AN OBJECT - ORIENTED SIMULATION MODEL FOR AIRPORT TRAFFIC CONTROL

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

by
İsmail Yıldırım
January 1995

AN OBJECT-ORIENTED SIMULATION MODEL FOR AIRPORT TRAFFIC CONTROL

A THESIS

SUBMITTED TO THE DEPARTMENT OF INDUSTRIAL

ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BİLKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

by

İnanç Yıldırım

January 1995

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

Prof. M. Akif Eyler (Advisor)

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

Assoc. Prof. Varol Akman

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

Asst. Prof. İhsan Sabuncuoğlu

Approved for the Institute of Engineering and Science:

Prof. Mehmet Baray
Director of the Institute

ABSTRACT

AN OBJECT-ORIENTED SIMULATION MODEL FOR AIRPORT TRAFFIC CONTROL

İnanç Yıldırım
M.S. in Industrial Engineering
Advisor: Prof. M. Akif Eyler
January 1995

In recent years airport congestion and delay problems have received a great deal of attention due to the rapid growth of air transportation services. Of all the elements contributing to the air terminal congestion, the Air Traffic Control problem is the best understood. In this thesis we present an object-oriented model for simulating air traffic flow at an airport. The application of object-oriented design to the simulation model construction process is identified as a need, particularly for modeling large and complex systems. Object-oriented paradigm has already demonstrated that it can help to manage the growing complexity and increasing costs of the software development. The model we present here, has been implemented on a personal computer by illustrating a major metropolitan airport.

Keywords: Object-Oriented Simulation, Air Traffic Control, Object-Oriented Design, Discrete Event Simulation, Airport Systems.

ÖZET

HAVAALANI TRAFİK KONTROLÜ İÇİN NESNEYE YÖNELİK BENZEŞİM MODELİ

İnanç Yıldırım
Endüstri Mühendisliği, Yüksek Lisans
Danışman: Prof. Dr. M. Akif Eyler
Ocak 1995

Hava taşımacılığı hizmetlerinin son yıllardaki artışı dolayısıyla, havaalanı tıkanıklığı ve gecikme sorunları daha da önemli hale gelmiştir. Hava terminali tıkanıklığına neden olan öğeler arasında Hava Trafiği Kontrolü sorunu en anlaşılabilir olanıdır. Bu tezde, bir havaalanında hava trafiği akışı benzeşimi için nesneye-yönelik bir model sunuyoruz. Nesneye-yönelik tasarımın, benzeşim modeli kurma işlemine uygulanması, özellikle büyük ve karmaşık sistemlerin modellenmesi için, bir gereksinim olarak belirlenmiştir. Buna ek olarak, nesneye-yönelik paradigma yazılım geliştirmenin büyüyen karmaşıklığı ve artan giderini düzenleyebilir olduğunu göstermiştir. Burada sunduğumuz model, kişisel bilgisayar üzerinde büyük bir metropol havaalanı benzetimi ile oluşmuştur.

Anahtar Sözcükler: Nesneye-Yönelik Benzetim, Hava Trafik Kontrolü, Nesneye-Yönelik Tasarım, Kesikli-Olay Benzetimi, Havaalanı Sistemleri.

ACKNOWLEDGMENTS

I would like to thank my advisor Prof. M. Akif Eyler who has provided motivating support during my M.S. study.

I would also like to thank Asst. Prof. İhsan Sabuncuoğlu and Assoc. Prof. Varol Akman for their valuable comments on this thesis.

Finally, I would like to address everybody who has in some way supported me, particularly Yusuf Sinan Hüsrevoğlu, B.S., Okan Yılmaz, M.Sc., and especially Mr. Kayıhan Kabadayıoğlu, M.Sc., Director General of Civil Aviation, for his encouraging technical and intellectual support.

I dedicate this thesis to those who sacrifice their lives for flying.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Air Traffic System

The demand for air transportation is growing rapidly. The growth is increasing faster than the improvements and modifications that are being made to keep the air traffic control (ATC) system properly responsive. Of all the elements contributing to the air terminal airspace congestion, the ATC problem is the best understood [22].

Each year, around 20 air-carrier crashes are detected in the world. Top causes of general aviation accidents are recorded as below [1]:

- loss of directional control during takeoff,
- loss of control in gusty-crosswind conditions,
- failure to maintain speed,
- attempted takeoff over-weight,
- malfunctioning components,
- runway misuse such as, landing long, short or hard.

cause	*accidents*	*percentage*
pilot	271	75 %
aircraft	40	11 %
weather	18	5 %
airport	15	4 %
maintenance	6	2 %
undetermined	13	3 %

Table 1.1. 1959-1989 air-carrier crash statistics, classified by the causes of accidents.

For the years between 1959-1989, a total of 363 jet air-carrier crashes were recorded, by the *International Civil Aviation Organization* (ICAO). Their categories are given in the table 1.1.

The implications deduced from these statistics show that:

- 28 % of all accidents occured in the first 5 minutes of flight, i.e. take-off and climbing.
- 41 % of all accidents occured in the last 10 minutes of flight, i.e. final approach and landing.

These results show the great danger in *aerodrome control areas*, and hence the importance of the Air Traffic Control (ATC) Systems, functioning in those areas.

Beside this danger, there is another important issue created by the heavy air traffic on the airports: The proportion of flights delayed by more than 15 minutes almost doubled between 1986 and 1989. The cost of these delays to airlines and the traveling community has been estimated by the researchers at $1.5 billion annually. The total loss due to delays, inefficient routings arising from poor route structure and military airspace restrictions, non-optimal flight profiles, low air traffic control system productivity, and other inefficiencies has been estimated at $5 billion.

LUFTHANSA, for example, reported 5,200 hours in holding patterns over Frankfurt, Münich, and Düsseldorf in 1987 with an increase 90 % from the preceding year. The resulting delays cost the airline more than $30 million for just these three airports alone.

Forecasts indicate that the number of passengers enplaned and deplaned will nearly double by the year 2000 and triple by 2010.

It is reported by the researchers that, sufficient capacity exists to accomodate likely growth to the year 2010, if it is efficiently organized. A commitment to timely implementation of existing airport development plans and adoption of procedural changes available in the next decade will postpone worsening airport congestion until 2010. In many cases, the cost of improvements is small with respect to continuing losses from inadequate capacity [2].

1.2 Air Congestion Problem

While present efforts to increase controller staffing, flow management, advanced radar installation and system harmonizations are essential to address the immediate problems of the air traffic control systems, the fundamental solution to the capacity deficiencies identified will be airspace rerouting and resectorization. For an airport system, most of the problems are caused by congested traffic. Congestion results in air crashes and huge delays; even those delays cause accidents. Therefore, the first aim of airport traffic controllers is to reduce the delays maintaining fast and safe flights.

An immediate solution to this aim can be achieved by reducing the controllers' workloads (e.g., resectorization of terminal airspace), and by reducing the spacing between flights which is enforced by the controllers. No simple description of the behaviour of the ATC system is available from which deductions could be drawn covering the effect of changes in the system parameters on the system performance.

In our thesis, we started with these points in mind. Then we began to

develop a simulation model to study the complex terminal ATC system, to discover congested air traffic delays in a terminal airspace, by illustrating with a typical airport scenario. Later, we decided to extend the model to be general enough to cover various airport system configurations. Several simulation studies were found in the literature; however, these studies, in general, have examined only specific subsystems of the ATC system.

1.3 Introduction to Simulation

Simulation is the process of designing a model of a real system and conducting the behaviour of the system and/or evaluating various strategies for the operation of the system. It is a powerful tool for the planning, design and control of systems [14].

Although simulation is very robust tool to analize systems, simulation programmers must be very careful about the effectiveness of their effort. Modeling is often difficult and time-consuming, and hence the modeling effort is usually much for a specific model.

Object-oriented paradigm in simulation responds to many of the questions above. It makes model building easier; objects are the natural way to describe many of the entities in a simulation study. It promotes *reusability*, by its *inheritance* principle, hence the created models can be extended to apply various similar systems (*extendibility*). By these two features, this paradigm increases the speed of software production. Encapsulating the code and data into an object type declaration in object-oriented design, allows *modularity* with the advantages of *readability* and *maintainability*. After having designed object types of the simulated system, the model can be constructed by combining these objects properly. Obviously, the objects can be combined in alternating ways to simulate similar, but different models.

The purpose of our thesis is to develop an extendible and generalizable "air traffic control system" model, by using an object-oriented approach.

1.4 Outline of the Thesis

The outline of the thesis is as follows:

- Chapter 2 introduces the object-oriented programming and describes its terminology. The issue of choice of simulation software and object-oriented simulation software are analyzed in this chapter. It shows the necessity and appropriateness of the object-oriented approach to simulation.
- Chapter 3 describes the Air Traffic Control System that we model. It introduces the basic air traffic procedures enforced in the airports, and analyzes the İstanbul Atatürk Airport System as a particular case. The chapter gives the design and the modeling approach for that airport.
- Chapter 4 deals with the design and implementation of the airport components that we used in our study. It gives the declarations of the object data types and their hierarchy. It describes the execution and outputs of the program. Then the verification and the validation issues of the simulation program are considered.
- Chapter 5 gives the results and the interpretation of these results. The simulation program was run with different parameters and input data files, to analyze the sensitivity of the model to these parameters, and to illustrate the applicability of the model to different air traffic control configurations. In this chapter, the results are shown numerically and graphically, then interpretations of these results are given.
- Chapter 6 concludes the research. It discusses the research and combines the ideas of the study. Finally, ideas and suggestions related to future work to extend the research study are stated in this chapter.
- Appendices provide with the object type declarations that are used in our program and a brief discussion of object-oriented Turbo Pascal language. A small glossary for the aviation terminology is also supplied as an appendix.

Chapter 2

Object-Oriented Programming

This chapter gives a brief introduction to object-oriented paradigm. The fundamental object-oriented concepts, which include *data abstraction, encapsulation, inheritance*, and *polymorphism*, will be explained in this chapter. The chapter also provides an overview of object-oriented languages and their evolution. Then object-oriented simulation will be introduced.

Object-oriented design and programming represent a major paradigm shift in software engineering during the 1980s. End-users, systems programmers, and application developers, in particular, simulation programmers, are all benefiting from object-oriented modeling and programming techniques.

2.1 The Object-Oriented Paradigm

A programming paradigm provides the system designer with techniques that guide problem solution. With the increasing demand in computer applications in all areas of science and business, software products have to satisfy more sophisticated requirements. Programmers must use more complex data structures and intelligent algorithms.

Program developers have long been educated mainly in the procedural paradigm. This is traditional, function-oriented approach to programming and

in this paradigm, subprograms (e.g.,functions, subroutines or procedures) are the most important part of the programs.

All code in a program is designed around the subroutines. Data are executed by the subroutines, they are *passive* in this process. It means that data are only passed to or from these subprograms.

For many years, programming languages and tools have been designed to take advantage of the concepts of structured design. Structured languages help software developers to write more organized code. However, in many respects these languages and tools haven't provided enough flexibility to handle the complexities of modern software requirements. Recently, by these motivations, there has been growing interest in utilizing some of the alternative paradigms to the procedural paradigm to facilitate the solution of certain types of problems [21].

1950s	**1970s**	**1990s**
Assembler	Structured	Object-Oriented
gotos	*blocks*	*objects*
jumps	*subroutines*	*messages*
variables	*recursion*	*classes*

Object-orientation is a newly accepted and important paradigm for improving software construction, maintainance and use. Object-oriented programming has changed the programming strategies and increased the speed of software production. In fact, it is not a new concept. The main ideas behind object-orientation occured in the late 1960s, but not widely used till 1990s.

Object-oriented paradigm has already demonstrated that it can help to manage the growing complexity and increasing costs of software development.

Object-oriented programming enables a program to be written with a focus on the description of the problem rather than algorithms for solving the problem. According to Meyer [17]:

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

Coding an application program in this paradigm involves creating a set of objects with the proper methods that will be invoked at the appropriate time through message passing among these objects.

Object-oriented programming has its roots in simulation. The first object-oriented programming language, SIMULA, was developed to provide simulation facilities within a general purpose programming language.

Object-oriented programming embodies four key concepts which result in making software systems more understandable, modifiable, and reusable. These concepts are: encapsulation, data abstraction, late binding, and inheritance. The definitions of these new terms will be explained in the following subsections.

2.1.1 Object and Class

In the object-oriented paradigm, objects and classes are the basic elements of the programs. A class is that software module which provides a complete definition of the capabilities of members of the class. An object is an instance of a class. In other words, a class is a template that is used to define an object. Rather than describing each object, objects of similar data content and behaviour are grouped together into a super-entity, called a class.

A class contains two types of components: data and procedures; so that it keeps both the characteristics of an entity (its data) and its behaviour (its procedures). By melding these characteristics and behaviors, a class, and hence its instances, know everything they need to do their work.

Class definitions declare reusable code in a common repository. Through

the *'class inheritance'*, new classes can be derived using the existing classes. This new classes, called *subclasses* or *childs* of the existing one, which is called, *superclass* or *parent*, have now their own code plus that of their parent.

Besides providing a brilliant mechanism for organizing information, the most important contribution of class inheritance is code sharing or code reusability.

2.1.2 Inheritance

Inheritance is a powerful object-oriented concept that provides software reusability and software extensibility. It 'taxonomizes' objects into well-defined inheritance hierarchies, by its natural organizing mechanism.

Inheritance is the ability to define new types of object which have some of the properties of old types, and some new properties of their own. New classes can be constructed from existing ones by extending, reducing, or otherwise modifying their functionality. New classes are specializations, or extensions of their superclass.

An inheritance hierarchy results from successive uses of this specialization principle. The inheritance hierarchy makes the definition of new classes more economical, since they can be derived from existing classes. Obviously, the deeper the hierarchy, the more functionality is inherited by the new class.

Inheritance provides a flexible programming environment that is organized in a hierarchical structure of object classes with reusable programs. This also leads to extensibility.

A base class may have multiple derived classes, and a derived class may in turn serve as the base class for other derived classes, producing a tree-structured organization of classes. In *single inheritance*, no more than one class can be base class for any derived class. However, in multiple inheritance, this can happen. Multiple inheritance permits a class to have multiple parent

classes. It is useful in building composite behaviour from more than one branch of a class hierarchy. When a class inherits from more than one parent class, there is the possibility of conflict of methods or variables with the same name but different, and unrelated semantics that are inherited from different superclasses. Furthermore, this feature violates the tree-structured organization of class hierarchies. Because of the reasons stated above, the multiple inheritance concept is not implemented by some of the object-oriented softwares [16].

2.1.3 Polymorphism

Polymorphism is Greek word for 'many shapes'. It is the method for changing the behaviour of a component that is shared by different objects.

As inheritance concept highlights, a derived class distinguishes itself from its base class by adding member variables, adding member functions, or redefining inherited member functions. In the last case, a member function declared in a base class may have several definitions since it may be redefined in multiple derived classes. When the function is called to perform an operation on an object, the definition actually used is determined at execution time based on the class of the object. This strategy is called *dynamic* or *late binding.* Binding in programming refers to the time at which values are associated with variables. In contrast to *early* or *static binding* (e.g., binding at the time of code construction) in traditional procedural languages, late binding provided in object-oriented programming languages delays the binding process until the software is actually running. Dynamic binding encourages placing the code that deals with a particular class of object in the implementation of the object's class, rather than in the client program, thereby making the client program more general.

By the dynamic binding feature, polymorphism places the responsibility for correct action on the object; the same message sent to many objects will illicit different, but appropriate behaviors.

Dynamic binding allows flexibility in the specification of object operations:

a message can be sent to an object requesting it to perform some action, without having to know what type of object it is, until run-time. Often the method for handling a message is stored high up in a class hierarchy. The method is located dynamically when it is needed. Binding then occurs as the connections are made between the method and the data local to the object. That is binding occurs at the last possible moment.

2.1.4 Encapsulation

Encapsulation is an object-oriented property that a software object is a completely self-contained entity, possessing all the data and code needed to perform a set of standard operations which form its only interface to other objects [17].

Generally, encapsulation serves three purposes:

- It protects an object's data from getting too much exposure,
- It makes it easier to use an object's data through the object's own interface,
- It is used to hide the details.

Encapsulation means that an object's data and procedures are enclosed within a tight boundary, one which can not be penetrated by other objects. Data stored within an object is directly accessible only by the procedures that have been defined as part of the class to which the object belongs. The use of objects therefore improves the reliability and maintainability of system code.

According to concept of information hiding, each software unit encapsulates its data and procedures and permits access to its internals only through a well specified interface. In object-oriented programming, objects communicate with each other by sending messages requesting them to perform their behaviors (*message passing*). Objects can query other objects to find the value of an internal state variable, but they can not directly change its value.

For many applications, it is desirable to decompose the program into a set of smaller parts which can be managed separately. There are two primary advantages of decomposing a problem into smaller parts: first, all elements and facets of the task can be seen more clearly; second, the parts can be developed and tested incrementally. The result of decomposition should be a set of modules which can be executed independently or combined to execute as an integrated whole. This programming technique is called as *modularity* and provides an important idea for structured programming. By the encapsulation concept in object-orientation, modularity is automatically reflected, since each class in an object-oriented program is treated as a module.

2.1.5 Data Abstraction

Data abstraction is the ability to treat the definition of a single object as an abstract entity which can then be used to define many further objects of the same type, all with independent data.

Data abstraction is implemented in a language as the ability to create new types of which variables are declared. In object-oriented languages, this is the ability to declare an object to be of a certain class. Class corresponds to an abstract data type and object corresponds to a variable of that abstract type. Data abstraction provides the ability to focus on what is relevant in modeling the problem. It makes the programming easier because there is more reusable code available.

2.2 Object-Oriented Software

Programming based on objects was first developed in SIMULA 67 language, evolved from SIMULA I, which is a simulation extension to the ALGOL-60 programming language. In the 1970s SMALLTALK was created as the first *pure* object-oriented language. It has evolved from the artificial intelligence

applications. SMALLTALK was influenced by SIMULA's model of computation and added the message passing paradigm, creating the object-oriented programming style [14] [21].

They are widely used in the academic area, but not so popular in commercial environment until the 1980s. At that time, C programming language was accepted as one of the best program development languages, by many software engineers. Bjarne Stroustrup introduced object-oriented paradigm to C to create C++ language. Because of its roots in C, C++ language introduced the object-oriented programming concept to many programmers without having to learn new and unfamiliar programming languages [11].

Authors state the fact that the choice of language is primarily based upon the availability of the language to the user and the user's knowledge of the language. This reason rather than the appropriateness or performance capability of the language is the overriding criteria.

Object-oriented programming usually requires less coding and its programs are easier to modify. However, the use of object-oriented programming was not likely to gain widespread use for simulations unless the language used is widely available and is familiar to a large number of simulation developers. Consequently, it appears that simulation through C++ or Turbo Pascal is more likely to be successful in this regard than other alternatives such as SIMULA, SMALLTALK, etc. Latter languages suffer from limited availability and a more restricted user base.

In addition, for the distributed simulation environments, object-oriented paradigm shows more promise, by assigning each object to different parallel processors.

Finally, it is important to emphasize that the use of object-oriented techniques is not dependent on any particular programming language. Rather it is an approach to organizing and planning computer programs which can be applied to a greater or lesser extent in all software development. Object-oriented programming is a paradigm, not a programming language. Therefore it can

be implemented in many languages, whether these languages are said to be an object-oriented programming language, or not. For example, object-oriented toolkits in ADA and FORTRAN, which are not object-oriented programming languages, can be found in [16].

2.3 Simulation Software

Recent improvements in computer hardware and software technology has brought wide horizons to simulation. Advanced technology provides with common and effective utilization of simulation models in the industry. This also resulted in the issue of selection of an appropriate simulation software for a particular application, among the wide variety of products.

2.3.1 Choice of Simulation Software

One of the most important decisions a simulation analyst has to make in the development of a simulation study is the choice of a particular language. A wrong decision may yield undesired results for the study.

For the selection of an appropriate simulation software, a simple methodology must be followed:

- detection of simulation software market,
- understanding of software classification and features,
- identification of the *modeler* and the *proposed end-users* of the simulation model and determination of their experience and skills.

Following the above approach, the main decision to be made occurs for deciding whether a general-purpose or special-simulation language to use.

For most of the applications, the features that are needed in programming discrete-event simulation models are common. Some of these common features may include:

- a reliable random number generator,
- probability distribution functions,
- future event list,
- simulation clock,
- abstract data structures such as queues,
- functions for data collection and database management,
- functions to generate reports,
- functions to handle erroneous cases,
- interfaces.

To build a simulation model by using a particular simulation software, the software is expected to have some more features;

- **Modeling flexibility:** Simulation software must have enough capabilities to handle variety of applications, to promote reusability.
- **User-friendly:** Simulation software must be easy to use and develop applications. *Interactive debuggers* and *on-line help* facilities are the two basic elements to improve model development time and model credibility.
- **Execution speed:** Since simulation models usually require multiple replications or multiple cases to be run, or both simultaneously, they use a lot of computer time. Even when computational resources do not present a direct cost to project, calendar time itself may limit the scope of the study.

- **Maintainability:** Experiences have shown that simulation projects are evolutionary in nature. Requirements change, the system being simulated changes, and the goals of the project change throughout the life of the study. Maintainability extends the life-time of the study.

- **Size:** Long code of an application means, difficulty for debugging and maintaining.

- **Portability:** If the simulation product is to be delivered to outside, or if there is a possibility of change of computer resources, usually this is the case; software is expected to be portable, that is, it can be run on various computing environments.

- **Customer support:** Good documentation and technical support assistance satisfy the simulation model developers' requirements.

After having discussed the desirable features of a simulation software, we can turn back to the question of selection among a general-purpose or simulation language. Now we can see some guidelines by the light of above discussion.

A simulation language, automatically allows the programmer to create models with the features listed above. This decreases the programming effort and development time. Studies result with smaller program codes, and hence the chance of making errors is reduced and debugging process is improved.

On the other hand, with the general-purpose programming languages, generally faster programs can be coded, since these programs are written for specific applications, ignoring the modeling flexibility. These languages are likely to be used widespreadly, known by many people. They support *duplicability*, that is more flexible to model any real-world system. General-purpose languages are available in most of the computing environments, with cheaper prices.

