because QueKit provides an OO event-driven modeling framework rather than the LP framework provided in SimKit. Moreover, QueKit is the same flexible as SimKit in modeling queueing systems because it preserves all the functionality of SimKit except the direct manipulation of event objects. Therefore, QueKit has a greater modeling power than SimKit in the queueing system simulation. However, the model execution efficiency

in QueKit is not as good as that of SimKit due to the hiding of event objects from the application programmer.

Among the three QueKit modeling architectures, the modeling power of the client architecture is greater than the server architecture in the aspect of comprehensibility, but less than the server architecture in the aspect of flexibility. The modeling power of the server-client architecture is between these two extremes. The execution efficiency of the server architecture is better than that of the client architecture. The execution efficiency of the server-client architecture is somewhere between these two extremes according to the distribution of the server architecture portion and the client architecture portion applied in a model.

Therefore, there are trade-offs between the modeling power and the execution efficiency when choosing an architecture for queueing system simulation in QueKit. For modeling a complex system, the more comprehensible a model is, the less the model execution efficiency will be, and vice versa. Thus the server architecture is preferred for an experienced user when the execution efficiency is the major concern in a simulation. The client architecture is preferred for a less experienced user when the model comprehensibility is more important in a simulation unless the client architecture is not flexible enough to cover the queueing scenario in the simulation.

The following three factors are responsible for the overall execution overhead of QueKit comparing to SimKit. The first one is to make events transparent to the user in QueKit. The second one is the overhead for statistics and trace collection because there is still overhead for testing whether those collections are needed or not even if they are not used. The last one is the extra number of events in a model with QueKit client architecture or server-client architecture compared to the corresponding model with QueKit server architecture.

The first two factors cause the overhead in a model with QueKit server architecture. All of the three factors are responsible for the overhead in a model with QueKit client

architecture or server-client architecture. The overhead is expected to be minimized in the future.

6.2 Thesis Contribution

The main contribution of this thesis is the design and implementation of an OO event-driven framework for queueing system simulation so that both the server architecture and the client architecture can be applied seamlessly in a single model. This approach of server-client architecture allows the user to adjust the balance between modeling effectiveness and execution efficiency in sequential execution. This is accomplished by adjusting the distribution between the server architecture portion and the client architecture portion applied in a model. Moreover, the flexibility of this server-client architecture is expected to permit more natural parallelism in a queueing system to be exploited in a simulation when running in parallel. This may result in a model with the server-client architecture that has better modeling effectiveness, as well as, better execution efficiency.

6.3 Future Work

The weakness of QueKit server architecture is that a model built with it is less comprehensible than the one built with the client architecture. More constructs such as scheduler that cover various queueing operations (round robin, priority, etc.) are expected to be developed in order to enhance model comprehensibility.

The QueKit client architecture is weak in the aspect of flexibility to cover a wide range of queueing scenarios. More constructs are needed to be developed in order to support various resources.

The modeling and simulation experiments in QueKit presented in Chapter 5 show promising results. These experiments were only conducted in sequential execution as the current Java version of QueKit can only support sequential execution. Therefore, a C++ version of QueKit is expected to be developed in the future in order to run simulations in parallel. The modeling power for the three architectures in parallel execution will remain

the same as in sequential execution. This is because the parallel simulation models are almost identical to the corresponding sequential models except for the addition of state saving calls for optimistic parallel execution. The interesting experiments will be about how to adjust the model architecture between QueKit server architecture and QueKit client architecture in order to get better modeling effectiveness and execution efficiency.

Bibliography

1. Balci, O., Nance, R. E., Derrick, E. J., Page, E. H., Bishop, J. L., *Model Generation Issues in a Simulation Support Environment*, Proceedings of WSC, 1990.

2. Ball, P., Love, D., *The Key to Object-Oriented Simulation: Separating the User and the Developer*, Proceedings of WSC, 1995.

3. Banks, J., Carson, J. S., Nelson, B. L., *Discrete-Event System Simulation*, Prentice-Hall International Series, 1996.