When we look at the history of simulation, we see that language of choice changes as the software technology evolves. Early simulation modeling was

performed using custom programs written in general-purpose computer languages, such as FORTRAN. Although this approach proved the viability of simulation modeling, the models were typically expensive and time consuming to design and maintain. Usually, the work done on a specific modeling project could not be easily utilized during subsequent modeling efforts. This resulted in simulation being used primarily on large, expensive projects.

In the early 1960s, as the field of simulation developed further, discrete event simulation languages such as GPSS, GASP and SIMULA were introduced. These languages were primarily written in general purpose languages but provided generic functions and subroutines to perform many of the tasks routinely required in simulation. At that time, the bulk of the simulation model development effort was still spent in developing problem specific code that had little reusability in future problems. In the late 1960s a second generation of simulation languages emerged. In most cases (e.g., GPSS V, SIMULA 67 and GASP IIA), these languages were more powerful replacements of their predecessors.

In the 1970s, as the use of simulation modeling grew, developments in simulation languages were driven toward the extension of simulation specific languages to facilitate easier and more efficient methods of model translation and representation. Many of the languages which evolved from these developments, GPSS, SLAM, and SIMAN, are still widely and actively used today [14].

In terms of continuing the growth of simulation modeling and expanding the use of simulation in general, construction of new simulation models and modification of existing models still provide formidable challenges to researchers. Also, the time required to construct and validate simulation models must continue to decrease through the use of concepts such as rapid prototyping and model reusability. Object-oriented programming appears to have the potential to be a major contributor to these areas of research.

2.3.2 Object-Oriented Simulation

Recent searches for better approaches for implementing simulation models is the recognition of the compatibility between simulations and recent developments in the area of expert systems; In particular, it seems that the object-oriented programming approach of expert systems is appropriate for implementing simulation models. Object-oriented design presents a more natural way of describing the problem with a one-to-one relationship between real-world objects and modules.

In general, modeling a real-world system can be viewed as identifying the components of the system and defining the interactions of these components with each other.

The guidelines of object-oriented simulation code development can be stated as follows [13]:

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

Consider the design of the discrete event simulation. Using a procedural design paradigm, the focus would be on an overall command loop which would be decomposed into subtasks as the design progressed. Data structures such as queues would be introduced as needed to support the algorithm. Using the object-oriented paradigm the main focus is on the entities in the simulation domain such as queues, servers and customers.

Each entity is defined abstractly in terms of a class. The actual entities in a problem solution are then represented as instances of these abstract classes. The instances are implemented as objects, independent regions of memory. For the discrete event simulation, classes would be defined for queues, servers, and customers. An instance of class customer would be created for each customer introduced into the simulation. In addition to identifying entities, object-oriented design also identifies relationships between these entities. These relationships help define the structure of the application design.

Once one accepts the principles of object-oriented program design, one must address the problem of decomposing the problem at hand into a suitable set of objects. This is usually not too difficult, if one can view the the proposed program as a model of some aspect of the real world. Then there is a natural correspondence between the objects being modeled and their program counterparts. This is most certainly true in the case of simulation where the specific intent is to represent objects from real life with a computer program. Nothing could be more natural than to organize the program structure around the objects being simulated. This aspect of the object-oriented paradigm is perhaps most significant to simulation.

Issues of modularity, maintainability, reusability, extendibility and the quite natural relation between real-world objects and their simulated counterparts all argue in favor of object-oriented programming technique. Object-oriented programming languages provide these advantages through such mechanisms as inheritance, dynamic binding, polymorphism, and automatic garbage collection, generally at the expense of execution time overhead.

As explained in the previous sections, inheritance mechanism of object-oriented paradigm reinforces reusability. By the class libraries, (e.g., toolboxes), frequently used classes are kept in separate program units. Subclasses of the classes in these libraries can be created by inheritance mechanism, without redefining their superclasses. Libraries provide modularity. If class libraries

are carefully designed, then these libraries do not contain any application-specific classes; rather, they have general classes to handle varieties of applications, so that flexibility is promoted. This approach also allow to build extendible models by the polymorphism principle; by which we can overwrite a member function of a class.

The key concepts of object-oriented paradigm reduce programming effort and increase project development speed. Carefully-designed simulation class libraries supports less sophisticated programmers, who are to assemble applications quickly from the prefabricated parts to model a system.

2.3.3 Object-Oriented Simulation Software

As stated above, object-oriented programming has its roots in simulation. SIMULA 67 was developed by Kristen Nygaard and Ole-Johan Dahl, to provide simulation facilities within a general purpose programming language. However, it has not gained widespread use for commercial simulations. This perhaps is due at least, in part to fact that it is an ALGOL based language and in many instances requires the writing of ALGOL subroutines in order to simulate a complete system. While SIMULA embodies some of the concepts of object-oriented paradigm, it is not a pure object-oriented programming language. One recent language based on the object-oriented approach is the SMALLTALK language, which added message passing paradigm to SIMULA.

SIMULA and SMALLTALK have found popular academic use, but have never gained widespread use in the commercial environment [13].

Until the 1980s, object-oriented simulation software is limited to academic research applications. At that time, C programming language was accepted as one of the best program development languages, by the authors. Bjarne Stroustrup introduced object-oriented paradigm to C to extend it to C++ language. The first implementation of C++, was developed as a preprocessor for C compilers at AT&T laboratories. Because of its roots in C, C++ language introduced the object-oriented programming concept to many programmers

without having to learn new and unfamiliar object-oriented programming languages. The availability of the language to the user and the user's knowledge of the language, makes the C++, the leading object-oriented language [11].

After the success of C++, some other general-purpose languages released their object-oriented versions. Object Pascal, Turbo Pascal 5.5, Objective-C, and COBOL are most popular ones among these compilers. They are not pure object-oriented languages, but provide more efficient and strongly type-checked implementations.

C++ features "in-line" function declarations, use of which can produce very efficient code. Another feature of C++ is useful in general and for simulation purposes in particular is the "friend" relation. A class can declare another class to be a friend, the second class then has free access to all of the internals of the first class. Currently, there are various simulation languages and packages based on C++, providing with the initial libraries to ease the model development. SIM++ is such a package for writing distributed simulations to run on multiprocessors, and DISC++ is a well-known discrete-event simulation library supporting event-scheduling and process interaction world-views.

With the version 5.5, that is released in 1988, Turbo Pascal had the object-oriented extensions. The "unit" feature, from previous versions, provides encapsulation of procedures with data and hiding. The object-oriented extensions provide single inheritance and virtual functions, hence polymorphism. The Borland integrated development environment facilities program development. It would be an acceptable medium in which to develop simulations. On the negative side is the fact that, at least at present, it is restricted to the PC world.

SMALLTALK-80 is a prototypical pure object-oriented language; everything without exception, is an object and all operations are accomplished by message passing. Automatic garbage collection is provided. It has a graphical interface that includes editor, class browser, debugger and object inspector. On the other hand, SMALLTALK programmers spend a substantial amount of time, familiarizing themselves with the workings of the initial class hierarchies

of the programming environment. SMALLTALK-80 has built-in support to facilitate the use of objects and classes of objects to model entities and events that occur in discrete-event simulations. The objects within such an environment, however, exist only for the duration of the simulation run. They must be created at the outset and saved (or destroyed) at the end. SMALLTALK provides the feature that automatic management storage reclamation of objects that are no longer needed. This mechanism is called *automatic garbage collection* in object-orientation. Recently, object-oriented methodology has been applied to data base technology. With the advent of object-oriented data bases it is possible to extend the life of objects beyond the run-time of a program.

Chapter 3

Air Traffic Control System

In recent years, airport congestion and delay problems have received a great deal of attention due to the rapid growth of air transportation services. The delay in an airport system increases rapidly when the air transportation demand approaches the maximum capacity of that system.

"Crowded sky syndrome" limits the development of air transportation. But, the fact is that the potential of the airspace have not been used completely as a transportation medium. It is the system that is crowded, not the sky.

In this chapter, we describe the general standard air traffic control system procedures and airport operations; and then look at a particular aerodrome to identify the practical use of these procedures to construct a model for air traffic control system.

3.1 Airport

Since all traffic originates and terminates at some point on the earth's surface, the overall efficiency of the air traffic control system is directly affected by the adequacy of the landing/takeoff areas, or "airports".

Improvements in airports thus must be undertaken constantly as part of

Aerodrome	*Passenger* (million)
O'Hare	64.44
Dallas	51.9
Los Angeles	46.9
London	45
Tokyo	42

overall improvements in the air traffic control system, and in keeping with the development of new aircraft and expanding air traffic volume. Unless airport progress keeps pace with all of the technological advances in aviation, airports can become one of the most serious bottlenecks in the path of efficient and safe air transportation.

An airport system can be divided into six components :

1. Airspace,
2. Runway,
3. Taxiway,
4. Apron-gate,
5. Terminal building,
6. Ground-access facility.

The first four items in the above list are involved in the air traffic system, while the last two items, theoretically, must not affect the air traffic. The layout of the airport terminal buildings and location of the airport have no influence on the aircraft movements, but they are likely to impose delays for the passengers (Figure 3.1).

Among the components of an airport system, the airspace and the runway are usually the critical components which limit the airport capacity.

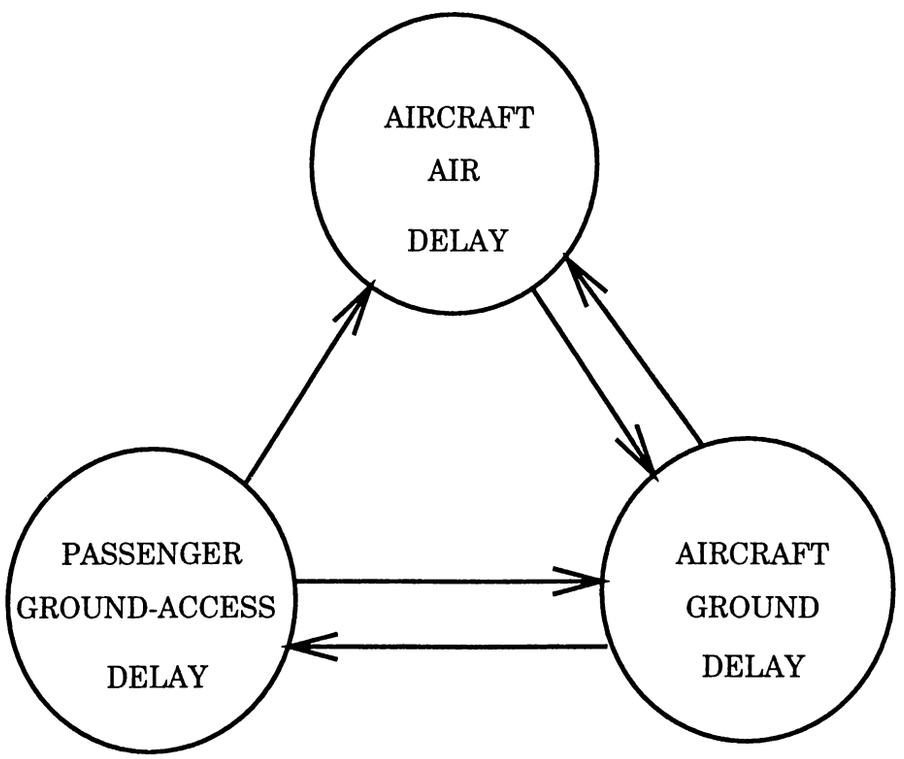

Figure 3.1. The classification of delays around an airport, showing the cause-effect relationships.

There are two major factors restricting the capacities of airspace and runway components: minimum separation criteria for two consecutive landing or departing aircraft and runway occupancy time. Limitation of the capacity results in delays. The delays which are imposed on airport traffic can roughly be classified into two categories; first is the delays due to the composition of the traffic, and second is that due to runway non-availability [20] [19].

The delays due to the composition of the traffic have been identified where the delay imposed was not due to runway non-availability, but due to the order of each aircraft in being sequenced by the air traffic controllers. Thus the same set of aircraft may require very different delays because of a slight change in their order. The minimum separation criteria are imposed on the landing and takeoff procedure to prevent aircraft mishap due to inadequacies in the air traffic control system, and due to the wake vortices generated by the leading aircraft. These criteria however, expected to decrease as a result of ongoing research efforts that address the reduction of wake vortex and the improvement of air traffic control system [5] [12].

The delays due to runway non-availability are affected by the number of runways in use, and separation minimum criteria, that is used similarly, to avoid collisions and wake vortex. Unfortunately, for the sake of safety, an aircraft is not allowed to use runway if another aircraft remains on the same runway, or in the final approaching phase of landing. Hence, the capacity of an airport would not increase proportionally as the minimum separation criteria decrease, unless the runway occupancy time decreases too; where the runway occupancy time is defined as the time interval from the instant the landing aircraft passes over the runway threshold until it completely clears the runway [7] [6].

3.2 Air Traffic Management

An easy solution to the delay problem is to limit the volume of air traffic which can use the airport under specified conditions or periods of time; but this is

not a cure for the problem.

In order to meet public demand in a total transportation concept is to apply more effective methods to achieve the highest possible degree of efficiency in the use of airspace and airports. This objective depends to a large extent on the capability of the overall Air Traffic Control System. The objectives of an air traffic control service, as defined by the International Civil Aviation Organization (ICAO), are to [1]:

- Prevent collisions between aircraft in flight
- Prevent collisions between aircraft on the maneuvering area of an airport and obstructions on that area
- Expedite and maintain an orderly flow of air traffic
- Provide advice and information useful for the safe and efficient conduct of flights and notify appropriate organizations regarding aircraft in need of search and rescue aid, and assist such organizations as required.

The degree of air traffic control exercised by this system depends basically on the meteorological conditions in which an aircraft is flown. When an aircraft can be flown clear of clouds and the pilot has good visibility, the flight is conducted in accordance with "visual flight rules" and is referred to as a *VFR Flight*. VFR flights are subject to little or no control by the ground facilities. It is up to the pilot, to watch out for the safety of his/her flight in the "see and be seen" environment. If a flight can not be conducted in accordance with VFR, it must be conducted under "instrument flights rules", or *IFR Flight*, and the ground facilities exercise positive separation control over all such flights. Traffic control procedures and parameters differ for the two types of flight conditions.

Long-range radars which provide position information, and computers which performs many of the routine functions of the controller, are utilized to assist the controller to achieve desired separation between aircraft.

3.2.1 Sectors

In order to maintain a controller's workload at a level which is within his capability to handle, the center's airspace is divided into *sectors*. This airspace is a defined geographical area which encompasses a number of airways or routes, airports, and navigation aids, and is also defined vertically [1]. Each such sector is assigned an appropriate number of controllers and assistants who are responsible for all aircraft in their designated airspace. In effect, the center's airspace is divided into small portions of the whole airspace, each of which will normally contain a small number of aircraft. Provision is made to combine sectors during periods of low traffic density and to further subdivide certain sectors when the volume of traffic reaches the point where a single controller can no longer handle the traffic.

Each sector has a controller who is directly responsible for the control of air traffic within his/her assigned airspace. The radar and communications equipment provide, in general, the means by which controllers receive position data on aircraft and through which air traffic center instructions are conveyed to pilots. In order to determine the correct instructions, it is essential that the controller be fully cognizant of the position and future plan of every aircraft within his/her sector.

3.2.2 Air Traffic Control Services

Air traffic control services are provided by *area control centre, approach control office* and *aerodrome control tower.* These services are provided to all IFR and VFR flights in controlled airspace and to all aerodrome traffic at controlled aerodromes.

1. *Area control service* : the provision of air traffic control service for controlled flights, except for those parts of such flight described in subparagraphs 2 and 3 below, in order to accomplish objectives of preventing collisions between aircraft; and expedite and maintain an orderly flow of

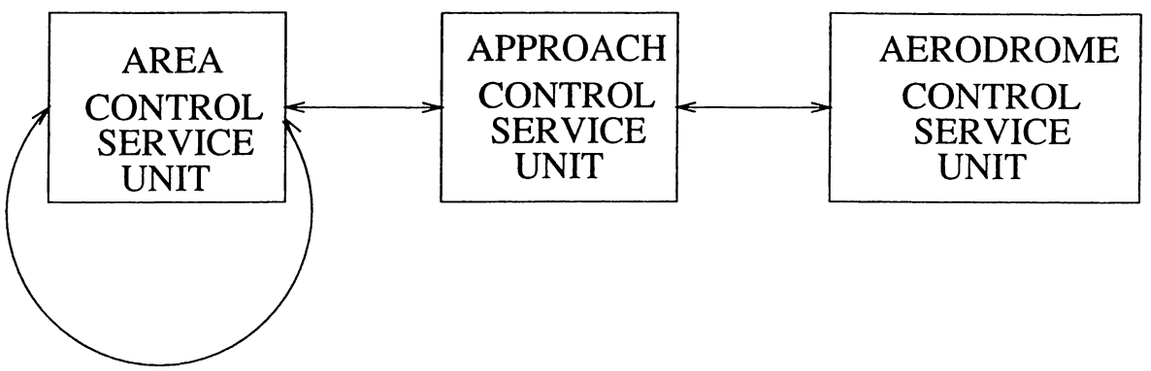

Figure 3.2. Flight path for an IFR-flight-aircraft in a controlled airspace.

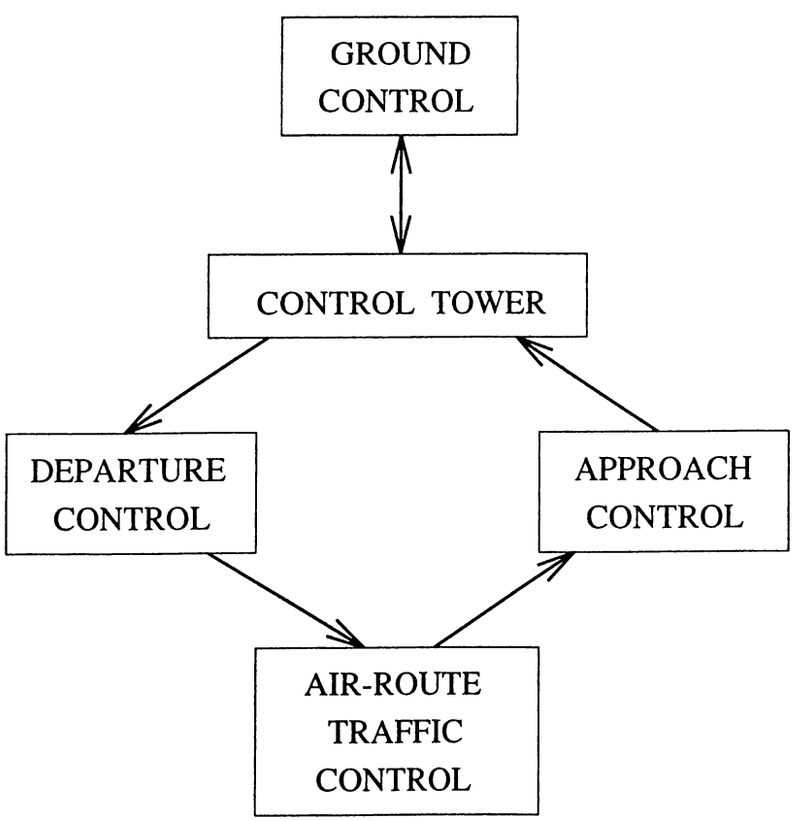

Figure 3.3. Air traffic control centers and their relations with each other.

air traffic. This service is provided by an *area control centre* or *approach control service unit*, if no area control centerΛ is established. They are also called, *air route traffic control center*, since they provide enroute traffic separation to IFR flight aircraft in controlled air space. They are responsible for large areas of controlled airspace. This facility is divided into several sectors, each of which use its own frequency and responsible for radar coverage of a particular area.

2. *Approach control service* : the provision of air traffic control service for those parts of controlled flights associated with arrival or departure, in order to accomplish the same objectives as listed above. This service can be provided by an *aerodrome control tower* or *area control centre* ; or by an *approach control office* when it is necessary or desirable to establish a separate unit.

   These facilities control an area of from 25 to 60 nautical miles from their airport. Arriving traffic is passed from the cognizant center to the approach control sector, and the reverse process takes place for departing traffic. In certain cases, the approach or departure control sectors may be subdivided into smaller sectors to reduce the controller workload.

3. *Aerodrome control service* : the provision of air traffic control service for aerodrome traffic, except for those parts of flights described in sub-paragraph 2 above, in order to accomplish the objectives stated before, plus for preventing collisions between aircraft on the manoeuvring area and obstructions on that area. This service is provided by an *aerodrome control tower*. The tower normally accepts air traffic from approach control service at about the point where the aircraft can be visually identified. The control of air traffic on or in the vicinity of an airport is provided by this facility.

   Departing aircraft are given instructions regarding when and how they may taxi from loading ramp to the runway in use, followed by takeoff clearance when the pilot is ready and the traffic permits. Arriving aircraft are handled by the control tower in a similar manner, by "clearing" the aircraft to land when airborne and ground traffic permits, and then

by issuing appropriate taxi instructions to guide the aircraft to its unloading point. There may be an additional unit to help the tower, called *ground control* or *ramp control.* This unit governs the movement of aircraft and other vehicles on the airport, excluded the active runways. It takes aircraft from apron to runways or vice versa.

Figure 3.3 gives the direction of control of an aircraft using an airport. Figure 3.2 shows a typical flight path of an aircraft through the air traffic control centers.

Under the present air traffic control system, when an aircraft reaches its destination, the approach control provides the necessary directive to maneuver aircraft and aline each with the desired runway for landing. Aircrafts are permitted to land on a first-come first-served basis, under the normal conditions; in approach sector, a separation minima between aircrafts is enforced. A normal landing will ensue if tower controller give the clearance, and if the pilot, in approach maneuver, is able to stabilize his/her aircraft in a landing configuration by the time the minimum decision altitude of 200 feet is reached. After exiting the runway, landing aircraft is directed to its assigned gate, if it is available, or to a parking place on the apron, using the taxiways. Departing aircrafts are similarly, put into a depart queue, and cleared to roll on the runway; if the aircraft is in the first place, and if there is no other aircraft on the same runway and in the *final approach sector.* Departure controller maintains the flow of air traffic till the departure fix, which is the exit point of aerodrome control area, on the air-route.

3.3 Simulation Modeling of ATC System

In this thesis, our objective is to study and analyze the effects of different parameters and strategies, that are applied by the air traffic control system, on the total delay of aircrafts in that airspace. In order to achieve this objective, we select a crowded airport, which has suffered from the critical capacity problems in recent years, to study on. We decided to use the İstanbul Atatürk

Airport for our modeling purpose. This was the biggest one with the most serious delay problems, among the total of 22 airports in Turkey.

We tried to build a model that can be applicable to any air traffic control system configurations. We also tried to be as realistic as possible, since the ICAO procedures mentioned earlier are subject to change by the particular countries, with their own responsibility. During our observations, we saw that, no one of the ATC systems, applies the ICAO rules strictly. The limits, that are imposed parameters by ICAO, can be implemented in a flexible way, to ease the job of air traffic controllers.

3.3.1 İstanbul Atatürk Airport ATC System

İstanbul Atatürk airport is the biggest one of the five international airports that accept scheduled flights, in Turkey. In fact, 47 % of total air traffic and 45 % of total passenger traffic of Turkey, are handled by this airport. For the last few years, the increment rate of aircraft and passenger traffic in this airport is beyond the all forecasts, that are made by international ATC authorities. The growth brings on the capacity problems; for both of passenger terminal buildings and aircraft maneuvering spaces.

The Atatürk Airport

The Atatürk airport has two *converging* runways, (36/18 and 24/06), that is, extension of centerlines are intersecting each other. The dimensions are: 3000 $\times$ 45 meters for runway 36/18 and 2300 $\times$ 60 meters for runway 24/06. The former runway has seven turn-off points to exit, and the latter has six. The current runway capacity is 40 movements (take-off or landing) per hour, and under adverse conditions (4 to 5 months per year), operations are limited to single runway use with the capacity of 24 aircraft movements per hour.

The only capacity enhancement for the runway, that is planned, is an extension of 700 meters to runway 24/06 is under study. By the implementation

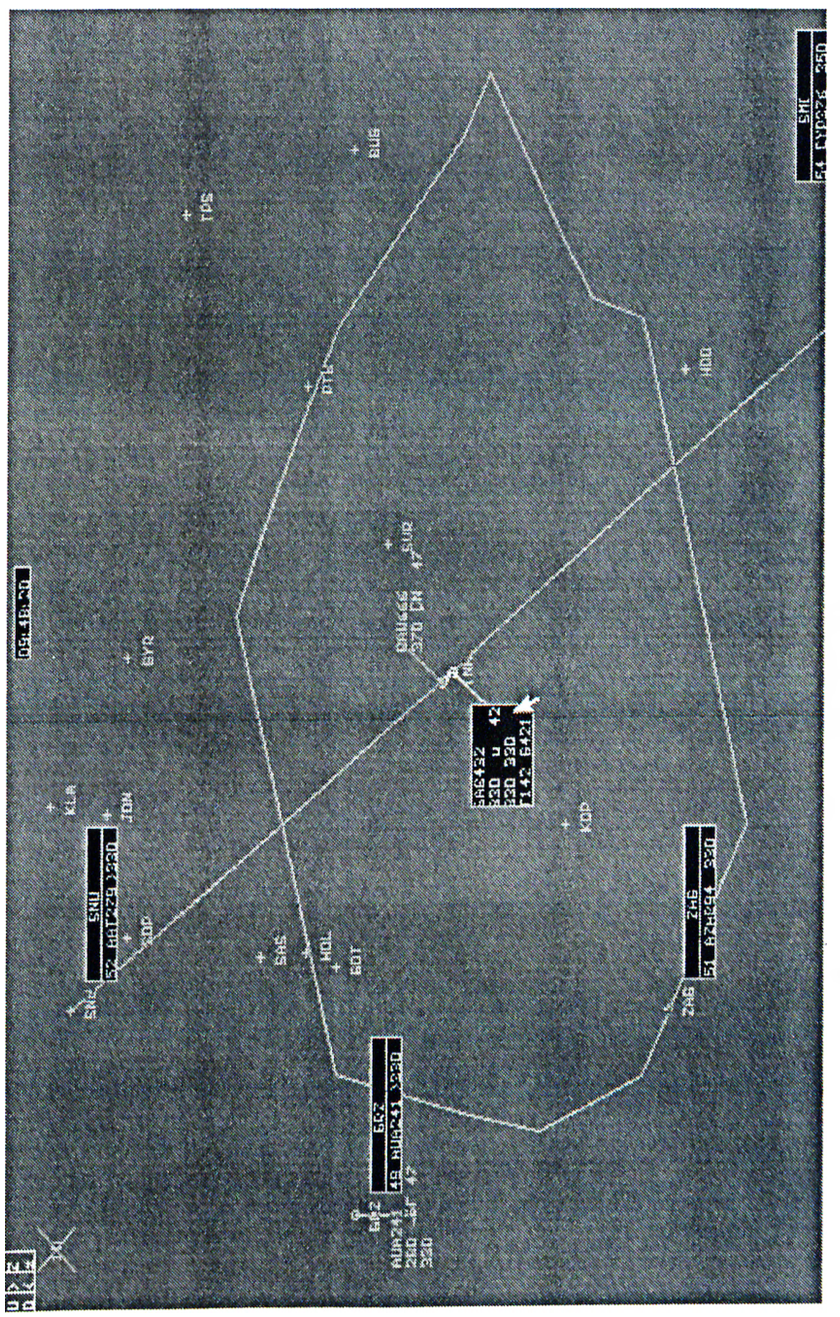

Figure 3.4. ATC radar screen.

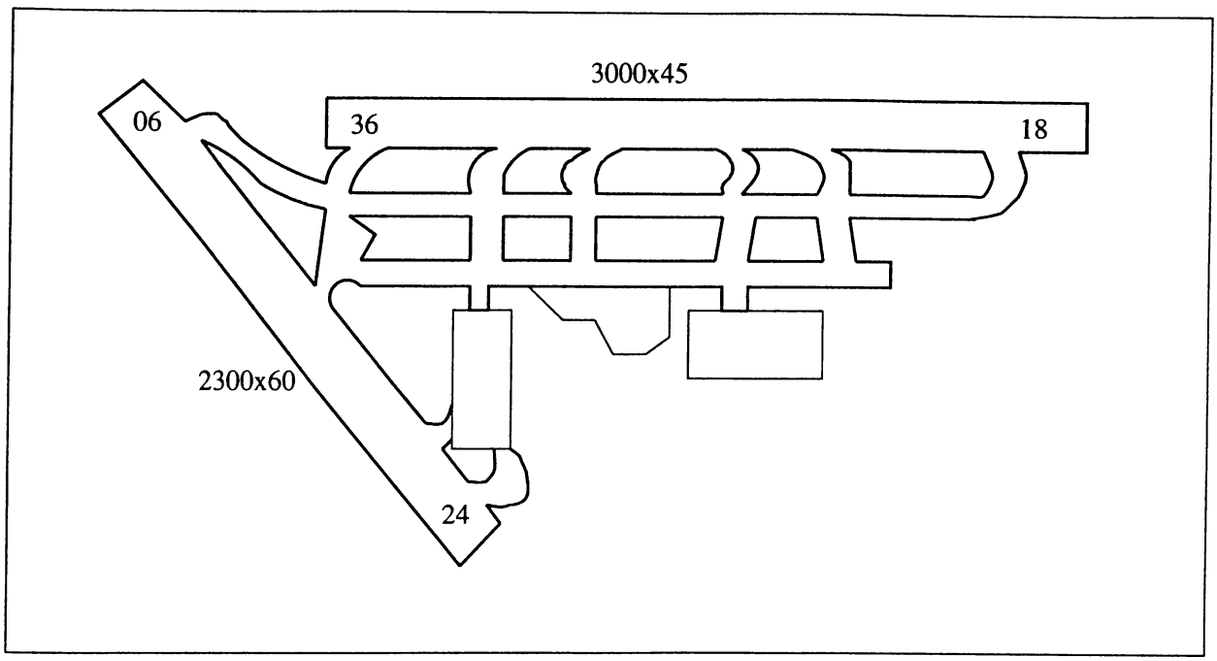

Figure 3.5. İstanbul Atatürk airport layout plan (not to scale).

year	*capacity*	*passenger*	*% increase*	*peak day*	*peak hour*
1991	7,500,000	5,204,500	-	21,188	1,674
1992	7,500,000	7,371,600	42	37,936	2,751
1993	8,500,000	9,396,230	27	47,675	3,215

Table 3.1. Passenger traffic and increase rate in the İstanbul Airport for the last 3 years.

of this project, runway performance capabilities will be enhanced.

The tables 3.1 and 3.2 reflects the growing traffic capacity problems in the İstanbul airport.

There are 11 *apron* areas for parking, and these areas can hold 64 aircrafts. Nevertheless, inadequacy of parking places imposes a constraint on the traffic of the airport. For the emplane/deplane processes, there are 9 gates at the international terminal building, and the rest is performed by mobile ramps.

year	*capacity*	*aircraft*	*% increase*	*peak day*	*peak hour*
1991	350,400	82,206	-	322	23
1992	350,400	109,508	33.21	390	29
1993	350,400	139,771	27.64	449	36

Table 3.2. Aircraft traffic and increase rate for aircraft traffic in the İstanbul Airport for the last 3 years.

In 1994, with the construction of the Charter Terminal building, the terminal capacity shifted up by one million to 8.5 million passenger. The airport has two car parking areas with a total capacity of 1236 cars.

The air traffic flow on the airport is currently under study by European Air Traffic Control Organization, called EUROCONTROL. For the temporary solution to the capacity problems, a constraint program is conducted by the airport management.

The İstanbul ATC System

As mentioned in the preceding subsection, the İstanbul airport management imposes a special flow control mechanism on the scheduled flights, to lower the congestion of airport traffic for certain days of seasons and certain hours of the days.

The main objective of flow control service is to regulate or restrict the flow of IFR traffic within an affected area or at specified altitudes to the maximum number of aircraft which can be safely accommodated by the ATC system.

Normally, for any busy airport a flow control mechanism, described below, is applied. Flow control forecasts are issued periodically to indicate the anticipated delays expected to apply during specified periods of time –usually not more than two hours. Flow control normally is applied when arrival delays will exceed 15 minutes and are expected to prevail for an extended period of

time; when route segments require preventive action to avoid traffic saturation; where traffic flow is disrupted due to a breakdown in navigation facilities; or where weather conditions have caused excessive delays in executing normal landing procedures.

Advanced flow-control procedures are implemented during peak traffic hours as may be required to hold aircraft on the ground at points of departure until the ATC system can safely and expeditiously handle them. The appropriate ATC center calculates the hourly demand on the airport and then determines an acceptance rate based on forecast weather and runway configuration. When the demand is forecast to exceed the acceptance rate beyond certain tolerances, the advanced flow-control procedures are placed into effect to regulate the flow of traffic so as to distribute delays equitably among all users. This eliminates en route holding except as may be necessary when approaching the destination airport, for delays on the ground are less costly than holding in the air; namely fuel consumption on the ground is 75 % of that in the air. Moreover, waiting on the ground is obviously, more safely than flying in a *Holding Stack* pattern, which is a portion of terminal airspace, where waiting aircrafts kept orderly, before proceeding. These are rectangular air patterns, separated vertically for the queued aircrafts.

The main difference between these two forms of flow-control procedures is that the first method achieves its objective by holding aircraft in the air while en route. The advanced flow-control concept, on the other hand, holds them on the ground at the departure airport and is designed to handle the more extreme congestion problems. In order to apply the advanced procedure form, any aircraft which wants to use İstanbul airport either for landing or take-off in its flights, must declare this beforehand, to the airport management and hence to the flow-control center. However, ambulance, VIP, technical and military flights, and the flights of the aircrafts that have less than 12 seat capacity are excluded from this obligation. The negative results of this exclusion can be best understood, if we see that runway 24/06 is sometimes used for training purposes of Turkish Air Forces; and VIP flights are very frequent in any hour of the day.

Various modifications to these basic systems are applied wherever congestion during peak periods presents a significant delay problem. These techniques, however, only provide temporary relief for air traffic delays. They do not present basic solutions to the underlying problem, which is to increase airport/airspace capacity. After having seen the capacity constraints, now it is time to analyze the operations of the İstanbul Air Traffic Control system.

3.3.2 Initial Approach

Air traffic flows through 7 gates, or arrival fixes, into İstanbul airport *Terminal Airspace*, which is a cylindrical portion of airspace above the airport and controlled by the air traffic control centers established on this airport. Each of the gates are approximately 60 nautical miles away from the airport runways; in fact, they are the intersection points of the terminal airspace and the airways, or flight corridors. The first sector of the terminal airspace that an aircraft flies into is called *Initial Approach Sector*. Normally, an aircraft's speed is about 140-200 knots (e.g., nautical miles per hour) at the arrival fix and it takes an airplane almost 20-25 minutes to fly to the threshold points of a runway. Arriving traffic is passed, or handed off, from the cognizant air-route traffic control center to the approach control sector, and the reverse process takes place for departing traffic. Approach control center sequence the aircrafts and establishes the appropriate spacing between them (approximately 5 miles of separation is required, that is, about 2 minutes). Spacing is required to avoid wake turbulence, which is a pair of counter-rotating vortices trailing from the wing tips, generated by a leading aircraft and therefore, depends on the characterization of the air traffic, that is, the specific features of the each aircraft in the traffic sequence. To illustrate this, if an aircraft follows a heavier one in this sector, the appropriate spacing may be 3 miles; but for the reverse case, spacing will be shorter than 3 miles. An approach controller watches the position and related data of each aircraft from the output screen of the radar, called *Secondary Surveillance Radar*. The data contains the altitude, speed, type, direction and flight number of the aircraft, as well as the name

of the sector which is currently controls the aircraft. The data on the radar screen are automatically send by the transponders of the aircrafts flying in the range of the radar; and they are updated in almost every 5 seconds for each aircraft. The controller of each sector recognizes and directs his/her traffic from the radar screen. If the aircraft flights in the visual flight rules, then all the responsibilities to avoid the collision belong to the pilot, by the principal: "see and be seen". In that conditions, pilots of the VFR flights can move their aircrafts faster, and reduce the separation between the aircrafts. However, if an aircraft has the sufficient instruments to fly in IFR, and the pilot is capable, then the pilot may require to fly IFR, even in the *Visual Meteorological Conditions.*

A sector controller can handle at most 10 aircrafts within the approach sector. If an aircraft comes into the sector exceeding this limit, controller just holds the aircraft to fly in circular patterns, to have some time to decrease the traffic. Such queued aircrafts are later sequenced into the path to landing sector on a first-come first-served basis. However, this rule can be broken, if an emergency condition occurs for an aircraft in the queue, or VIP flight arrives to the sector. The proper way to handle these queues is to establish *Holding Stacks* in the certain portions of the approach sectors. Holding stack is a rectangular patterns separated vertically for the queued aircrafts. An aircraft arriving to the holding stack, joins to the queue from the lowest pattern, and the aircraft currently flying in this pattern is shifted up to the next layer. Aircrafts leaves the stack from the top layer. This strategy enables an effective way of airspace utilization. Holding stack establishment has been experienced in İstanbul airspace, but cancelled later, due to inadequacy of technology and trained personnel.

It is experienced that, normally a controller can handle 24-30 air traffic within one-hour period. This limit can be shifted upwards by dividing approach sector into smaller sectors and assigning controllers for the new approach sectors. In current configuration, İstanbul airport has only one approach sector and one depart sector, both of which are controlled by the same person at any time. Most of the congested airports employs more than one approach sector;

and for the İstanbul airport a new sectorization of terminal airspace is studied under the supervision of EUROCONTROL.

The Initial approach controller directs the arriving aircraft to a point from which an aircraft can directly land on the runway-in-use. If the weather conditions (especially wind direction) allow, the preferred runway normally is 36/18. If the number of aircraft movements are greater than 20 per hour, and if the weather conditions allow, then landings are assigned to runway 24/06, and take-offs are assigned to runway 36/18, to minimize the delays. However, in any case, most of the heavy aircrafts require the use of the longer runway 36/18, for landing and take-off.

İstanbul airport employs an Instrument Landing System which is the basic ICAO standard navigation aid for landing. During the approach, aircraft slows down to *Instrument Landing System Glide Slope*, which provides navigational guidance for landing derived from radio signals transmitted from ground-based electronic aids located on the airport. By this aid, pilots can also land in bad weather conditions with low visual range.

3.3.3 Final Approach

After aircrafts fly through the initial approach sector, they reach the *Final approach Sector*, where the aircrafts are finally sequenced in order of their landing and runway usage. In the final approach sector, the longitudinal separation between aircrafts is determined and a minimum of a 3-mile separation is maintained. After the aircraft departs from the Instrument Landing System (ILS) gate, it continues to descend the glide slope until it reaches the decision point. If the runway is vacant, a normal landing will ensue; otherwise, the aircraft must wave-off.

The final approach sector begins from a distance 6 nautical miles from the runway. Average flight time in the sector is 1.5-2.5 minutes unless a wave-off is required by the sector controller. This sector is controlled by the control tower

on the airport. The separation minimum enforced by the sector controller is 3 miles (1.5 minutes).

The Instrument Landing System in this sector provides guidance to the pilot in both the horizontal and vertical planes with respect to the landing runway of the airport. Along the ILS approach path, *Very High Frequency Fan Markers* are provided at three positions to supply the pilot with spot-checks as to the his/her distance from the runway. These are called *Outer Marker* (about 6 miles from the runway end), *Middle Marker* (3,500 feet from runway end), and *Inner Marker* (at the end of the runway).

For the precision-approach procedures, that is instrumented approaches there is a *decision height*, which is 200 feet measured from the sea level, at which the pilot must make a decision to either continue the approach or execute a missed approach procedure (turns and re-enters the landing process, as the first aircraft on the queue). A pilot may miss the landing process, if the runway-in-use is not available or the pilot can not stabilize the aircraft in the desired position when it reaches the decision altitude.

During the flight to the touchdown point on the runway, the arriving aircraft in this sector not only has exclusive use of the runway it is approaching, but in addition holds the other runway unavailable to use. The hold on the runways are released and the alternate runway becomes available to subsequent arrivals and departures when the landing aircraft touches down. But the active runway used by the landing aircraft is still occupied by the aircraft rolling on it and becomes available to the other aircrafts as soon as the landing aircraft turns off a taxiway. The rule here is, if there is an aircraft (landing or take-off) on a runway, then no other aircraft is allowed to enter the runway. Moreover, if there is an aircraft in the final approach sector, no departing aircraft can go on the runway-in-use.

These rules are to satisfy the safety requirements; but can be broken by a mistake of a pilot or by the controller to decrease the length of the depart queue. Latter case is applied at some times when the departures are more than arrivals. In this intervals, if the length of the depart queue exceeds 5 or 6 aircrafts and

if the queue in the air is shorter than this, then the tower controller may decide to merge the these two queues by increasing the spacing between the landing aircrafts and inserting departures into these time gaps. Although the cost of flying is higher than that of waiting in the queue, if the overall waiting times are considered, on the average, the delay in the departure queue is much more than other delays occured during a flight. For example, the İstanbul air traffic management reports the delay times of the aircrafts departing in the morning, exceeding 30 minutes for the summer seasons. (To illustrate, taxiing cost for a Boeing B747 aircraft is $ 24.50 per minute.) Fortunately, the departures are congested around 8:00 a.m. (GMT) and batches of arrivals come approximately 4 hours later, around 12:00 p.m. (GMT).

Finally, after an aircraft touches down on the active runway, pilots of the aircraft put brakes on and decelerate the aircraft to a necessary speed to make a turn to exit the runway. The runways has the *speed exits* before the final exit at the end of them, to allow capable aircrafts to taxi to apron quickly. An average runway utilization time of an air-carrier is about 1 minutes. By exiting the runway, the control of the aircraft is passed to the next air traffic unit, called *Ground Control.*

3.3.4 Turn-Around

After an aircraft starts to move on the taxiways, the ground turn-around segment is started. The ground controller *ramp controller* assigns an appropriate gate to the aircraft for emplane, deplane, loading and unloading operations. If no gate is available, but will be available in 15 minutes, then ramp controller holds the aircraft on the taxiway, to wait for the gate; otherwise, the turn-around operations are made on a parking place of an apron. This may lead another delay for the *through flights*, that is, the flights landing to use the airport for short period of time not as the terminating airport. The *terminating flights* using a gate also assigned and sent to the apron after the completion of deplane and unloading operations. The *originating flights* taxi from their parking place to a gate, if one is available, to load and emplane passengers.

The gate assignment policy used at the Atatürk airport is to give the priority to international and VIP flights, because of the lack of the gates.

In fact, the number of parking places on the aprons are insufficient either. There are 64 places, 4 of them are always kept reserved for *emergency landings* and VIP aircrafts. If no place or gate is available, then landing aircrafts held on the taxiways till a place is emptied. Obviously, no turn-around operation is allowed for an aircraft in this position. İstanbul ground traffic control unit reports such kind of delays up to 45 minutes.

After completion of the turn-around segment, pilots of the aircraft asks permission to push back from the gate, 15 minutes before starting engines. Because of the parking place inadequacy, the assigned push-back priority is:

- Leaving (departing) aircraft
- Arriving (landing) aircraft
- Aircraft moving from one place to another place

Aircraft moves from the gate or apron and taxies to the take-off runway, using the taxiways. At each taxiway intersection a check is made of the traffic ahead on the assigned route to runway. Typical taxiing speeds are 12 to 20 miles per hour. At the end of the route, aircraft stops to take the last clearance for entering the runway. If there is a queue of departing aircrafts, then aircraft joins this queue, and waits its turn on the taxiway.

Upon claiming the runway, the departing aircraft leaves the holding pad. Approximately 20 seconds later, the aircraft reaches its take-off point. A final check is made to determine whether the runway is clear of crossing planes before the aircraft rolls along the runway to its lift-off point. The aircraft airbornes in about 45 seconds. After the airborne departure control takes the control of the aircraft and watches the aircraft till it reaches the departure fix, which is at the same time, an arrival fix.