4. Beilner, H., Sczittnick, M., Wysocki, Ch., *Look at HIT*, Demo Version, Document Version 1.0.00, technical report, August 1995.

5. Birtwistle, G. M., *Discrete Event Modeling on Simula*, The Macmillan Press Ltd., 1979.

6. Carson, J. S., *Modeling*, Proceedings of WSC, 1992.

7. Carson, J. S., *Modeling and Simulation World Views*, Proceedings of WSC, 1993.

8. Coad, P., Yourdon, E., *Object-Oriented Design*, Object International, Inc., 1991.

9. Davies, R. M., O'Keefe, R. M., *Simulation Modeling with Pascal*. Prentice Hall International (UK) Ltd., 1989.

10. Ferscha, A., *Parallel and Distributed Simulation of Discrete Event Systems*, McGraw-Hill, 1995.

11. Ferrante, P., Mussi, P., Siegel, G., Mallet, L., *Object Oriented Simulation: Highlights on The PROSIT Parallel Discrete Event Simulator*, in INRIA Research Report 2235, April 1994.

12.Fishwick, P. A., *SimPack: Getting Started with Simulation Programming in C and C++*, Proceedings of WSC, 1992.

13.Fishwick, P. A., *Simulation Model Design and Execution*, Prentice-Hall, Inc. 1995

14.Franta, W. R., *The Process View of Simulation*, Elsevier North-Holland, Inc., 1977.

15.Fujimoto, R. M., *Parallel Discrete Event Simulation*, Communications of ACM, October 1990.

16.Gomes, F., Franks, S., Unger, B., Xiao, Z., Cleary, J., Covington, A., *Simkit: A High Performance Logical Process Simulation Class Library in C++*, Proceedings of WSC, 1995.

17.Hares, J. S., Smart, J. D., *Object Orientation*, John Wiley & Sons Ltd, 1994.

18.Joines, J. A., Roberts, S. D., *Design of Object-Oriented Simulations in C++*, Proceedings of WSC, 1995.

19.Kheir, N. A., *System Modeling and Computer Simulation*, Marcel Dekker, Inc., 1988.

20.Kleinrock, L., *Queueing Systems*, Volume I: Theory, John Wiley & Sons, Inc. 1975.

21.Law, A. M., McComas, M. G., *Simulation of Communications Networks*, Proceedings of WSC, 1995.

22.Lommow, G., Baezner, D., *A Tutorial Introduction to Object-Oriented Simulation and Sim++*^{TM}, Proceedings of WSC, 1990.

23.MacDougall, M. H., *Simulation Computer System*, Massachusetts Institute of Technology, 1987.

24.Mallet, L., Mussi, P., *Object Oriented Parallel Discrete Event Simulation: The PROSIT Approach*, in Modeling and Simulation, Lyon, June 1993. Also in INRIA Research Report 2232, April 1994.

25.Molloy, M. K., *Fundamentals of Performance Modeling*, Macmillan Publishing Company, a division of Macmillan, Inc. 1989.

26.Mussi, P., Siegel, G. *Sequential Simulation in Prosit: Programming Model and Implementation*, in INRIA Research Report 2713, November 1995.

27.Page, E. H., Nance, R. E., *Parallel Discrete Event Simulation: a Modeling Mthodological Perspective*, PADS 1994.

28.Pidd, M., *Object Orientation & Three Phase Simulation*, Proceedings of WSC, 1992.

29.Pooch, U. W., Wall, J. A., *Discrete Event Simulation: A Practical Approach*, CRC Press, 1993.

30.Schriber, T., *Simulation Using GPSS*, John Wiley, 1974.

31.Schwetman, H. D., *Introduction to Process-Oriented Simulation and CSim*, Proceedings of WSC, 1990.