Although a departing aircraft flies in the same sectors as it was arriving, the

behaviors of these sectors are different in the sense of modeling. Because the spacing enforced on departing aircrafts is shorter than the arriving ones, and since aircrafts are faster in the departure route, the pass times of the sectors will also be shorter than that was previously.

3.3.5 The Modeling Approach

To build a model to simulate all the details of a system to the same degree of fineness would be not only time consuming and expensive, but foolish, wasteful and in fact, impossible. Because of this reason, at the beginning of a project; the results desired and obtainable from simulation must be clearly defined. The simulation model can then be built around these to a degree of fineness to give sufficient details to aid the planning and design.

By establishing aircraft arrival and departure patterns through the different sectors, and with defined procedures for arrival, landing, take-off and departure; as well as the separation factors between the successive aircrafts, the basic model of the airport system described above, can be constructed. The queue time, on the ground and in the air, of each aircraft, using the airport, hence can be simulated.

For the simulation modeling, the object-oriented design is used because of its appropriateness to simulation design and easiness to code in a object-oriented programming language. By these features, we were able to construct a very simple model for the İstanbul airport air traffic control system, and later we could enhance it to more realistic cases and to different airport models. The object-oriented paradigm provided us with modularity and extendibility as well as reusability. One of the aims of our study was to create a class library, or toolkit, for an airport simulation model, and demonstrate its benefits. Therefore, the model had to be flexible enough to be easily applied to any particular airport system using present or future component equipments, procedures and configurations. The main goal in constructing the model was complete versatility in describing the system, i.e., the model was designed to be general

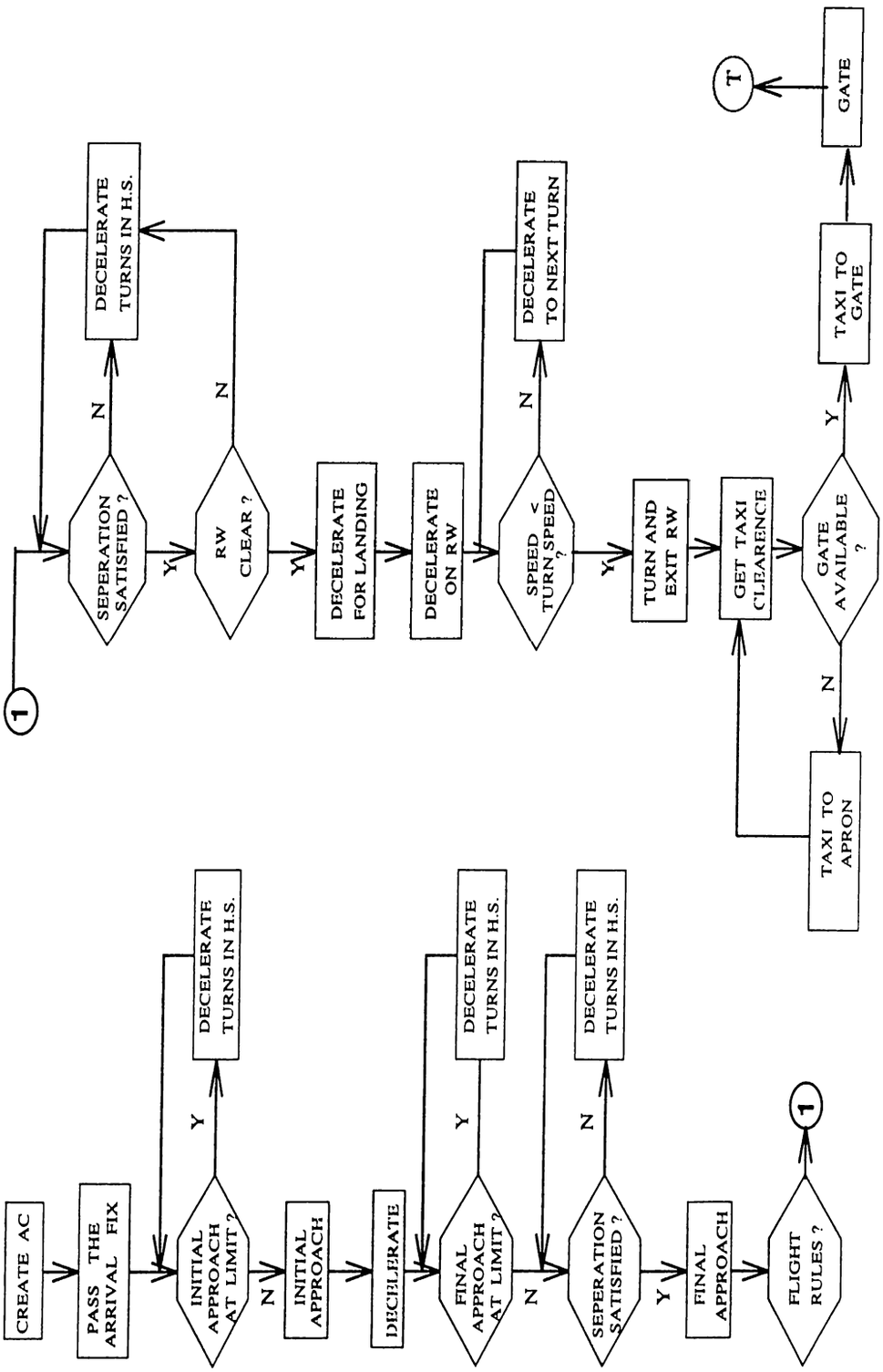

Figure 3.6. Arrival segment flow diagram.

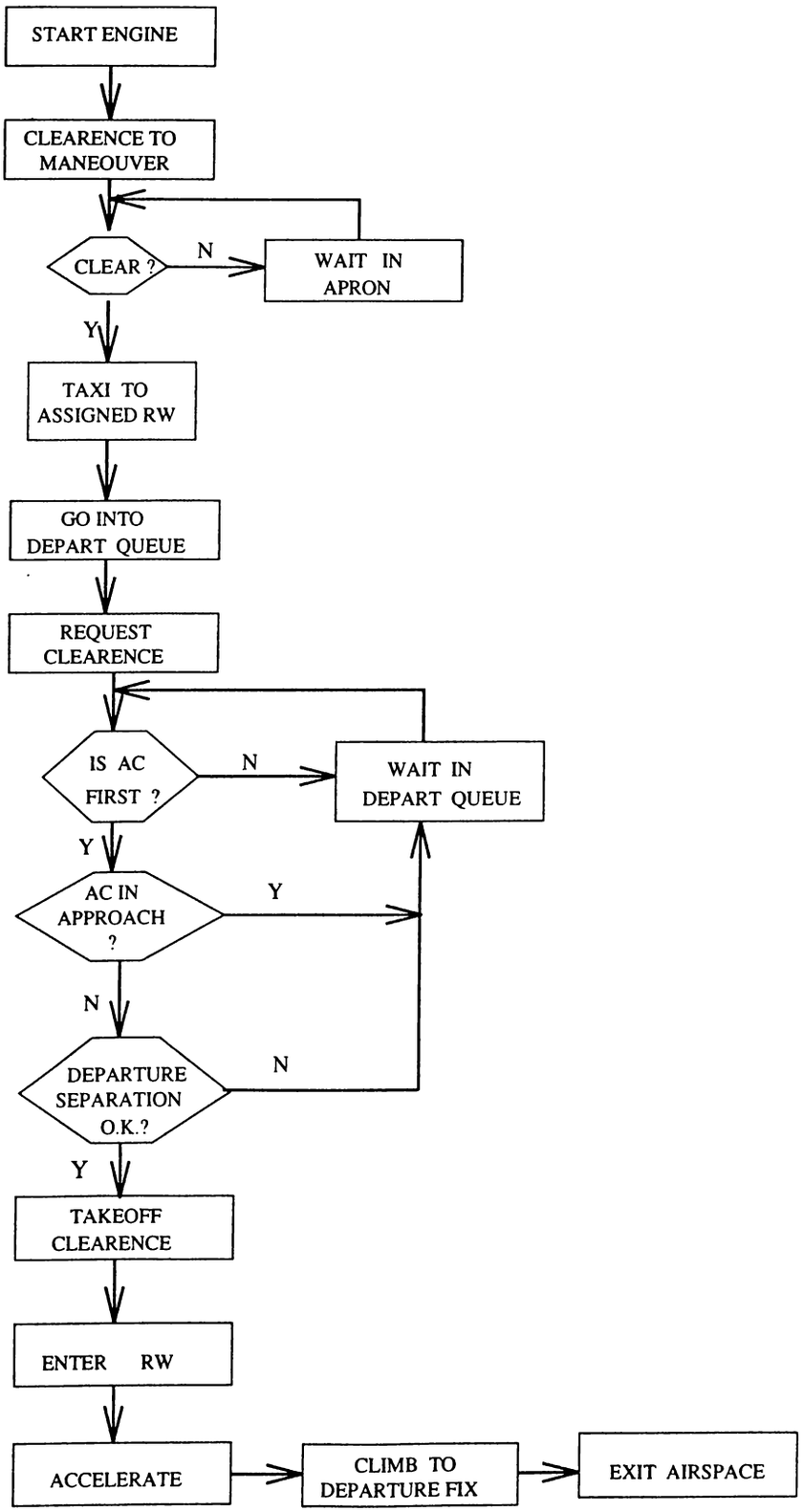

Figure 3.7. Departure segment flow diagram.

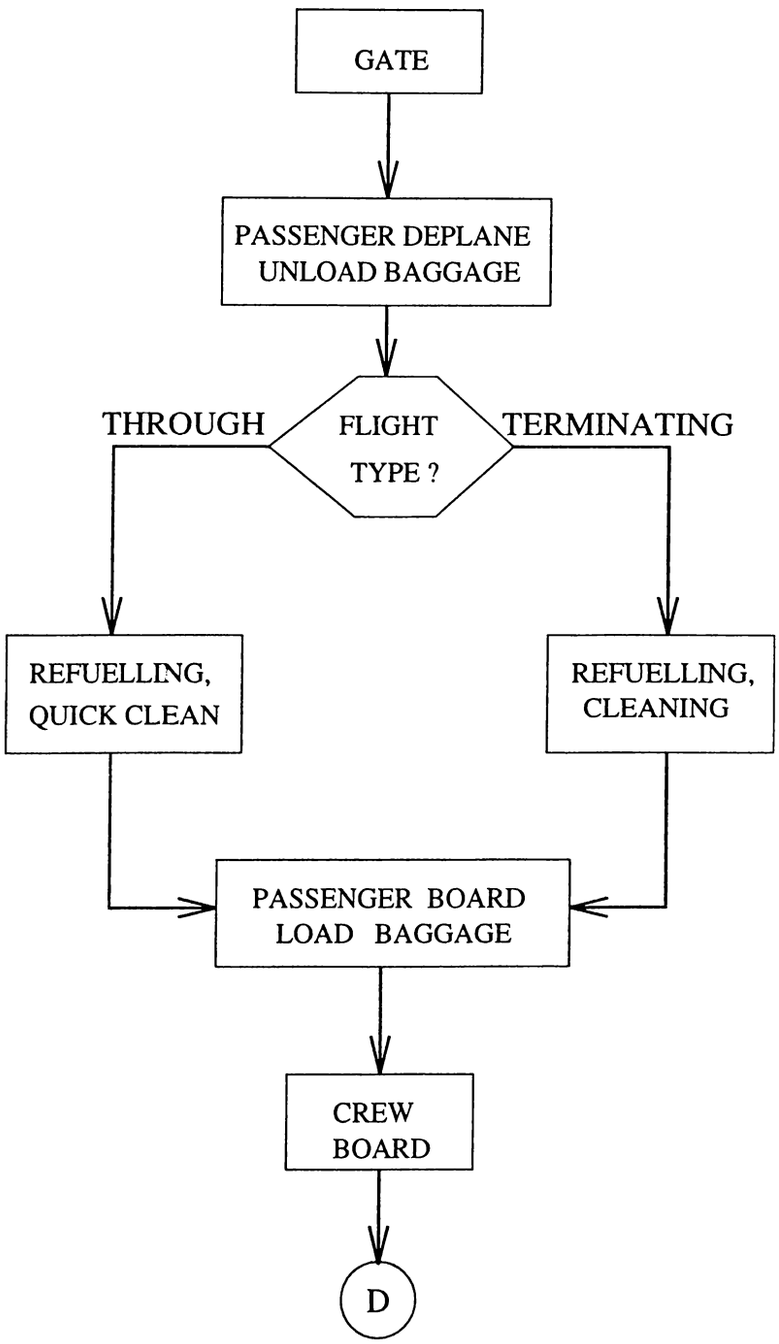

Figure 3.8. Turn-around segment flow diagram.

enough to simulate the terminal area operation of any airport, regardless of its size, location, or geometric constraints.

The main task of a modeling study is to identify the entities of the modeled system and interaction of each entity with the other entities of the system clearly. Therefore, we started the study by analyzing the system. For this purpose, we started with reading the ICAO documentations about the standard airport procedures, and we talked to air traffic controllers, two major airports (İstanbul Atatürk and Ankara Esenboğa International Airports) were visited. From the beginning of the study, visits to the Ministry of Transportation Civil Aviation General Directorate, have been made regularly.

By this analysis, we decided on the basic design of the model. The airport system is viewed as including initial approach sector, final approach sector, turn-around segment, various types of aircrafts, runway capacity and procedures, and the air traffic control capacity and procedures. Each of the approach sectors has the characteristics such as, capacity (determined by the controllers of this sector), separation minima enforced, holding stack (a queue on a first-come first-served basis), and length (pass time of an aircraft). Each type of aircrafts has a flight type (terminating, originating or through), an assigned turn-around time for apron operations. Hence the system components are identified as:

- *Entities:* Aircrafts.
- *Resources:* Runways, Initial Approach Sector, Final Approach Sector, and Apron Areas,
- *Events:* Arriving to and departing from the approach sectors and apron.

A through flight aircraft arriving the terminal airspace, can be seen to move into three segments:

- *Arrival Segment* includes Initial Approach Sector and Final Approach Sector.

- *Turn-Around Segment* includes Apron Areas.
- *Departure Segment* includes Final Approach Sector and Initial Approach Sector.

A terminating aircraft uses only the first two segments and an originating aircraft uses the last one.

The time of an aircraft's arrival at the system is the time when an arriving aircraft enters the initial approach sector. When an arrival occurs, the time of the next arrival is determined, by reading from the input data file.

The aircraft flying in the initial approach sector follows a route by directions of the sector controller; if the sector is at capacity or the separation minimum is not satisfied for the sector, then the controller delays the aircraft by the directions, to gain some time. Consequently, the aircrafts in the sector are queued, but in the real case, there is no physical queue in the air. To model this behaviour of the initial approach sector, we order each aircraft flying in this sector and determine time of exiting the sector for the first aircraft by checking the previous exiting time. To find the exiting time of the first aircraft, we take the later of, current time plus sector pass time and the previous exiting time plus sector separation minimum. As a result of this procedure, no similar delays are imposed in the next sector, the final approach.

If a runway is available at the time of arrival to the final approach sector, the aircraft is given an immediate landing clearance and proceeds on final approach without any delay. If neither runway is available, the aircraft joins a queue and waits its turn. This is the only reason for an aircraft in the final approach sector to wait in the air. Aircrafts, flying in this sector, are also served on a first-come first-served basis. A landing aircraft in the final approach sector occupies the all runways of the airport. Therefore, no aircraft movements (landing or take-off) is allowed, from that time till the landing aircraft exits the runway.

After an aircraft lands, the ground turn-around time is generated, if the flight is a through flight, by calculating the maximum of departing time of the

aircraft read from input file minus the current time and the minimal required turn-around time. An apron departing event is then scheduled at the time, current time plus the generated turn-around time. This event is inserted into current eventlist at proper position regarding its event time.

For a departing aircraft, the processes are in the reverse order of an arriving one. Departing aircraft enters the control of the final approach sector controller, when it comes the area on the taxiway, where take-off queue is established. If there is no aircraft in the air flying in the final approach sector or on the runway, and moreover, if the required spacing with the previous aircraft using the runway is satisfied, then the first aircraft in the take-off queue is cleared to depart. The model decides the take-off time, by regarding the runway-use-priority-policy used at the airport. After an aircraft airbornes, it reaches the next sector at 6 miles away, and immediately flies to an appropriate departure fix at the boundary of the terminal airspace. A departing aircraft in the air is supposed not to be delayed by the airport air traffic control system. At the 60 nautical miles from the airport, the aircraft is handed off to an *Air-route Traffic Control Center* to keep contact till the next control center, which is either an approach control center for the destination airport or another air-route control center.

In the airport model, the design of the operations of which is given above, we model airspace and runway utilization of an airport; although we said that these are the only 2 of 6 components of an airport. For the sake of simplicity and generality, we decided not to include the ground components and operations of an airport into our model. The complete model is very complex and procedures applied are dependent on particular system being modeled. Furthermore, in our belief, the airspace and the runway is the most important components and more sensitive to slight improvements, within an airport system. By a consequence of object-oriented design that we applied to our model, the other components of a particular airport system can be designed and build later on. And each component, as a module, can then be connected to our model.

Figures 3.6, 3.7 and 3.8 illustrate the three consecutive segments of a flight

for an aircraft which is arriving an airport for landing.

3.3.6 Modeling View

The discrete event simulation of the system, under study, is a sequence of *events* taking part at different points of time; where an event describes a happening to an entity of the system. The simulation executes each of these events, one-by-one, as time progresses. The *executive* is the part of a simulation program that makes all events take place in the correct order with respect to their pre-determined occurence time. Moreover the executive advances the simulation time from its current value to a new value, which is determined by the next event's occurence time. To keep track of events of a system, an event list is employed in the model. This event list is usually called the *Future Event List*, or simply the *Calendar*, in the discrete event simulation modeling. Each cell of an event list contains an event of the system with its activation time. The list is created by connecting these cells in the order of their occurence time.

In order to implement the future event list in the simulation computer program, a linked list of records, containing events and their *event-times* together with a pointer to the next record of the list, is coded.

When the simulation program starts, it needs a non-empty calendar, i.e., the list must be initialized at least by one event cell beforehand, to keep the simulation proceed as needed.

Mainly there are two distinct types of events in the discrete event simulation:

1. *Bound Events* (or *B-events* shortly) are the predictable events, so that they can be scheduled. Because of this reason, they are also known as *scheduled events.*

2. *Conditional Events* (or *C-events*) are the ones whose occurence is dependent on certain circumstances.

Among a few modeling views in the discrete event simulation, one we choose for our model is the *Three-Phase Approach* to structure the simulation. The reasons behind that choice will be explained in the succeeding paragraphs.

In the three-phase approach, bound events and conditional events are programmed as separate procedures, and conditional events are performed independently of bound events. Only the bound events are scheduled and put into the future event list, since their occurence time can be pre-determined. At each time advance (that is, occurence of bound events), after executing the bound event which is to happen due that time; the executive checks all the conditional events to find the ones whose trigger conditions have already fulfilled as a result of the performed bound events. The conditional events may schedule bound events, or even other conditional events. Since the latter case is possible, the simulation modeler must take care of the checking order of the conditional events. That completes the main cycle for a three-phase model [18].

To summarize, for a simulation model employing the three-phase approach, the three phases of the executive, during a simulation run, are as follows:

- advance the clock to the time of the next scheduled event,
- execute all bound events due to happen at this time,
- test all conditional events and execute those whose conditions are satisfied.

By this view, event procedures are shorter and more readable than those of any other views, when they are coded. The simulation programs written by three-phase approach is more modular, and more robust to changes than other methods.

The main disadvantage of the three-phase approach is its relative inefficiency. The executive of the simulation program must do more work because all the conditional events have to be tested after each bound event has been

performed. As the size of a model increases, the number of conditional events increases and thus, the number of wasted calls to conditional events increases.

Following the three-phase approach, in our simulation program, we initialize the first bound event, which is either an arrival to the terminal airspace or departure from the apron. The former event is an arrival to the initial approach sector for landing, whereas the latter is an arrival to the final approach sector for takeoff. The data is read from the input file which must contain only those two types of events and their occurence times.

The simulation program checks the type of the event currently read, and invokes the proper subroutine to handle this type of event. Obviously, the program executive read and then execute only one bound event, in each turn. Each of the bound events determines a departure from a sector and an arrival to next sector (for an arrival to the first sector, there is no departure from any sector; similarly, for the last one no arrival to any sector). A departure from a sector, sets a flag to indicate that, preserving the spacing between the aircrafts, an aircraft succeeding this one may depart from the sector. This flag is recognized by the corresponding conditional event for this sector and if there is an aircraft after that one, an arrival event (which is a bound event) for the next sector will be scheduled by this C-event.

Each bound event is responsible for updating of its sector status and sector queue. The data input, or aircraft generation, is also determined, only by the B-event subroutines, as well as the exit of an aircraft from the simulation model.

After that phase, program comes to a loop to visit and check all the conditional events one-by-one. Among the C-events, the ones whose conditions have satisfied, schedule one bound event and may change the parameters for other conditional events. Each conditional event checks the queue status of a sector; if the queue is non-empty and the sector is in ready-for-depart state then this event schedules the next depart event at the appropriate time (of course, if it is in the simulation run period). In addition, this event causes to change the sector status being "READY" to "BUSY", unless its tests fail.

Event Name	Event	Effects
INIARR	Arrival to Initial Approach Sector	Insert aircraft into sector. Read input file to generate next arrival. Schedule this as a bound event. Changes sector status into READY
FINARR	Arrival to Final Approach Sector	Depart aircraft from initial approach sector, and insert into this sector. If take-off then generate new one from input file, schedule as B-event. Changes status into READY.
APRARR	Arrival to Apron Place	Determine flight type. If it is through then find its turn-around time.
EXITING	Exit Simulation	Aircraft exits simulation. Collect the statistics for it, then free its memory allocation. Changes status into READY.

By having visited all the conditional events, the executive turns control of the simulation program to the point where it started to cycle by reading a node from the future event list. In each turn, the simulation time, or *simulation clock* is advanced to the time of the next scheduled event.

This modeling view of the terminal area traffic control system is fairly realistic and well represents the current system in the İstanbul Atatürk airport terminal airspace, chosen to illustrate a model for the general situation.

3.3.7 The Model Input Data

In the development of the simulation program and later in the experimentation phase we use various kinds of data. The data and related documents for the airport operations are found by various resources, including:

- The Ministry of Transportation Civil Aviation General Directorate,

Event Name	Event	Effects
INIARRCHK	Check Initial Approach Arrival Queue	Schedules FINARR event if Queue is nonempty and status is equal to READY. Changes the status into BUSY.
FINARRCHK	Check Final Approach Arrival Queue	Schedules APRARR event if Queue is nonempty and status is equal to READY; unless departs congested, changes the status into BUSY.
FINDEPCHK	Check Final Approach Departure Queue	Schedules INIARR event if Queue is nonempty, status is equal to READY and if Arrival Queue is empty; unless departs congested, changes the status into BUSY
INIDEPCHK	Check Initial Approach Departure Queue	Schedules EXITING event if Queue is nonempty and status is equal to READY. Changes the status into BUSY.

- Middle East Technical University, The Department of Aeronautical Engineering,
- The İstanbul Atatürk Airport Control Tower and Ramp Control,
- The İstanbul Atatürk Airport Flow Control,
- The Ankara Esenboğa Airport Control Tower,
- The General Directorate of the State for the Management of Aerodromes,
- Turkish Air Lines Management for Schedule,
- European Organization for the Safety of Air Navigation (EUROCONTROL),

In the beginning of the study, ICAO documents about the standard airport operations and airspace control procedures are analyzed. The ICAO procedures were flexible to different implementations of the countries. They propose necessary airport operations for minimal safety requirements. The real airport operations are watched and recorded at the two airports mentioned above.

In order to construct the model, the necessary parameters are determined by speaking to air traffic controllers of the two airports, observations at those airports and the chief controllers from civil aviation general directorate, who have previously worked at the İstanbul Atatürk Airport Control Tower. These data were:

- separation minima used in the terminal airspaces,
- separation minima for the runways,
- alternative runway operation strategies,
- runway occupation times,
- pass times for sectors of the airspace,
- park place counts,

```
W93IST OHY0352          03NOV23MAR 931103 9403230030000 174320
       0000 1815ESB     C NW9320OCTIST     K          ILKHALOO
W93ISTAZ 0700AZ 0701    31OCT26MAR 931031 9403260004007 150M80
FCO    1110 1220FCO     JJNW9310SEPIST     X          ILKHALOO
W93ISTXQ 0790XQ 0791    31OCT20MAR 931031 9403200000007 148733
VIEVIE 2055 1550VIEVIECCNW9316OCTIST          O      ILKHALOO
W93ISTLH 3846LH 3847    31OCT26MAR 931031 9403261234567 144320
FRA    1035 1135FRA     JJNW9306OCTIST     X          ILKHALOO
W93IST        TCT1441   01NOV21MAR 931101 9403211000000 176722
       0000 0500BCN     C NS9319MARIST          ES    N
W93ISTTCT1442           01NOV21MAR 931101 9403211000000 176722
BCN    1255 0000        C NS9319MARIST          ES    N
W93IST        TCT0131   01NOV20DEC 931101 9312201000000 176722
       0000 1545GRZ     C NS9319MARIST          AT    N
W93ISTTCT0132           01NOV20DEC 931101 9312201000000 176722
GRZ    2115 0000        C NS9319MARIST          AT    N
W93IST        TCT0211   02NOV22MAR 931102 9403220200000 167M83
       0000 1500DUS     C NS9319MARIST          DE    N
```

Figure 3.9. A sample page from the raw data file which is obtained from the İstanbul Airport. Each record contains flight season, airliner name, date and time of flight, aircraft type and flight type.

- maps of the airports and airspaces,
- capacities of the air traffic controllers,
- other operations such as gate assignments.

Furthermore, the main input data for the simulation program was the arrival and departure times and hourly counts of all aircrafts that use the İstanbul Atatürk Airport. Fortunately, since currently an improvement program is under study for parking areas of the Atatürk Airport, the airport management with the help of Turkish Air Lines, records the statistics of movements on the airport. These records consist of the *flight plans* of all scheduled flights in year 1993. The data size was 6 megabytes in the computational environment. The data was in compact formal ICAO format. A sample page from these records

is given in Figure 3.9. We processed the data and divide it into daily format for each of the days we want to simulate. Each records contains, aircraft type, seat capacity, flight number, flight type and days of the flight. This information helps us to derive various results by processing the daily files. To illustrate, we can trace the number of passengers arrived and departed each day; or the number of parking places that are available at the end of the each day.

However, those are intended data and subject to change by air-carrier companies. Moreover, these data do not include the followings, which are equally important for our model:

- Ambulance flights,
- Some technical flights,
- Military operations,
- Some small aircrafts,
- VIP flights.

Although these flights are sometimes very frequent, they can be accepted as insignificant for the congested summer seasons.

In addition to these files, for the validation purposes, two-day real data in the August of 1994 is obtained from the airport control tower. The simulated results of the operations of these days are used to compare with real values.

Chapter 4

Implementation of the Model

This chapter deals with the design approach and implementation of the air traffic system that we have developed and described in the previous chapter.

The object-oriented design of the classes and objects will be introduced and these classes will be listed. The declaration of the object classes, i.e., the body of the classes consisting of data (attributes) and behaviors (methods), will be given. Alternative source codes to model different airspace compositions will be illustrated, in order to demonstrate the power of the object-oriented design and programming paradigm that we used in the development of our simulation study. Finally, outputs of the program are listed and sensitivity of the results will be discussed.

4.1 General Object Type Design Issues

First action to map a model to program code, programmer should decide the data representation of the components of the model. In object-oriented design process, to identify objects and object-classes of the program, the following issues must be considered to evaluate the data types for the components of the simulation model. In fact, the items listed below are the necessary and sufficient conditions for a data type to be accepted as an object in a program.

1. The entities of the system must be listed, and proper data types for them should be described.

2. It must be determined that whether the data type is a self-contained unit, or it is more appropriate to represent this data type as a self-contained unit. This determination can be made by observing the conditions that users of the simulation program should or should not care about all aspects of the data type.

3. Then, it must be decided that whether the data type, or data structure, contains information that can be kept out of control of the main program; that is, if the main program does not need to access to all attributes of the data structure, an interface can be written for this data type to request to access to any part of it.

4. Finally, the following criteria should also hold to declare a data type as an object: if the data type is seem like to be used to derive new and similar data types later in the program or for the future use; furthermore, if this derivation is more appropriate and easy by inheritance of the data structure, instead of redefining the later data types.

These points lead us to use object-oriented design for the data types. By analyzing the above issues, we decide to declare a data type of the program as an object.

In our program, as well as we have class definitions of entities of the airport system we have also the classes of the simulation models, that are the *auxiliary components* of the program such as the *eventlist* and the *clock*. Obviously these components also have to satisfy the same conditions as any other entity of the system, to be declared as an object.

Although, the object-oriented approach to problem decomposition encourages the development of objects which reflect the underlying structure of the problem, developers must decide the level of detail to include in an object's definition. Moreover, the object type definitions must be general enough to allow flexibility for new simulation developments, easily.

4.2 Design of the Airport System Components

For the airport simulation model most of the objects of the model is obvious for us. All parts of the system that change their status, physically or logically, by the events of the model are implemented as objects of the program. For example, all aircrafts moving (or parking) in the simulated system, air-sectors that are the skeleton of the airspace and the airport itself are among such components that can be treated as objects. In fact, in our design all objects that have physical counterparts in the system are the ones listed above. The other objects of our design are the ones that are generally used in the simulation programs. These are the *auxiliary* objects and are used for running and analyzing the system. These include eventlist and statistical data collector.

Each of the components of the system, and program, mentioned above were good candidates to be declared as objects, because their assigned data types have "private" information fields that should be kept out of attention of the program; furthermore, they are self-contained units, that is, each of them should update its status (data) by a request after each step of the simulation, so that they can communicate with each other through their own interfaces; finally, their importance for the model promising reuse and enhancements for the future simulation studies.

As an example, we can look at an aircraft data type declaration, moving in the airspace. Each aircraft coming into airspace are generated by the program, by reading a data file, possibly containing aircrafts as records and arrival, departure time, flight number (which is the *key field* of each aircraft record), flight type, aircraft type and name of airliner as the fields of each of these records. Each aircraft's data structure must be constructed and initialized at the time of aircraft's arrival to airspace and it must be destroyed when it is no longer needed; because of less memory usage. Therefore, an aircraft data type needs appropriate creator and destroyer subroutines, in order to initialize and destruct this data type, that are specific to it. Clearly, the input file's detailed information for each aircraft is not important to main program and users, but by implementing these information, aircraft's behaviour in the

Object-type	Purpose
ACObject	represents an aircraft that is flying or parking in the terminal airspace
Queue	keeps a list of aircrafts flowing orderly in the sector
SecObject	represents skeleton of a sector
IniApp	represents Initial Approach Sector
FinApp	represents Final Approach Sector
Apron	represents Apron area of the airport system
Airport	represents current status of the airport
DelayObject	represents an auxiliary object type to collect statistics of cumulative delay times of aircrafts leaving airspace
ClockObject	stores the simulation clock
Eventlist	represents future event list of the simulation

Table 4.1. The objects of the program.

terminal airspace is determined. During the program run, the executive calls the aircraft data type to respond to an action made by any component of the system. In this call, the executive does not care of the actual type of the aircraft, but the correct reaction is performed by a function of this aircraft type. Since various aircraft object types have the same function name for that reaction; however, the source code of the function is different and specific to its object type. In other words, the same request arrives to each object types, but each object type responds in its own way. Each aircraft is, therefore, a self-contained unit by its own "private" data (attributes) and own methods that manipulates its data. And as a last point we can consider a new aircraft type that can be introduced later in the study for an extension of the definition of the current aircraft data structure. To illustrate, for a better and detailed simulation study, one may want to add more data fields to the definition of the aircraft, such as, velocity, position, fuel-state, or some different behaviors, such as, turning, acceleration, etc.

This observations about the aircraft data type are enough to give the reasons to declare aircraft as an object type.

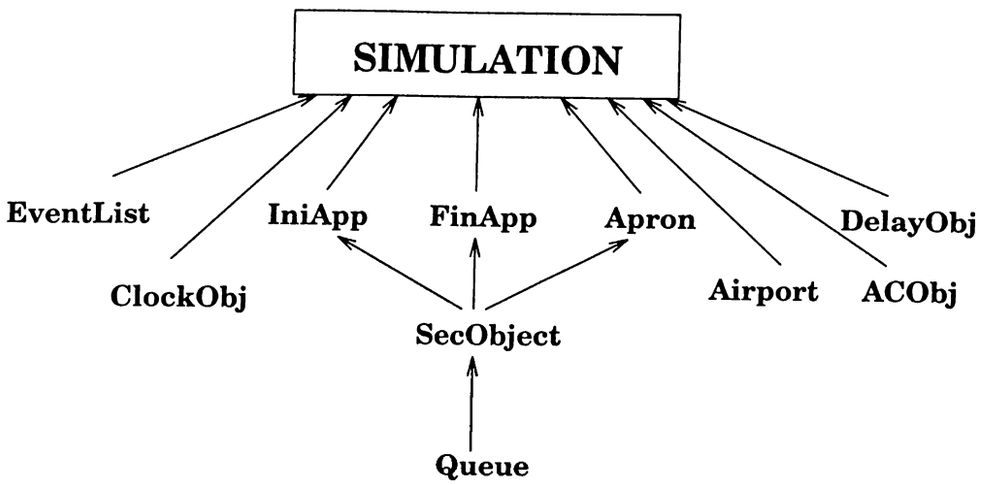

Figure 4.1. The hierarchy of the object types that are used in the airport simulation program.

```
Eventlist = object
        eventcell : evptr;
        countcell : integer;
        constructor initialize;
        procedure Addevent(etype:Etypes; actyp:ACptr;
                                    etime:clock_rec);
        procedure Getevent(var nextevent:anevent;
                                      var LOG: text);
        procedure Printlist(clock:clock_rec;  var LOG: text);
    end;
```

Figure 4.2. The type declaration of object type EVENTLIST.

Table 4.1 shows all of the objects that are included in our simulation program.

The hierarchy of the object types is given in Figure 4.1.

Figure 4.2 through figure 4.11, on this subsection, show the some parts of the type declarations of these objects listed in table 4.1. The complete listing of the simulation program objects will be given in the appendix. (Appendix A).

Figure 4.12 illustrates the structure of the QUEUE object-type graphically,

```
ClockObject = object
        clockrec : clock_rec;
        constructor Initialize(t:clock_rec);
        procedure Advancetime(mm:real);
        procedure Printtime(var log:text);
      end;
```

Figure 4.3. The type declaration for the object type CLOCKOBJECT.

```
DelayObject = object
        aclist : delayptr;
        constructor Initialize;
        procedure Delaystats(delay:clkrec; name:string);
        procedure FindMaxDelay(var LOGF:text);
     end;
```

Figure 4.4. The type declaration for the DELAYOBJECT.

```
ACobject = object
      flight : flighttyp;
      typeAC : ACtypes;
        ...
      constructor Initialize(key:string; deltime,cretime,
                        enttime:clock_rec; kind: actypes;
                                           ant: integer);
      destructor  ExitSim(var LOG:text);
   end;
```

Figure 4.5. The type declaration for the ACOBJECT object.

```
Airport  = object
      wpark,npark : integer;
      procedure Starting;
      function Full(arracount:integer):boolean;
      procedure ParkAC(AC:acptr);
      procedure UnparkAC(AC:acptr);
   end;
```

Figure 4.6. The type declaration for the AIRPORT object.

```
queue = object
    que       : quetype;
    hourlywait  : array [1..(endtime div 60)] of real;
    hourlyaccnt : array [1..(endtime div 60)] of integer;
    nextsect : sectorptr;
    constructor Initialize(nexts:sectorptr;
                                    passt,sm:clock_rec);
    procedure Insert(timerec:clock_rec;
                            var newAC : ACptr); virtual;
    procedure Remove(timerec:clkrec;
                    var waited:clkrec; var oldAC: ACptr;
                     k:integer; var LOG:text);  virtual;
    procedure Quedep(timerec:clkrec; AC: ACptr;   waitd:
            clkrec; key:integer; var LOG:text); virtual;
    procedure Average;
    procedure Little(min:real);
    procedure Find_w;
    function  Empty:boolean;  virtual;
    procedure Print( var LOG: text);
    procedure Stopping(timerec:clock_rec);
    procedure Print_hourly(var LOGF:text);
 end;
```

Figure 4.7. The type declaration for the QUEUE object.

```
SecObject = object
     sectname : string;
     Inque,Outque : Queue;
     constructor Setdata(sm1,sm2,pass:clkrec; key: string;
                  nextone1:sectorptr; nextone2:sectorptr);
     procedure Check_Queue(var Q:queue; var attime:clkrec;
                                 var sch:boolean); virtual;
  end;
```

Figure 4.8. The type declaration for the QUEUE object.

```
IniApp    = object(secobject)
       procedure Exiting(timerec:clkrec; ac1:acptr;
                                        var LOG:text);
       procedure ArriveSect(timerec:clock_rec; var ac1:
               acptr; var cal:eventlist; var LOG:text);
    end;
```

Figure 4.9. The type declaration for the INIAPP object.

```
FinApp   = object(secobject)
      procedure Check_Queue(var Q:queue; var attime:
                  clkrec; var sch:boolean); virtual;
      procedure ArriveSect(timrec:clkrec; var ac1:acptr;
                      var cal :eventlist; var LOG:text);
   end;
```

Figure 4.10. The type declaration for the FINAPP object.

```
Apron  = object(secobject)
    procedure  ArriveSect(timrec:clkrec; var ac1:acptr;
      var cal:eventlist; var ap:airport; var LOG:text);
    procedure SortQue;
 end;
```

Figure 4.11. The type declaration for the APRON object.

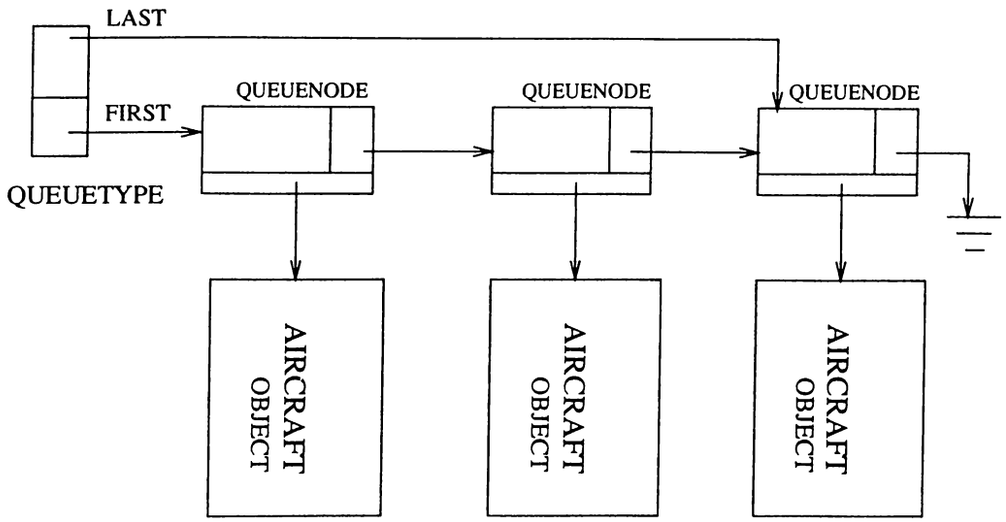

Figure 4.12. The structure of QUEUE object that uses dynamical allocation.

```
(* main program *)
begin
    STARTSIMULATION;                { Initialize Simulation }
    while not ISLATER(clock,endclock) do
      begin
        TIMEADVANCE(clock);      { Update Simulation Clock }
        EXECUTEEVENT(clock,calender);    { Update Calender }
      end;
    REPORTS(clock);                      { Report generation }
end.
```

Figure 4.13. The main program source code of the simulation model.

which has a complex data structure.

4.3 Program Execution

After having discussed the design of object types, we can consider now the cooperation of these object types in the simulation program.

The simulation program has mainly three stages, and in fact, the main program body reflects this stages clearly (Figure 4.13):

```
0745 1700 SLL 0746 01AUG 24AUG 0801083 10000007 149 AB4
2355 9999 THY 0791 01MAR 17MAY 0801083 10000007 177 320
1210 1310 BSP 0242 01AUG 30SEP 0801083 10000007 144 733
2050 2145 BSP 0912 01APR 01AUG 0801083 10000007 144 737
1400 9999 WK  0272 01AUG 31AUG 0801083 10000007 164 TU4
8888 1225 WK  0368 21AUG 01SEP 0801083 10000007 164 TU4
0925 2330 OHY 0753 01JUN 31AUG 0801083 10000007 174 320
```

Figure 4.14. Sample records from the input data file for the simulation program. It contains the processed records of the raw data file.

1. Initialization
2. Model run
3. Report generation

4.3.1 Initialization

In the first stage of the program, the simulation is initialized by a procedure call which invokes all the system objects to prepare themselves for the simulation run. The objects have their constructor methods (procedures), by means of which, they are introduced to the system. All data structures are initialized and first input record is read from the data file by a procedure call. The form of the input data file is shown in Figure 4.14.

After the construction of the system objects, the links between related sector objects are set by their own methods, so that the physical terminal airspace layout is constructed logically within the program. The links guide the aircrafts to navigate through the airspace. These are the permanent objects that are used in the program, and they are allocated in the memory till the termination of the program.

In contrast, some of the system objects are created when they are needed, and destroyed when they are no longer needed in the simulation. The Turbo Pascal programming language allows this type of *Dynamic Memory Allocation*

of objects, providing memory saving. In Turbo Pascal, pointers, that are the variables that contain the memory address of a data type (e.g., object type in this case), are allocated by the `New` command and de-allocated by the `Dispose` command. Furthermore, linked lists of dynamically allocated objects can be created by connecting objects to other objects by pointers contained in each object. The aircraft object is the one of such object types. Just before adding a new aircraft object, memory is allocated for the object; after removing the object, the memory associated with the object is disposed of. Thus many thousands of such objects can enter and leave the simulation as long as only some subset of them is present at a given time. This is the exact situation for the aircraft objects in an airport system.

When the initialization process are finished, the program is ready to run with the specified input data file. The execution of the simulation is performed in the next phase of the program, by iteratively scheduling and executing the pre-determined events of the system.

4.3.2 Model Run

The second phase of the program is the main part and most time consuming stage of the simulation.

The modeling philosophy of this study was that aircraft were not treated dynamically by calculating position coordinates or velocities each step through the system, as was done by some of the previous studies. Rather, aircrafts were "flown" through the system by following the links connecting sectors. The system as perceived by the simulation model is an abstraction of the real system.

The program is initialized, as described in the preceding subsection, by reading a record from the input data file. This record contains data for an aircraft that arrives to the terminal airspace. Therefore, the simulation *future event list* has only one future event, which is an arrival to the initial approach sector at a specified time, immediately after the program start. In fact, in

general, each new aircraft that is introduced to the program, causes a data read and hence scheduling of new arrival event (*aircraft regeneration*); without regarding this aircraft arrives to or depart from the airport. The actual model run begins with executing the initial event and as a result of the execution, at least a new event is scheduled. We note here that, some events trigger more than one event.

The aircraft arriving a sector flies into the sector and arrives to the next sector and hence, directly schedules to an "arrival to next sector" event, if the sector is empty and the aircraft satisfies the spacing enforced by the sector controller. Conversely, if at least one of the preceding conditions does not hold, then the aircraft is not allowed to pass to the next sector, as far as a departure from the same sector takes place.

The complete execution of the model is as follows:

The program calls the TIMEADVANCE subroutine to advance the simulation clock to the next event's time of occurence. The TIMEADVANCE subroutine request the "next-event" from the future event list, that is, CALENDER which is a variable of EVENTLIST object type in our program. Then the CLOCK object is invoked for increment its time value, and hence the simulation clock. If the simulation time does not exceed the ending-time, ENDTIME, the EXECUTEEVENT procedure is called to execute the next event.