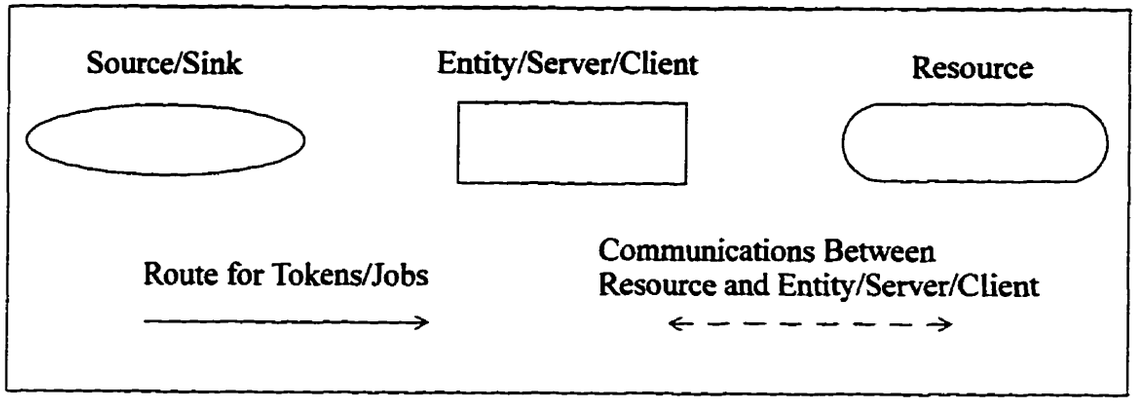
32.Stroustrup, B., *The C++ Programming Language*, (Second Edition), Addison-Wesly Publishing Company, 1991.

33.Unger, B. W., Gomes, F., Xiao, Z., Gburzynski, P., One-Tesfaye, T., Ramaswamy, S., Williamson, C., Covington, A., *A High Fidelity ATM Traffic and Network Simulator*, Proceedings of WSC, 1995.

34.Wang, P. S., *C++ with Object-Oriented Programming*, PWS Publishing Company, 1994.

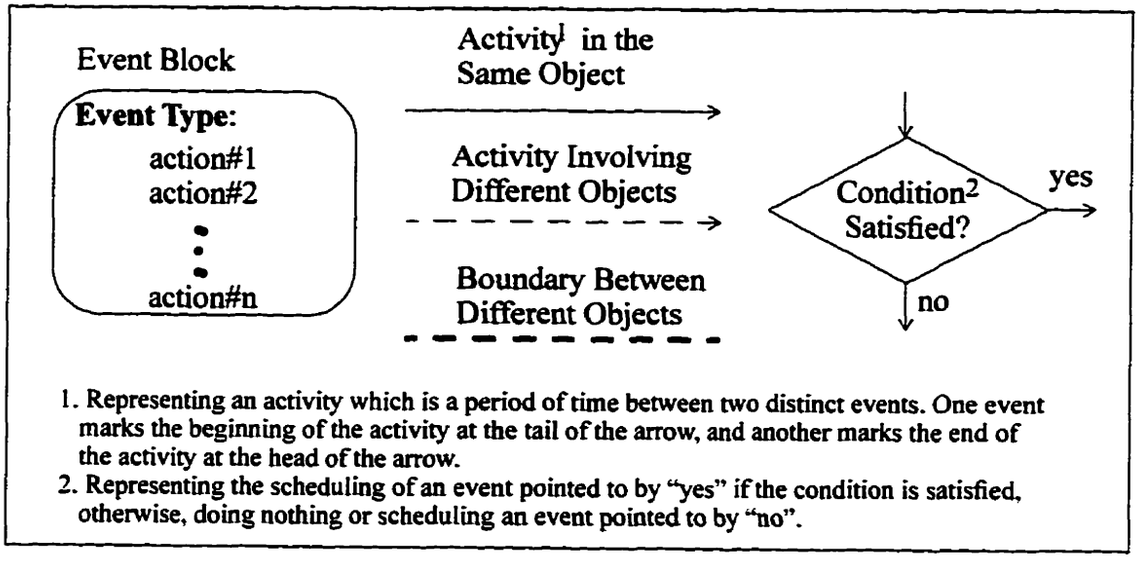
Appendix A

Functional Model Notation



Appendix B

Declarative Model Notation



1. Representing an activity which is a period of time between two distinct events. One event marks the beginning of the activity at the tail of the arrow, and another marks the end of the activity at the head of the arrow.
2. Representing the scheduling of an event pointed to by "yes" if the condition is satisfied, otherwise, doing nothing or scheduling an event pointed to by "no".

Appendix C

Model for the CPU_Disk System

(QueKit Server-Client Architecture)

```
/****************** Task.java ************************/
1. import Utility.*;
2. import QueKit.*;

3. class Task extends Job {
4.   static EXP CPUTimeHigh = new EXP (5);        // mean = 5 ms.
5.   static EXP CPUTimeLow = new EXP (10);        // mean = 10 ms.
6.   static EXP DiskTime = new EXP (50);          // mean = 50 ms.

7.   static int numTours;              // total # of tours
8.   static double totalTourTime;      // total tour time of all tasks

9.   private int stageSeq,
10.              diskIndex;            // record the disk index
11.  private double serviceStart,      // start time for a service
12.                 demand = -1.0,     // the remain CPU service time if > 0
13.                  myPriority,       // task priority
14.                  tourTime;         // record a tour start time
15.  public Task (String type, double prio) {
16.      super(type);
17.      myPriority = prio;
18.      setPriority(prio);
     }
19.  public int seq() { return stageSeq; }
20.  public int diskIndex () { return diskIndex; }
21.  public void seqInc() { stageSeq++; }
22.  public void seqDec(int n) { stageSeq -= n; }
23.  public void resetSeq() { stageSeq = 0; }
24.  public double tourTime () { return tourTime; }
25.  public void setDiskIndex (int i) { diskIndex = ((i < 0) || (i > 3)) ? diskIndex : i; }

26.  public double CPUServiceDemand() {
27.      if (demand < 0) { // not a preempted task
28.        if (priority() > 0)
29.              demand = CPUTimeHigh.nextEXPDouble();
30.        else
31.              demand = CPUTimeLow.nextEXPDouble();
         }
32.      return demand;
     }
```

```
33. public double DiskServiceDemand() { return DiskTime.nextEXPDouble(); }
34. public double startTime() { return serviceStart; }
35. public void setStartTime (double t) { serviceStart = t; }
36. public void resetCPUServiceDemand () { demand = -1.0; }
37. public void setCPUServiceDemand (double t) { demand = t; }
38. public void resetPriority() { setPriority (myPriority); }
39. public void setTourTime (double t) { tourTime = t; }
    } // end of class Task

    /***************** TaskClient.java  ********************/
40. import Utility.*;
41. import QueKit.*;

42. class TaskClient extends Client {
43.  static Uniform diskIndex = new Uniform (0, 3);
44.  public TaskClient () { super (); }

45.  public void schedule (Token token) {
46.       Task task = (Task)token;
47.       if (task.currentSet() == null)
48.            hostSet().put (task);
49.       task.seqInc();
50.       switch (task.seq()) {
51.         case 2:
52.            task.setDiskIndex (diskIndex.nextUniformInt());
53.            task.requestRes (SimControl.disk [task.diskIndex()]);
54.            break;
55.         case 3:
56.            task.keep (task.DiskServiceDemand());
57.            break;
58.         case 4:
59.            task.freeRes (SimControl.disk [task.diskIndex()]);
60.            Task.numTours++;
61.            Task.totalTourTime += QueSimulation.currTime() - task.tourTime ();
62.            task.setTourTime (QueSimulation.currTime());
63.            task.resetSeq ();
64.            task.resetPriority();
65.            task.send (SimControl.cpu, 0.0);
       }
     }
    } // end of TaskClient


    /************ CPU.java ***************/
66. import QueKit.*;

67. class CPU extends Server {
68.  static int n0, n1;       // # of n0 tasks, n1 task

69.  public CPU (int v1, int v2) { super (); n0 = v1; n1 = v2; }
```

```
70.  public void initialize() {
71.       Task task = null;
         // inilialize all n0 and n1 tasks
72.       for (int i=0; i < n0; i++) {
73.                 task = new Task ("n0 task", 0);