EXECUTEEVENT causes control to jump to a subroutine, regarding the type of the event, which is a "bound event". (Figure 4.15 illustrates the form of a bound event.) The appropriate subroutine updates the related SECTOROBJECT object(s) and the aircraft variable, which is of the type of ACOBJECT. By the end of the execution of the bound event, program checks all the "conditional events". (In Figure 4.16, a typical conditional event execution is shown.) Each conditional event checks a sector and its queue to see whether the last bound event execution arises the need for a new event scheduling. This action is performed by calling each sector's CHECKQUEUE method. If the method returns an affirmative answer, then a new bound event is inserted into the CALENDER with respect to its occurence time, by this object's ADDEVENT

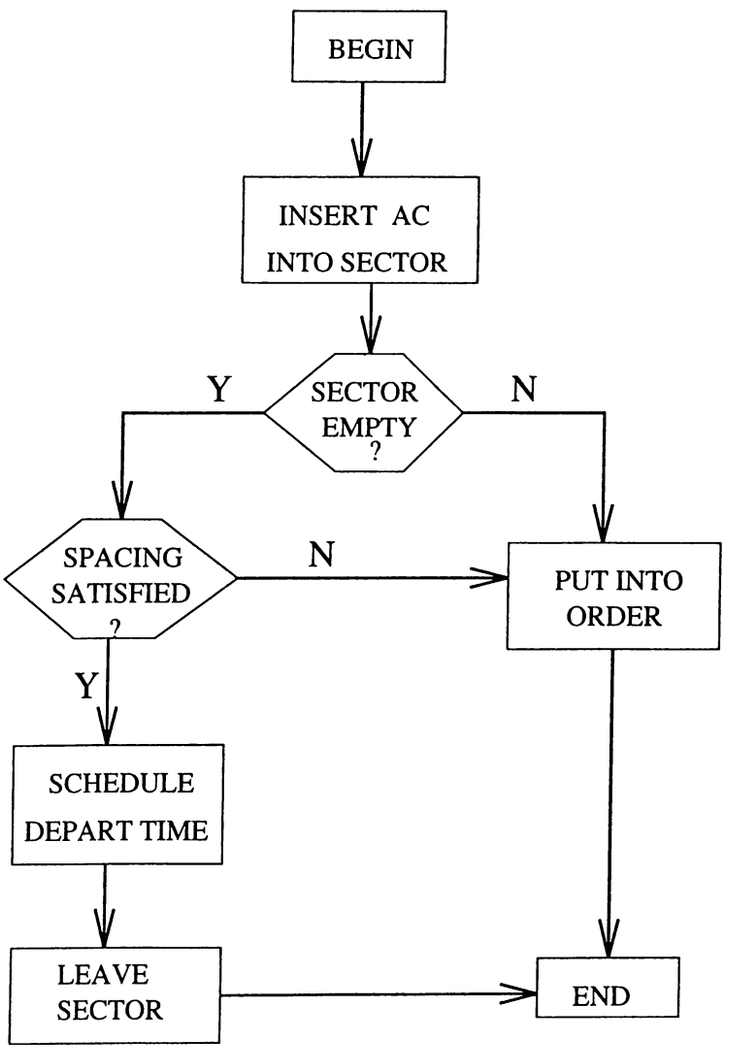

Figure 4.15. A bound event flow diagram

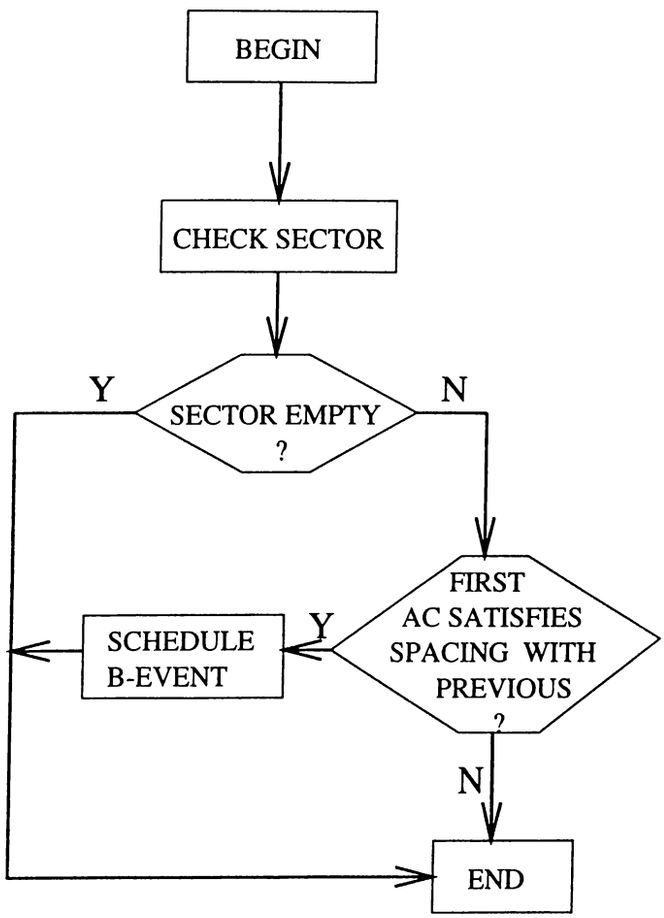

Figure 4.16. A conditional event flow diagram

method. And this completes a cycle of program execution.

Meanwhile, for debugging purposes, in the program, there is an option to trace the future event list, by invoking the CALENDER object through its PRINTLIST method. If the flag is on, which shows the user wants to trace the event list, the current content of the CALENDER is printed into the output file, in each EXECUTEEVENT pass.

The program exits the main execution loop and goes to reporting phase, at the time of one of the conditions below occurs, which comes earlier;

- If there are no records in the input file, that is, last input line is empty.
- If the next event, which is going to be executed, has event time exceeding the ending time of the simulation, ENDTIME, which is read from the input file.
- Any time an error occurs (e.g., File input error).

There are some points left which need to be highlighted about our program, for the sake of completeness:

In our program, we treat the apron as a sector, since the same properties hold for it as the sectors. However, apron has no separation minima to provide spacing between aircrafts in it, as opposite to the other sectors. Moreover, its departure procedure is not the same. Because its queuing strategy is not on a first-come first-served basis. The departing aircrafts are queued up to use runway in the order of their departing times; without regarding their arrival times to the apron. Furthermore, some of the arriving aircrafts do not depart the apron in the same day. Because of these reasons, apron has a "private" method to re-sort the arriving aircrafts. The *Insertion Sort* algorithm was used for this sorting process, since its appropriateness for the situation. We note here that apron, departure queue is already in the correct order, at any time of the simulation. It is known that, insertion sort is an excellent method whenever a list is nearly in the correct order and furthermore, its analysis shows that

its number of comparisons to sort a list is very few with respect to the other sorting algorithms.

Finally, an aircraft departing the airport schedules an "exiting event", which is also a bound event in the sense that it can be thought as an arrival to next sector (but, a null sector) event, when it reaches the initial approach sector. The exiting event invokes this aircraft object to dispose of its memory allocation. Before the destruction of the aircraft, cumulative delay time and name of the aircraft is collected by the DELAYOBJECT object for statistical purposes. These data is stored in a linked list in the order of arrival times of the exiting aircrafts.

4.3.3 Report Generation

When the program reaches the end, by any one of the reasons listed in the preceding subsection, the last action is to generate the reports of the simulation run. This is performed by a REPORTS subroutine call.

The program is designed to evaluate the operational performance of an air traffic control system. The delays in the sectors, idle times of the sectors, and maximum queue lengths in the sectors are examples of "measure of performance" among various ones.

In our program delays may be experienced in a holding stack on arrival, on taxiways when waiting for departure due to congestion. In fact, all the delays that aircrafts are faced in our program is caused by the followings:

- due to congested air traffic and spacing enforced on this traffic,
- runway non availability.

Delay times are identified according to their point of occurence in the airspace and additionally by their occurence hour. Similarly, the number of arrivals and departures for the sectors are identified separately, for the whole

item	represents
MAXWAITED	waiting time of aircraft which has longest delay
MAXQUELENGTH	number of aircrafts in queue waiting to proceed
TOTALWAIT	sum of passed aircraft's delays in sector
AVERAGEWAIT	average of TOTALWAIT per aircraft
WAITEDAC	number of aircrafts which have waited in sector
ENTEREDAC	number of aircrafts which enters into sector
EXITEDAC	number of aircrafts which departs from sector
IDLE TIME	total times that sector is empty
SEPARATION MIN.	spacing enforced in sector in minutes (input)
PASS TIME	flying time of sector when it's empty (input)
AVE. AC SYS.(L)	time averaged number of aircrafts in sector
AVE.TIME SPEND (w)	sum of AVERAGEWAIT and PASS TIME
ARRIVAL RATE (l)	number of aircraft arrivals per minute

Table 4.2. Descriptions of data fields of sectors in the output file, consisting of the statistics that are collected throughout the duration of the simulation run.

simulation period and for each one-hour sub-period within this period. Thus the areas of the airfield having congested traffic movements (hence ideas about the reasons of delays in the air-space) may readily be identified.

The simulation output consists of two parts:

1. Summary of output results, such as measures of performance of the model.

2. Brief information about simulation environment, such as data files.

A sample output file content is given in Figure 4.17 and Figure 4.18. Table 4.2 gives the descriptions of the fields in the output file shown in Figure 4.17. The data shown in Figure 4.18 consists of the hourly counts of the cumulative delay times in sectors and number of arrivals to this sectors in the same one-hour intervals. In other words, they are the hourly-divided details of the TOTALWAIT and ENTEREDAC fields shown in Figure 4.17, which have been calculated over the whole simulation period.

```
                        ARRIVAL              DEPARTURE
                    INI_APP   FIN_APP    FIN_APP   INI_APP
                    -------   -------    -------   -------
MAX_WAITED           00:14     00:01      00:09     00:00
MAX_QUE_LENGTH         6         1          4         0
TOTAL_WAIT           12:02     00:33      05:57     00:00
AVERAGE_WAIT         00:02     00:00      00:01     00:00
WAITED_AC             155        55        149         0
ENTERED_AC            256       256        271       271
EXITED_AC             256       256        271       270

IDLE TIME            01:35     14:56      13:26     01:51
SEPARATION MIN.    00:03.0   00:01.5    00:01.5   00:02.0
PASS TIME          00:25.0   00:02.0    00:01.4   00:22.0

AVE. AC SYS.(L)      4.95      0.38       0.62      4.13
AVE.TIME SPEND (w)  27.82      2.13       3.32     22.00
ARRIVAL RATE  (l)    0.18      0.18       0.19      0.19
 l*w                 4.95       0.3       0.62      4.13

MAXIMUM DELAY TIME  of PASSED ACs     : 00:18.0

  *  1 RUNWAY IN USE
  *  SIMULATION RUN-TIME 24:00 Hrs.

TOTAL AIRCRAFT READ FROM INPUT FILE  :    449

   INPUT  FILE  IS    <D:\ist5aug.dat>
   OUTPUT FILE  IS    <C:\log_ist.dat>
   CURRENT DATE IS      25/11/1994

     ***  RUNNING TIME (in seconds) :  8.57
```

Figure 4.17. The whole content of output file while TRACE option flag is off (continued on the next page).

THE HOURLY DELAYS AND NUMBER OF AIRCRAFTS FOR SECTORS

HOUR	INI-APP-ARR DELAY	INI-APP-ARR COUNT	FIN-APP-ARR DELAY	FIN-APP-ARR COUNT	FIN-APP-DEP DELAY	FIN-APP-DEP COUNT	INI-APP-DEP DELAY	INI-APP-DEP COUNT
0:00	6.0	6	0.0	5	0.5	2	0.0	2
1:00	2.0	5	0.5	5	0.0	4	0.0	4
2:00	0.0	1	0.0	1	0.5	5	0.0	5
3:00	1.0	6	0.5	5	1.0	9	0.0	8
4:00	16.0	11	1.0	9	2.5	13	0.0	14
5:00	17.0	14	0.5	10	17.5	14	0.0	13
6:00	30.0	6	2.0	12	28.5	19	0.0	19
7:00	2.0	11	0.5	6	4.0	16	0.0	16
8:00	26.0	12	0.5	13	3.5	9	0.0	9
9:00	14.0	9	2.5	11	13.0	13	0.0	14
10:00	12.0	15	2.0	11	9.0	13	0.0	12
11:00	120.0	18	2.5	18	22.5	13	0.0	13
12:00	126.0	21	4.0	19	43.5	18	0.0	18
13:00	111.0	14	1.5	16	51.5	19	0.0	18
14:00	42.0	16	0.0	19	62.5	20	0.0	18
15:00	12.0	10	2.5	8	58.0	19	0.0	23
16:00	13.0	9	3.0	11	12.5	18	0.0	18
17:00	23.0	11	3.0	11	2.0	10	0.0	9
18:00	29.0	17	2.0	14	7.5	10	0.0	11
19:00	57.0	11	1.0	15	3.0	6	0.0	6
20:00	6.0	9	2.5	9	3.0	8	0.0	8
21:00	13.0	14	0.5	10	10.0	8	0.0	8
22:00	34.0	5	0.5	11	1.5	2	0.0	2
23:00	10.0	6	0.5	7	0.0	3	0.0	3
TOTAL	722.0	256	33.5	256	357.5	271	0.0	271

Figure 4.18. The whole content of output file while TRACE option flag is off(continued from the previous page).

```
** TIME : 05:02.5          ** Number of events: 5
      E.TIME     AIRCRAFT     EVENT      SECTOR
      -------   ----------   --------  ----------
      05:03.5   734 IL 2451  Leaving   Initial-App
      05:04.0   313 TK 4690  Landing   Apron
      10:15.0   743 SQ 0404  Arriving  Final-App
      14:15.0   312 TK 3882  Arriving  Final-App
      22:50.0   733 TK 0484  Arriving  Final-App

** TIME : 05:03.5          ** Number of events: 5
    ** 734 IL 2451 has left the Terminal air-space.
      E.TIME     AIRCRAFT     EVENT      SECTOR
      -------   ----------   --------  ----------
      05:04.0   313 TK 4690  Landing   Apron
      10:15.0   743 SQ 0404  Arriving  Final-App
      14:15.0   312 TK 3882  Arriving  Final-App
      22:50.0   733 TK 0484  Arriving  Final-App
```

Figure 4.19. Sample lines from the output file while TRACE option flag is on (output by the command CALENDER.PRINTLIST).

The alternative output file content is obtained by including the trace option. If the TRACE option flag is on in the program run, then all the events that will be executed, are reflected as each one in a separate line, by traversing and printing the future event list. The only change in the output file is therefore, the addition of the lines such as shown in the figure 4.19.

4.4 Verification and Validation of the Program

The aim of our thesis is not to simulate a particular airport system, but to construct a model which is robust, reliable, flexible to handle the various cases, and also reusable to be easily modified for future use. We designed our model with these expectations. We used the most appropriate programming technique,

called object-oriented programming, for this purpose. We used a typical example system to model, which is the İstanbul Atatürk Airport system, since we could obtain its input data and compare the results.

4.4.1 Verification

The simulation model has been subjected to a number of tests for complete internal verification. Such verification has been accomplished to provide assurance that the model is operating as planned. The different sets of simulation input data have been run to verify the simulation results. We started these experimentations by taking simplest inputs, such as only one aircraft arrival and departure in a day. Then, we increment the input data size to check correctness of the results.

For the purpose of testing, fixed interarrival times were used to generate arrivals in the terminal area and to provide consistency of verification of all portions of the model. In the earlier runs, we chose a constant (and the same) separation minimum for each of the sectors, and we imposed greater aircraft interarrival time than this separation minimum. We decreased the interarrival time, while we fixed the separation minimum. In the former case, no delay were observed in any of the sectors because of the spacing; whereas, for the latter, we obtained growing delay statistics as we decrease the interarrival time more. Then the same process were repeated with a varying separation minima for the sectors. A similar positive growth were obtained.

Later, the configuration of the airspace was changed to simulate different airfield compositions, that can be used in other airport systems or can be applied to the same airport, but in the future. This was experienced to see the flexibility and versatility of the model.

4.4.2 Validation

In validating the simulation model, the objective is to determine whether the simulation model output matches the operation of its real life counterpart. For this purpose, the best strategy is to run the simulation program with a real input data set, that has been recorded in an airport system. We had some of the yearly flight plans of the airliners that use the İstanbul Atatürk Airport in these flights. We began with this data and simulated some randomly chosen summer days within the year; since the summer traffic is much more congested than any season in this airport. The model were applied to simulate the present conditions, that is the sectorization and runway usage policies, in that airport. However, the data files were containing more aircrafts than they should. This was because of that we use the files of the planned flights; and some of the flights were not realized. After that, one-day real input data was obtained from that airport air traffic control center and it is simulated.

The results of these runs are discussed with the experienced air traffic controllers including the air traffic chief controller of the Atatürk Airport. The controllers were satisfied by the generated delay outputs. However, since there was no serious study for the delays of the aircrafts using this airport, we did not have the opportunity to compare the results with realistic ones. The controllers approved the statistics for the delay time of the maximum waited aircraft in a day, idle times of a sector controller, hourly delay statistics, and partially on the number of waited aircrafts in a day, but no one of the averaged or cumulative results of the program can be commented on. Because, each one of the ATC staff were using his experiences, and this yielded subjective comparisons. Among these statistics, the hourly counts were easiest to guide us for the validation, since it is clearly known that the arrivals are congested around the 12:00 a.m. and departures are congested around the 8:00 a.m. (with respect to GMT, or the Greenwich Mean Time). The hourly-total-delays and hourly aircraft counts generated by our program reflects this situation.

In addition to these studies, a day-time observation study was made, in the summer season of İstanbul Atatürk Airport. In this study especially, the

departure queue length and departure delays of the aircrafts were recorded. However, this was incomplete and small amount of data for a comparison to generated program output, in order to validate it.

Chapter 5

Simulation Results and Interpretation

In this chapter we will give the results of the simulation program, whose design and coding phases have been described in the preceding chapters. Among the many parameters, two are chosen to illustrate the sensitivity of the model to the parameters. The two parameters are the most likely to be changed in the real model, because of their easy implementation. The results are obtained by plotting some of the performance measures of the system while these two parameters being assigned to certain logical values.

5.1 Results of the Simulation Program

The Airport System Simulation program has been implemented using the Turbo Pascal programming language (Version 6.0). The object-oriented programming capabilities of this language, which are introduced by version 5.5, were used to code the model. Program has been coded and run on a IBM compatible machine with Intel 80386 DX-40 processor, under the Microsoft disk operating system MS-DOS 6.2.

The sensitivity of the results to various spacing parameters that are enforced by air traffic controllers and sector passing times which are in fact, also determined by speed limits allowed by the controllers, is illustrated by a number of simulation results. In addition, the number of sectors, which has also

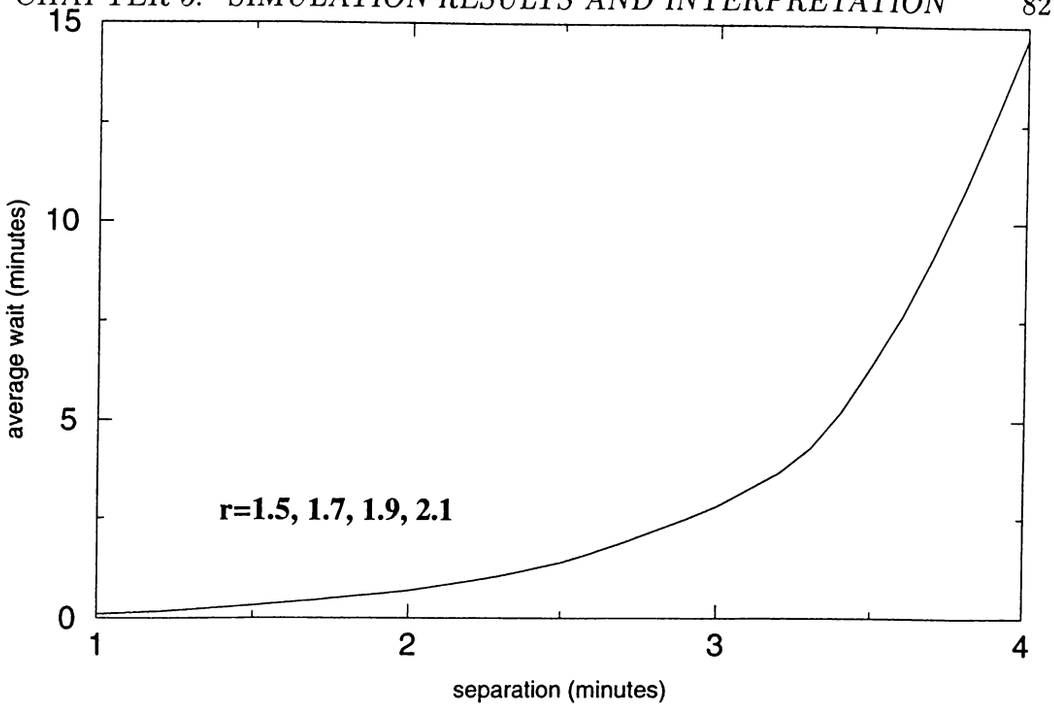

Figure 5.1. Average waiting times in the Initial Arrival Approach sector. In this figure and following ones, **r** indicates the separation minimum of the Final Approach Arrival sector.

parameter	Arrival Sectors		Departure Sectors	
	Ini. App.	Fin. App.	Fin. App.	Ini. App.
Spacing (min)	*varies*	*varies*	1.2	2.0
Pass Time (min)	25.0	1.3	1.0	22.0

Table 5.1. The input parameter values for the sample plottings.

influence on the delay times of the aircrafts flying in it, is changed to demonstrate the sensitivity of the model to a specific sectorization. This enhancement also showed the flexibility of the program to model various terminal airspace configurations.

The input data used in the following example runs is obtained from the İstanbul Atatürk airport, which is recorded on August 5, 1994. It is a typical summer-day 24-hour aircraft movement data. Total of 526 aircraft movements were recorded in that day: 256 arrivals were observed for the Initial Approach sector, and 270 departures were observed from the apron.

Table 5.1 lists typical values for the system parameters, which are also used

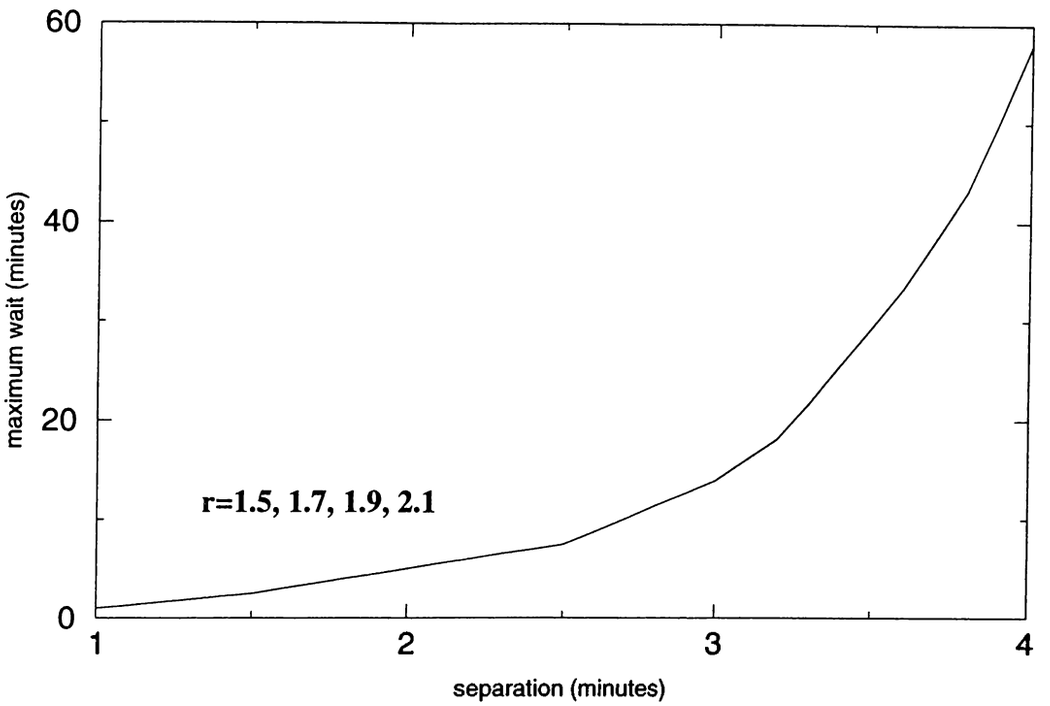

Figure 5.2. Maximum waiting times in the Initial Arrival Approach sector.

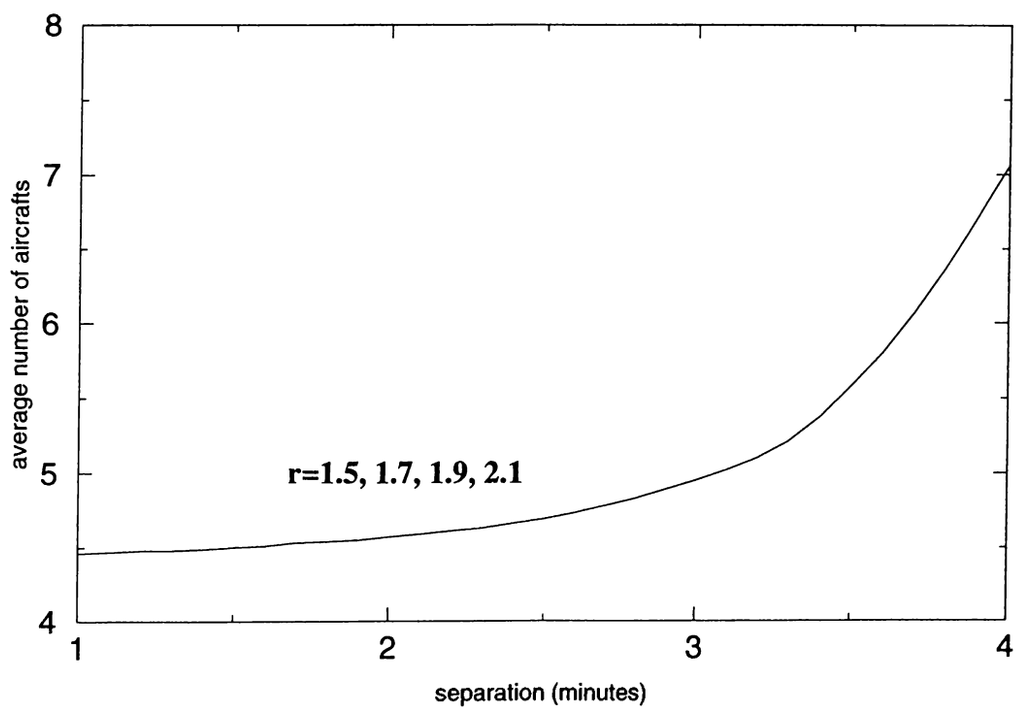

Figure 5.3. Average number of aircrafts in the arrival segment of the Initial Approach sector.

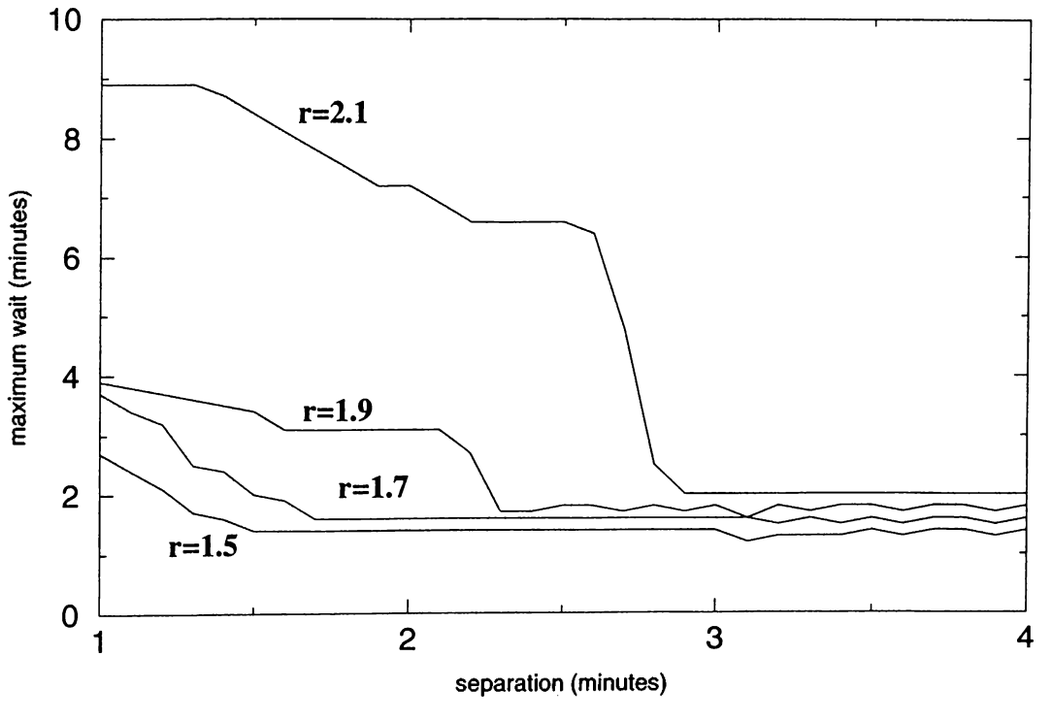

Figure 5.4. Maximum waiting times in the Final Arrival Approach sector.

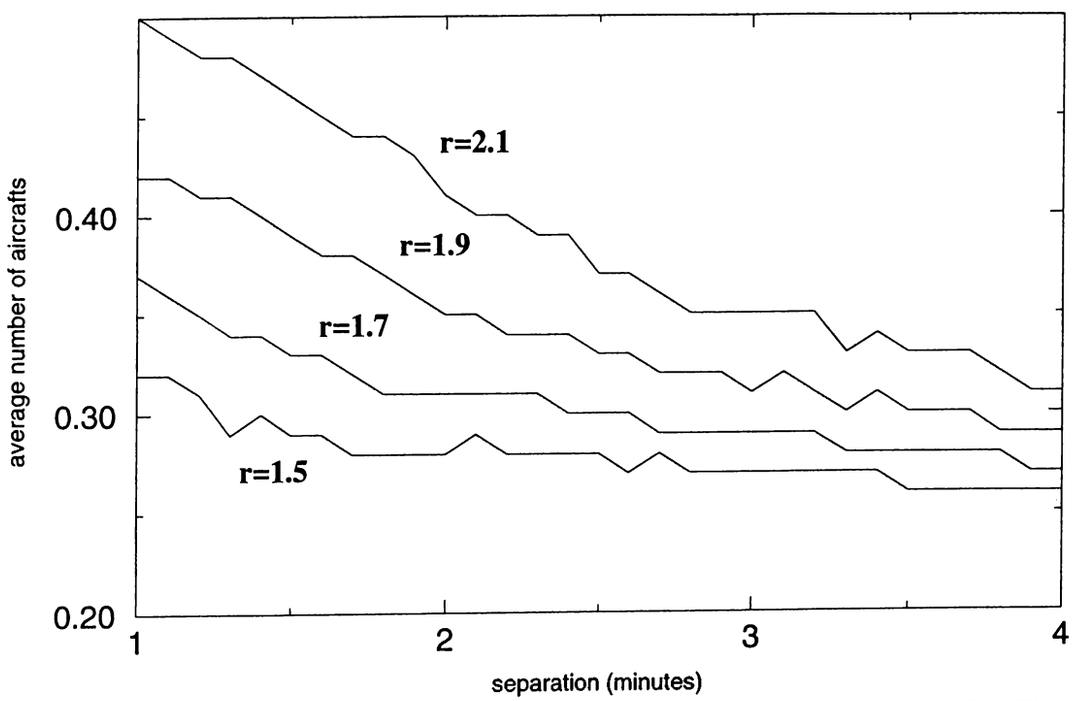

Figure 5.5. Average number of aircrafts in the Arrival segment of the Final approach sector.

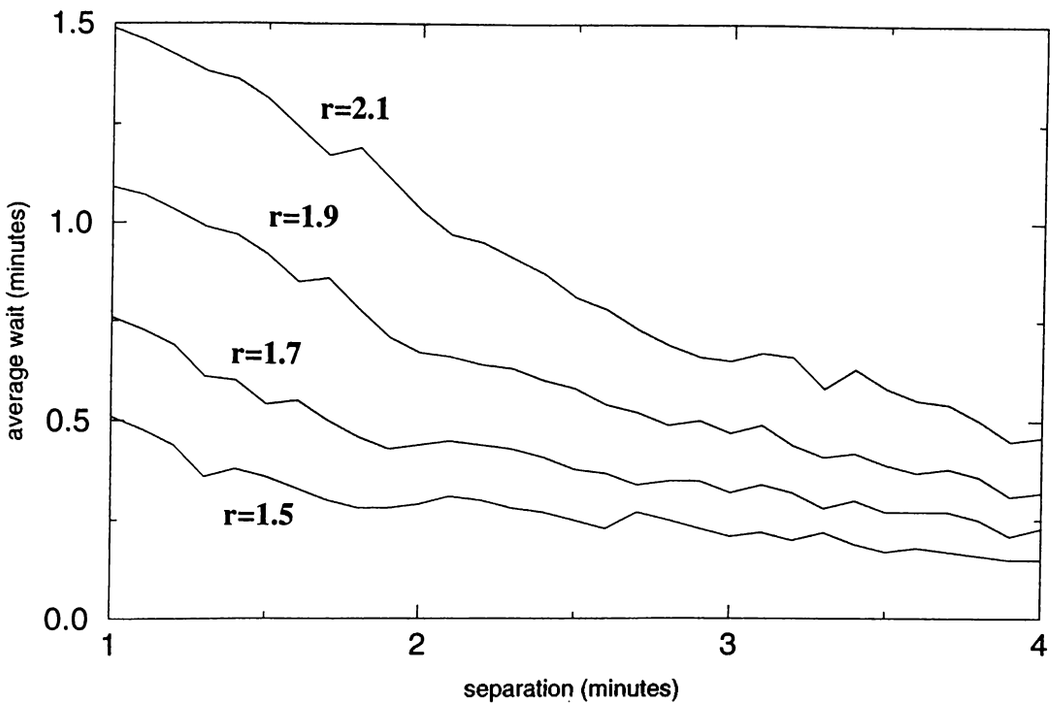

Figure 5.6. Average waiting times in the Final Arrival Approach sector.

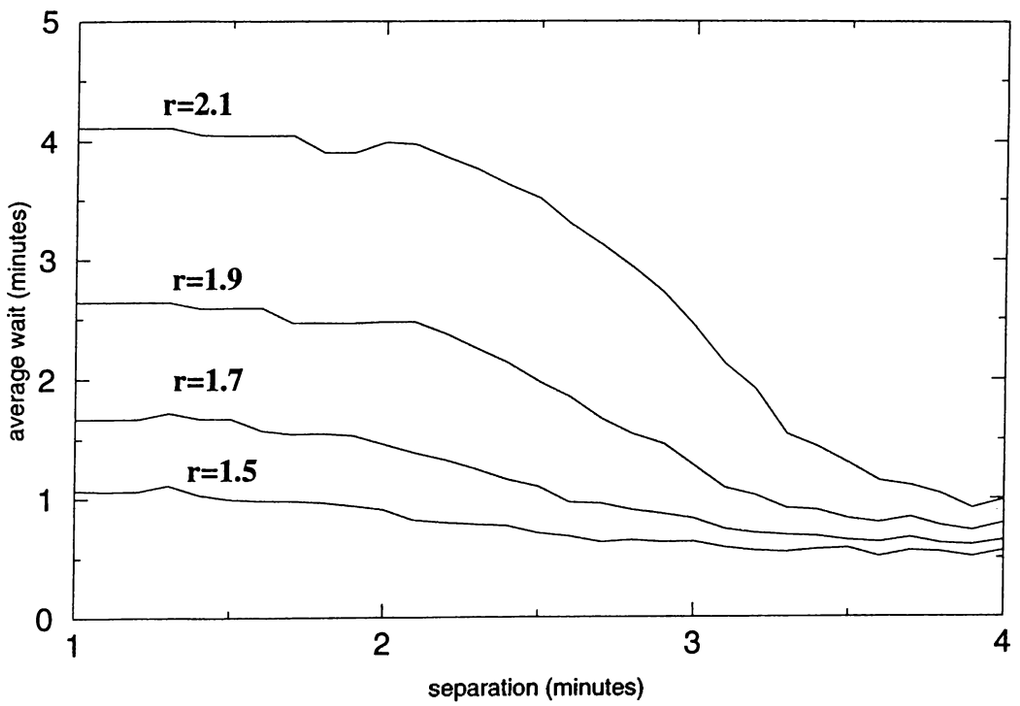

Figure 5.7. Average waiting time in the departure segment of the Final Approach sector.

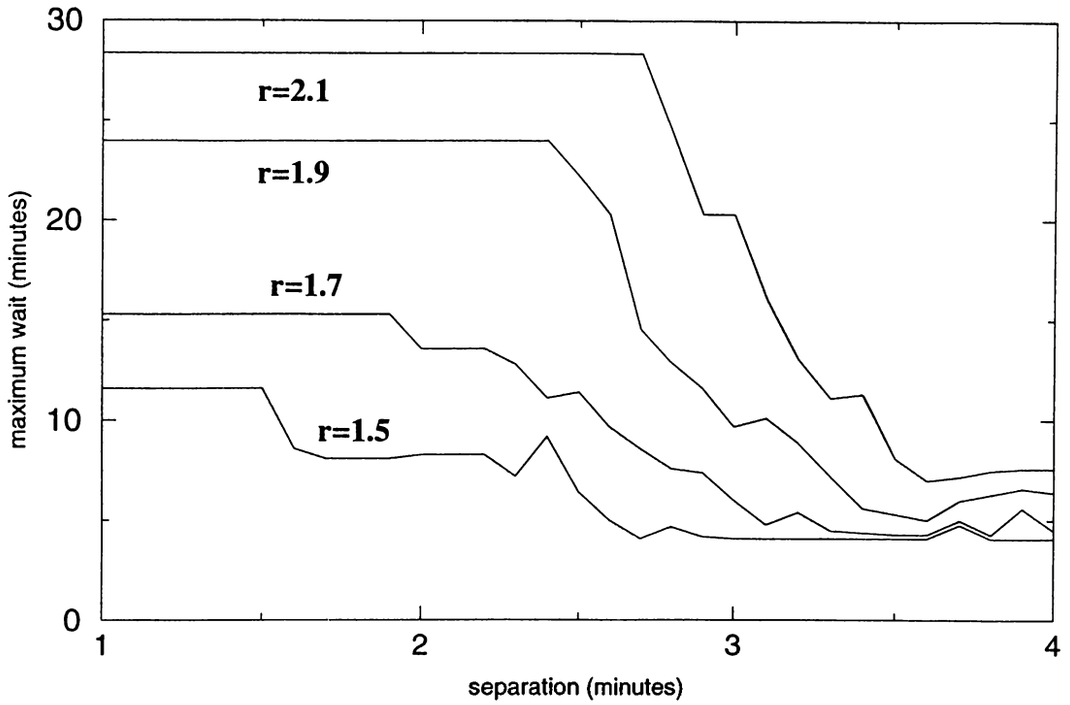

Figure 5.8. Maximum waiting time in the departure segment of the Final Approach sector.

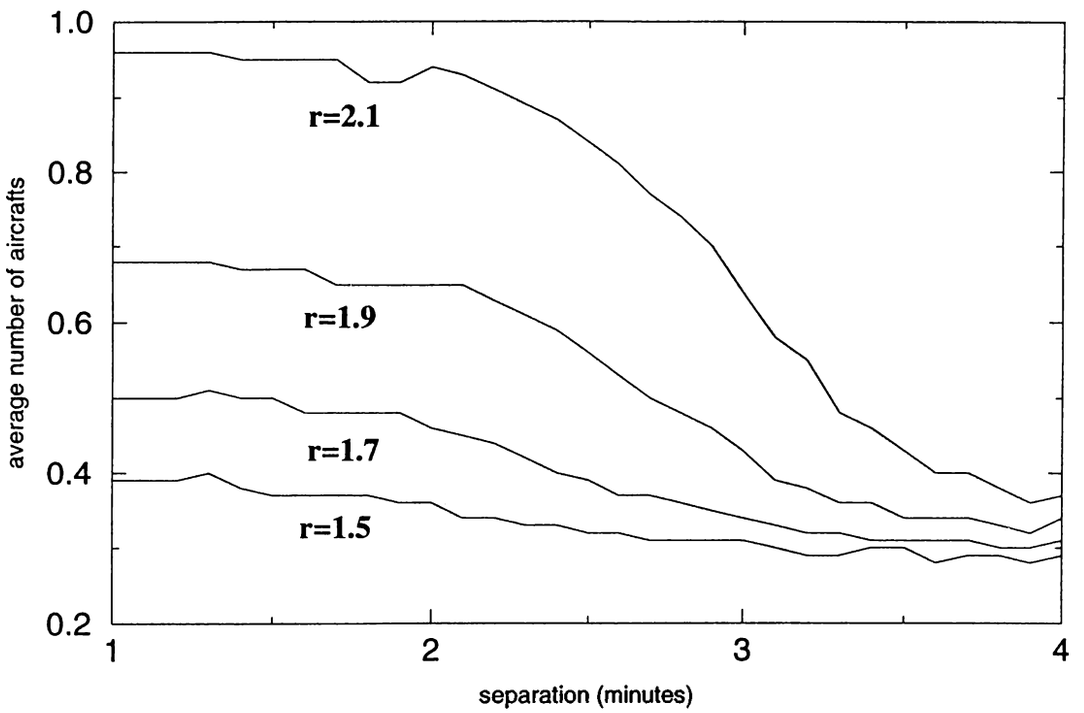

Figure 5.9. Average number of aircrafts in the departure segment of the Final approach sector.

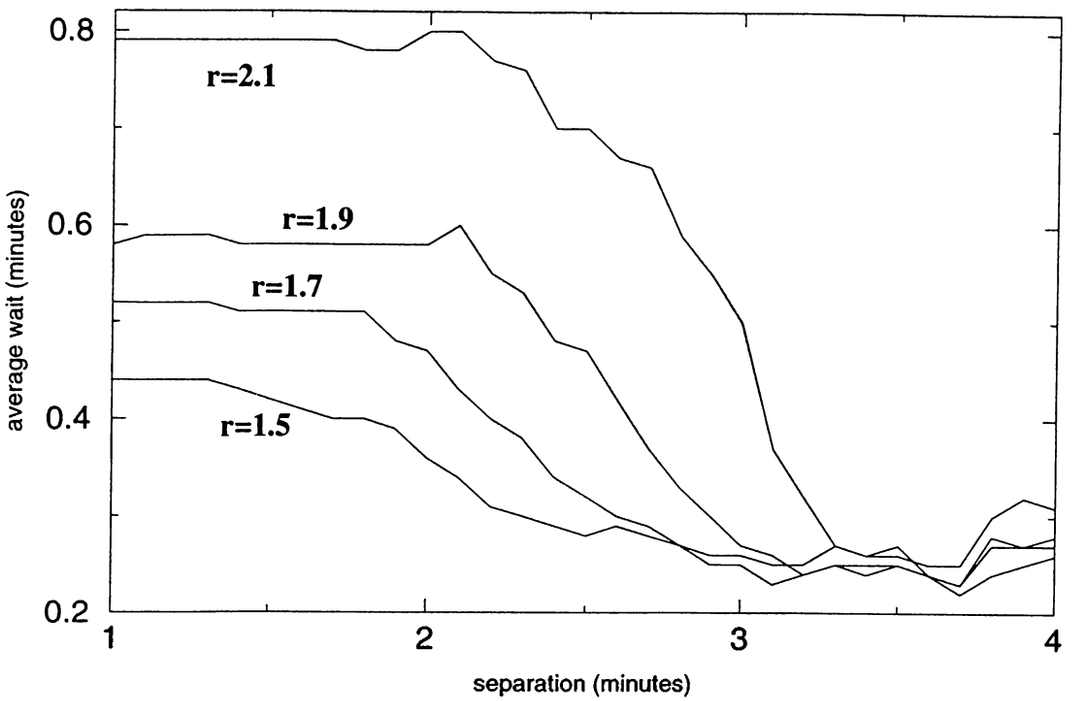

Figure 5.10. Average waiting times in the departure segment of the Initial Approach sector.