74.                 task.send (this, 0.0);
          }
75.       for (int i=0; i < n1; i++) {
76.                 task = new Task ("n1 task", 1);
77.                 task.send (this, 0.0);
          }
     }

78.  public void schedule (Token token) {
79.       Task task, newTask = (Task)token;

80.       switch (newTask.seq()) {
81.        case 0:                                  // a task requests the CPU
82.                 newTask.seqInc();
83.                 if (serveSet().full()) {
84.                   task = (Task)((Queue)serveSet()).last();
85.                   if (newTask.priority() > task.priority()) { // preempt the CPU

                            // cancel task's current activity
86.                         task.cancel ();
                            // increase task's priority and put it back to the waitSet
87.                         task.setPriority (0.5);
88.                         task.setCPUServiceDemand (task.CPUServiceDemand()
                                  - QueSimulation.currTime() + task.startTime());
89.                         task.move (waitSet());

                            // token preempt the CPU and get served
90.                         newTask.move (serveSet());
91.                         newTask.keep (newTask.CPUServiceDemand());
92.                   } else {
93.                         newTask.move (waitSet());
                      }
94.                 } else {  // token gets the service if serveSet() not full
95.                         newTask.move (serveSet());
96.                         newTask.setStartTime (QueSimulation.currTime());
97.                         newTask.keep (newTask.CPUServiceDemand());
                    }
98.                 break;
99.        case 1:                          // a task leaves the CPU after service
100.                newTask.resetCPUServiceDemand();
101.                newTask.setPriority (0);
102.                newTask.send (SimControl.taskClient, 0.0);

103.                task = (Task)((Queue)waitSet()).first();
104.                if (task != null) {
105.                        task.move (serveSet());
```

```
106.                    task.setStartTime (QueSimulation.currTime());
107.                    task.keep (task.CPUServiceDemand());
                    }
            } // end of switch
        } // end of schedule

108.  public void terminate () {
109.      System.out.println ("Task n0 = "+n0+" Task n1 = "+n1+
                    " Total number of task tours = "+Task.numTours+"  Ave. tour time = "+
                    Task.totalTourTime/Task.numTours);
      }
    } // end of class CPU

    /************** SimControl.java ******************/
110.import SimKit.*;
111.import QueKit.*;

112.class SimControl extends QueSimulation {
113.    double fEndTime;
114.    static Resource [] disk = new Resource [4];
115.    static CPU cpu;
116.    static TaskClient taskClient;

117.    public static void main (String argv[]) {
118.      Arguments args = new Arguments(argv);
119.      new SimControl (args).run();
        }

120.    public SimControl (Arguments args) {
121.      super(args);
122.      String val = args.retrieve("EndTime");
123.      if (val != null) { fEndTime = (Double.valueOf(val)).doubleValue(); }
        }

124.    public void initialize() {
125.      QueSimulation.dbgPrint(" CPU_Disk System Simulation initialize()");
126.      QueSimulation.setEndTime(fEndTime);
127.      cpu = new CPU (6, 2);
128.      taskClient = new TaskClient ();
129.      for (int i = 0; i < 4; i++)
130.              disk[i] = new Resource (1);
      }
131.    public void terminate() { QueSimulation.dbgPrint ("Similation terminated."); }
    } // end of SimControl
```

Appendix D

Model for the Harbor System

(QueKit Server-Client Architecture)

```
/************Boat.java (file name)*******************/

1. import QueKit.*;

2. public class Boat extends Job {
3.   private int seqNum;// stage sequence #
4.   private double arriveT;// time when the boat arrives at the harbor
5.   public Boat () {
6.        super ("Boat");
7.        arriveT = QueSimulation.currTime();
     }
8.   public final int seq() { return seqNum; }
9.   public final void incSeq () { seqNum++; }
10.  public double arriveTime() { return arriveT; }
    } // end of Boat

    /***************Inspector.java (file name)***************/

11. import QueKit.*;

12. class Inspector extends Server {

13.  public void schedule (Token token) {
14.       Boat boat = (Boat)token;
15.       boat.incSeq();

16.       switch (boat.seq()) {
17.        case 1:          // stage #1 -- a boat arrives for inspection
18.          if (serveSet().full()) {
19.            boat.move (waitSet());
20.          } else {
21.            boat.move (serveSet());
22.            boat.keep (3.0);
              }
23.          break;
24.        case 2:          // stage #2 -- a boat finishes the inspection
25.          boat.send (SimControl.boatClient, 0.0);
26.          boat = (Boat)((Queue)waitSet()).first();
27.          if (boat != null) {
28.            boat.move (serveSet());
```

```
29.             boat.keep (3.0);
                }
            }
        }
    } // end of Inspector

    /*****************BoatClient.java (file name)************/

30. import QueKit.*;

31. class BoatClient extends Client {
32.   static Resource jetty, tug;
33.   static BoatSink sink;
34.   public BoatClient () { super(); }

35.   public void initialize () {
36.       jetty = new Resource (2);
37.       tug = new Resource (3);
      }
38.   public void schedule (Token token) {
39.       Boat boat = (Boat)token;
40.       if (boat.currentSet() == null)
41.           hostSet ().put (boat);
42.       boat.incSeq();
43.       switch (boat.seq()) {
44.         case 3:
45.           boat.requestRes (jetty);
46.           break;
47.         case 4:
48.           boat.requestRes (tug, 2);
49.           break;
50.         case 5:
51.           boat.keep (2.0);
52.           break;
53.         case 6:
54.           boat.freeRes (tug, 2);
55.           boat.keep (14.0);
56.           break;
57.         case 7:
58.           boat.requestRes (tug);
59.           break;
60.         case 8:
61.           boat.keep (2.0);
62.           break;
63.         case 9:
64.           boat.freeRes (jetty);
65.           boat.freeRes (tug);
66.           boat.send (sink, 0.0);
        }
    } // end of schedule
  } // end of BoatClient
```

```
/***************BoatSink.java (file name)***************/

67. import QueKit.*;

68. class BoatSink extends Sink {
69.   private double totalTime;
70.   public BoatSink () { super (); }

71.   public void schedule (Token token) {
72.       Boat boat = (Boat)token;
73.       totalTime += QueSimulation.currTime() - boat.arriveTime();
74.       addTotalNumTokens (1);
      }
    } // end of Sink

/*******************SimControl.java (file name) ***************/

75. import SimKit.*;
76. import QueKit.*;

77. class SimControl extends QueSimulation {
78.   static Inspector inspector;
79.   static BoatClient boatClient;
80.   static Source source;
81.   double fEndTime;

82.   public static void main (String argv[]) {
83.       Arguments args = new Arguments(argv);
84.       new SimControl (args).run();
      }
85.   public SimControl (Arguments args) {
86.       super(args);
87.       String val = args.retrieve("EndTime");
88.       if (val != null) { fEndTime = (Double.valueOf(val)).doubleValue(); }
      }

89.   public void initialize() {
90.       QueSimulation.dbgPrint(" Harbor System Simulation initialize()");
91.       QueSimulation.setEndTime(fEndTime);

92.       boatClient = new BoatClient ();
93.       inspector = new Inspector ();
94.       source = new Source ((new Boat ()).getClass(), "EXP", 18.0);
95.       source.setDestination (inspector);
96.       BoatClient.sink = new BoatSink ();
      }

97.   public void terminate() { QueSimulation.dbgPrint (" Simulation terminated"); }
    } // end of SimControl
```