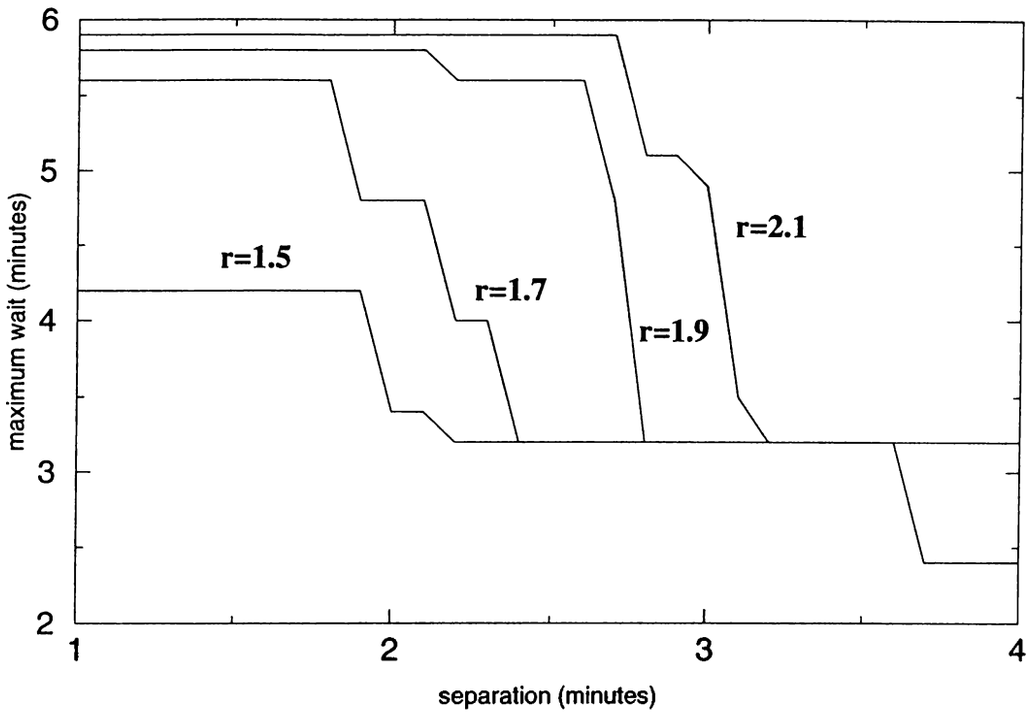

Figure 5.11. Maximum waiting times in the departure segment of the Initial Approach sector.

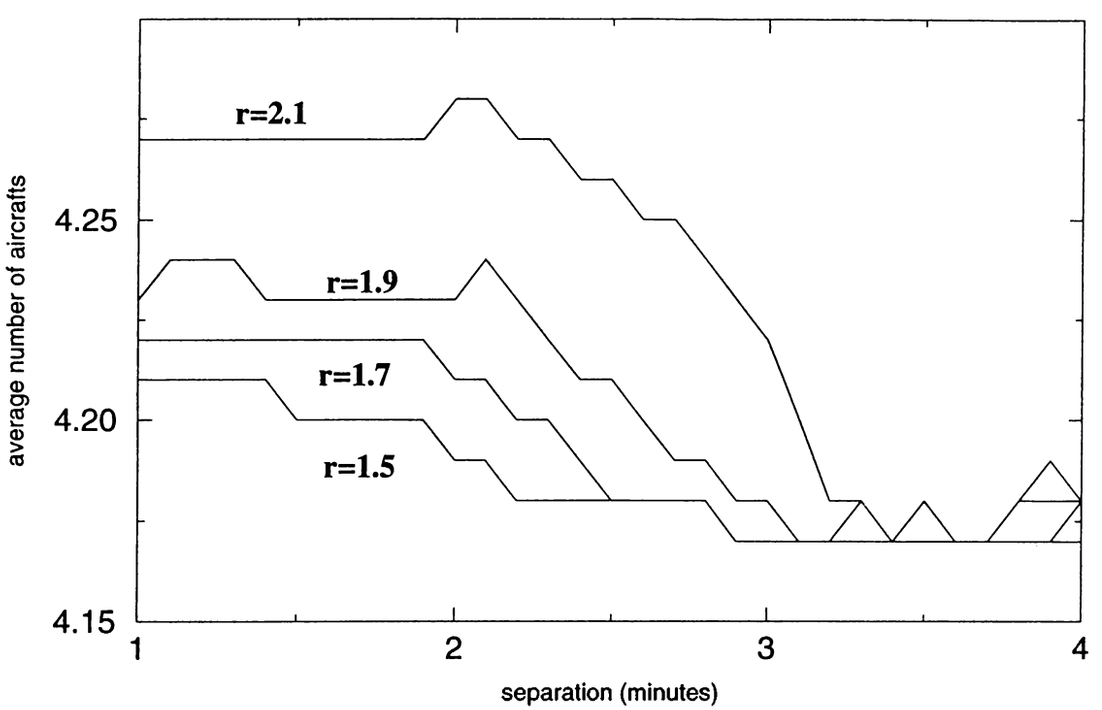

Figure 5.12. Average number of aircrafts in the departure segment of the Initial Approach sector.

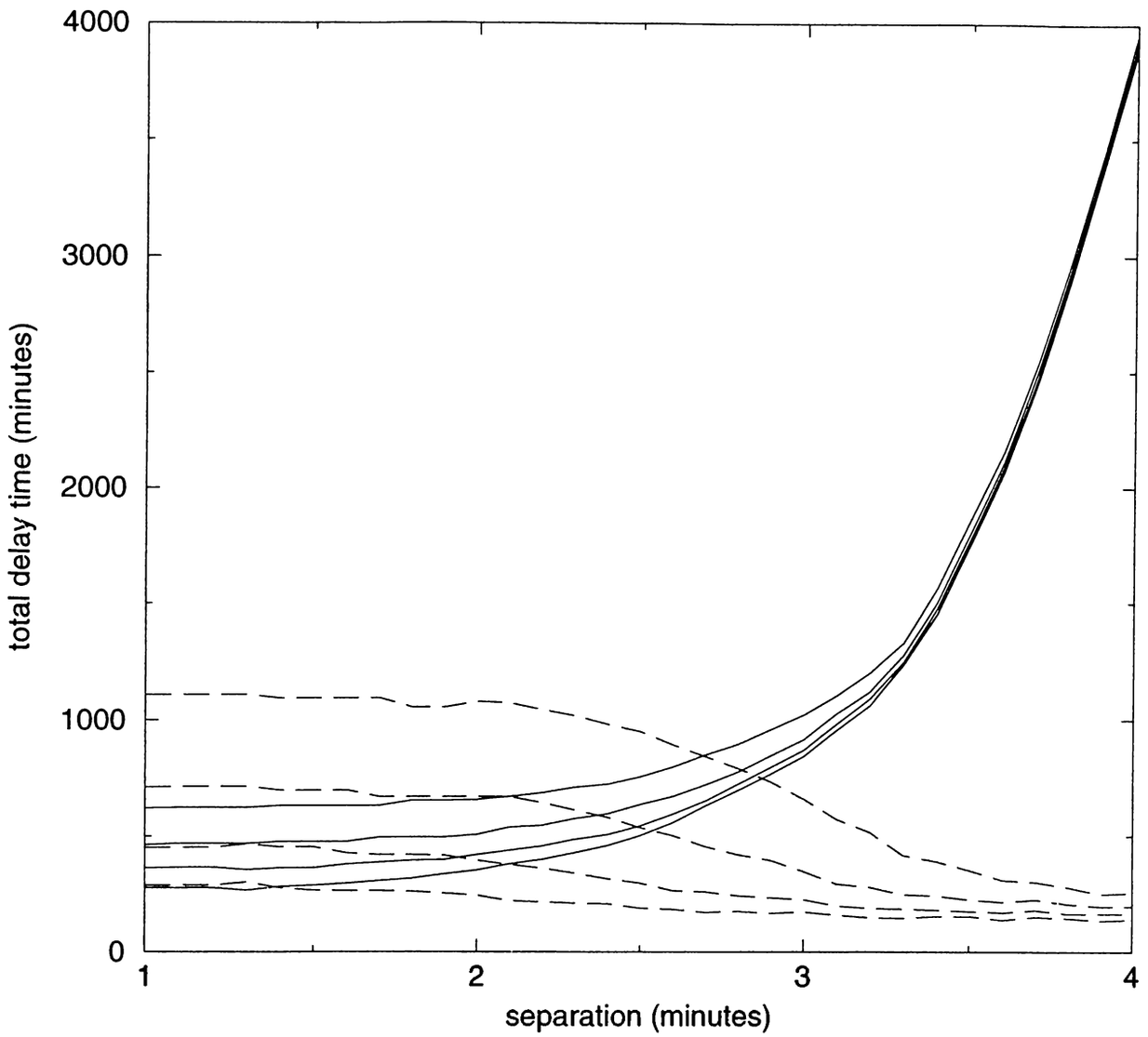

Figure 5.13. The resultant changes in the total air delay (shown as solid lines) and total ground delay (shown as the dashed lines), by the change of the Initial Approach Arrival separation minimum and for the four different Final Approach Arrival separation minima. Higher curves belong to larger r values.

for the sample runs of the model. In the table 5.1 the separation minima entries for the two arrival approach sectors are left empty since they are used as the test data, with various values. The required Initial Approach Arrival separation minimum is 3.0 minutes and Final Approach Arrival separation minimum is 2.0 minutes. But they are subject to change by the sector controller's directives.

The easiest way to increase a terminal airspace capacity is to decrease the separation minima enforced in the airspace. Because of the safety requirements, the separation minima for the arriving aircrafts are higher than the departing ones. However, the standards can be improved by using advanced air traffic control procedures. The departures are allowed with a minimal possible separation in between. The arrival separation values have effects on the succeeding sector queues. We have used the Initial Approach Arrival and the Final Approach Arrival separation minima to measure the sensitivity of each sector.

5.1.1 Performance Measure Outputs of the Program

The simulation program calculates most of the important performance measures of the terminal airspace system, such as, *maximum waited aircraft in sector, maximum queue length in sector, cumulative wait of the aircrafts in sector, average wait times, number of waited aircrafts, average number of aircrafts in sector, average time spend in sector* and finally, *idle times of sector controllers.*

The performance of the sectors, hence the overall system can also be represented by average values, in an efficient manner. Because of that reason, we have chosen the three performance measures among the above ones. For each sector, the three best describing ones are plotted against the Initial Approach Arrival separation minimum changes. And this plotting process is repeated for four different Final Approach Arrival separation minima.

We have plotted the following measures in the graphs:

- **average waiting time** of aircrafts in each sector, which is calculated as, total waiting time divided by number of aircrafts entered into the sector.
- **maximum waiting time**, which is the waiting time of the mostly delayed aircraft along the simulation run time.
- **average number of aircrafts in sector (L)** is the time-average number of aircrafts in the system; calculated by the formula:

$$L = \frac{1}{H} \int_0^H L(t)dt$$

where $L(t)$ is the number of aircrafts in the system at time t, and H is the simulation end-time.

In order to plot the graphs, the Initial Approach separation is assigned to the value of 1.0 minutes initially, which is an impossible value for the current configuration of the airport. Then we increased this value 0.1 minutes and calculate the three parameters for each of the four sectors. We repeat this process till it reached 4.0 minutes, which is likely to be an unacceptable high number. After that, we repeatedly plot the graphs of three parameters by different Final Approach separation values; 1.5, 1.7, 1.9 and 2.1 minutes.

5.2 Interpretation of the Simulation Results

In the simulation sample runs we analyzed the sensitivity of the results to the two parameters, namely, the separation minima which are enforced by the sector controllers of the Initial Approach Sector and the Final Approach Sector.

The airspace model does not have the capability to dynamically handle traffic in a flexible way. In the reality, air traffic control centers are more flexible and the order of landing may be modified and other methods used to reduce overall sequencing delays.

The delays of the aircraft traffic in the terminal airspace is categorized into two parts:

1. **delays due to the composition of the traffic:** this problem occurs at the aircraft generation points, which are the Initial Approach Arrival sector and the Final Approach Departure sector. However, it affects the statistics of the other sectors indirectly.

2. **delays due to runway non-availability:** this problem affects only the two Final Approach sectors.

From the graphs of the preceding subsection, we see that, all of the three measures of the last three sectors, are decreasing by the increment of the Initial Approach separation minimum, this is an improvement for these sectors. However, the first sector for the arriving aircrafts, the Initial Approach itself, behaves in the opposite direction: its performance is decreased quadratically, by the increments. Obviously, this sector is not affected by the changes of the other sectors' parameters. Therefore it is insensitive to the changes in the Final Approach separation minimum. (Figures 5.1, 5.2 and 5.3) For the Initial Approach sector, the two waiting time curves are increase dramatically, as separation value approaches to 4 minutes. (Figures 5.1 and 5.2)

It is clearly understood that more spacing enforced in one sector yields, smaller waiting times (because of separation) in the following sectors.

5.2.1 The Initial Approach Arrival Sector

The Initial Approach sector is the first sector as to come into the terminal airspace. Therefore, it is the first point of enforcing separation. The separation value used here, effects all other sectors of the airspace directly or indirectly. The delay statistics of all sectors are reciprocal to the Initial Approach separation. The statistics of this sector are not sensitive to the parameters of the other sectors. By this fact, it can be seen in Figures 5.1, 5.2 and 5.3 that the curves are the same while the Final Approach separation is changed. Figures also shows that, the all statistics are sharply increase after 3.0 minutes separation. This is the usual separation enforced in the terminal airspaces.

In fact, the most of the delays that are observed in the terminal airspace, are caused in this sector; by the congested air traffic. The separation of this sector is the dominant factor in the whole delays of the airport system. Therefore, the separation must be analyzed carefully in this sector.

5.2.2 The Final Approach Arrival Sector

The Final Approach Arrival sector is very sensitive to the Initial Approach Arrival separation minimum. As the Initial Approach separation is incremented, the delays in this sector shrink. Obviously, the situation reverses when the sector's own separation is increased. If the sector's spacing is greater than that of the Initial Approach, the delays are very big. When this situation changes, the delays decrease immediately. Especially, for the maximum waiting time, this decrease is very sharp.

When 2.1 minute separation is enforced in this sector, maximum waiting time value is doubled, until the separation in the Initial Approach reaches 2.6 minutes. After 2.8 minute separation all the curves level out. The correspondent graphs are given in Figures 5.4, 5.5 and 5.6.

5.2.3 The Final Approach Departure Sector

The Final Approach Departure sector is the first sector for the departing aircrafts. It is affected by the traffic in the Final Approach Arrival sector. Therefore, its parameters are very sensitive to the parameter changes of that sector. We see that, average wait and number of aircrafts in this sector level out until the Initial Approach separation minimum is 2.0 minutes; then decreases sharply until 3.0 minutes. Maximum waiting time behaves in the same way for the separation values 2.5 and 3.5 minutes. These situations are illustrated in Figures 5.7, 5.8 and 5.9.

5.2.4 The Initial Approach Departure Sector

The Initial Approach Departure sector is the sector where the smallest portion of the airspace delays are observed. It is the most insensitive sector among all sectors. This is because of the fact that, it is the last sector of the airspace and aircrafts are already in the well-order when they arrive into this sector. Since the Final Approach sectors have separation smaller than 2.0 minutes, they are directly reflects the air traffic congestion to this sector if the separation enforced in the first sector is less than 2.0 minutes. After that, the delay statistics are decreased with the increasing separation. For the separation values greater than 3.2 minutes this sector has very small and constant delay statistics. (Figures 5.10, 5.11 and 5.12.)

When we look at the hourly-delay statistics and arrivals, we notice the changes in the delay curves. As for arriving flights, total delay in the Initial Approach Arrival sector is considerably higher around the 12:00 p.m. GMT. These delays are resulted by the congested traffic arriving to the terminal airspace. The traffics coming from different directions meet at the air-route gate in this sector. The sector controller separate the traffics, orders them a line to the Final Approach sector. When the Final Approach separation minimum is greater than that of the Initial Approach, a further separation is required for the traffic by the Final Approach sector controller. This results a second waiting period for the aircrafts. But in the normal situation this is not the case, i.e., the separation minima are decreases as coming in.

This congested traffic also effects the departing aircrafts, since the arriving aircrafts have the precedence to use the runway. This situation results an increase in the delays of the Final Approach sector. However, the policy is changed to allow departures when the departure queue length is greater than the arrival queue. Therefore, the delays do not increase much any more.

Around the 7:00 a.m. GMT, the departures are congested at the airport. The delays increase at that time period. Since the separation minimum of Final Approach Departure sector is smaller than that of the next sector; in the

Initial Approach Departure sector, aircrafts are again separated. Because of the runway non-availability at that time period, the arriving aircrafts in the Final Approach sector are enforced to wait the aircraft using the runway.

To interpret the delay statistics and compare the alternative operation policies, we need a pre-processing on the delays. Conversion of the total delays to operations costs and the inclusion of the value of passenger or cargo time provide a convenient bases for the comparison of the alternatives. The air-delays have more importance than the ground-delays. Because, the air-delays cost more, and they may yield emergency cases, then the cost value can be accepted as infinite. In general, the cost of ground-delay is accepted as 75 % of that of air-delay [10]. Figure 5.13 shows the total delay times of the airspace in a simulated day, as total ground delays (delays in the Final Approach Departure sector) and air-delays (delays in all other sectors). (The total ground-delays curves are the same as the curves in the Final Approach Departure sector "average wait" graph. But the scale is multiplied by the number of aircrafts entered into this sector.) A separation minimum greater than 3.0 minutes yields a considerable delay for the system. Moreover, since in that case, the air-delays are very high; the crash probability is high. The safety requirements are violated in such a situation.

Chapter 6

Conclusion

A set of classes has been presented that has improved the efficiency of writing simulation programs. We have shown the major advantages of a modern, object-oriented language to write simulation programs. We have used the object classes, to illustrate different but similar airport scenarios. The ease of developing Turbo Pascal units for standard classes, which can be used as building blocks for further simulations has been shown.

Objects are the natural way to describe many of the entities in a terminal airspace air traffic control system simulation model.

Through the simulation of a terminal air-space, the ability to write discrete event simulations with an object-oriented language has been demonstrated. It has been used to implement a prototype model for the air traffic control system. Consequently, the feasibility and advantages of using this object-oriented language has been shown. However, the current implementation of the language does not have an adequate associated standard library with which to write systems applications. In addition, it does not support independent compilation of modules, which is necessary for team projects and software development. In these regards, a more advanced implementation, can be seen as more promising language than this one.

Overall, programming with Turbo Pascal seems less error-prone than with

other simulation languages due to its strong type checking, dynamic allocation of objects, small size, and efficient compilation. But the most important feature of the Pascal is its readability and its not-so-steep learning curve among similar implementations.

Our program has a set of system objects that are coded to model the real-world entities of the simulated system. These objects, as well as the other objects that are used to represent data types for the general simulation applications (e.g., simulation clock and future event list) are designed in a flexible way to enable extensions and reusability. New system objects can be added to the simulation by writing a small amount of code. The methods (behaviors) of the system objects can be changed to implement different scenarios, without changing the declaration of the object type; since the methods are declared as virtual functions in the program to allow late-binding.

The results of the program indicate that discrete event modeling of system can adequately simulate air terminal operations. Many decisions concerning the system can be made with assistance of such model. The model is capable of providing results for a wide range of terminal ATC problems, such as the evaluation of terminal ATC system capacity, and the evaluation of design parameters for future equipment.

Additionally, we have described in this thesis, sectorization of air-spaces which is a hard problem in ATC. We evaluate the alternative resectorization strategies and then calculated the workloads of the controllers of this sectors, so that a comparison can be made in order to decide new sectorization designs. The alternative sectorizations are implemented by minor changes in the type declarations. We have coded the following configurations:

- *Four-sector ATC system:* that configuration is made up of arrival and departure segments of the Initial Approach and Final Approach sectors. It is the most general ATC system structure.

- *Two-sector ATC system:* that version has only the Final Approach sectors for arrivals and departures. It is applied to small airports.

- *Six-sector ATC system:* that version is similar to the first one physically, except it further divides the Initial Approach sectors into two to enable aircrafts to come in from two air-route gates, and go out from other two. It is designed for congested airports, it uses more air traffic controllers but decrease their workloads.

We have used the first configuration to implement the İstanbul Atatürk airport system for simulation. That is the exact case for this airport. The six-sector configuration is studied to apply for this airport in the near future.

The results obtained from the simulation model would enable the ATC planners to identify the parts of the system that are restricting efficiency and capacity. Despite the model's capability, further study is indicated to make the model more realistic.The extensions of the model are visualized as follows:

1. the modeling of flight dynamics of aircraft is not viewed as necessary to answer the questions concerning terminal operation. However, the model can be extended to include a computational procedure to update aircraft position in small time increments. This extension could result in a combination of the discrete events philosophy with the continuous system simulation, adding the power of calculating air crash probabilities in the terminal airspace.

2. the weather effect on the operation of the terminal ATC system can be extended; and other effects on the system performance, such as, human factors and wave-off probabilities (i.e.,probability of not landing of an aircraft in its turn), can be included.

3. airport terminal facilities could be modeled in more detail to include *speed exits, taxiways, turn-offs* and *gates* with characteristics of aircrafts, using those facilities.

In addition, several improvements of the existing systems can be studied to cope with the future surge in air travel. These improvements include *optimal*

- *Six-sector ATC system:* that version is similar to the first one physically, except it further divides the Initial Approach sectors into two to enable aircrafts to come in from two air-route gates, and go out from other two. It is designed for congested airports, it uses more air traffic controllers but decrease their workloads.

We have used the first configuration to implement the İstanbul Atatürk airport system for simulation. That is the exact case for this airport. The six-sector configuration is studied to apply for this airport in the near future.

The results obtained from the simulation model would enable the ATC planners to identify the parts of the system that are restricting efficiency and capacity. Despite the model's capability, further study is indicated to make the model more realistic.The extensions of the model are visualized as follows:

1. the modeling of flight dynamics of aircraft is not viewed as necessary to answer the questions concerning terminal operation. However, the model can be extended to include a computational procedure to update aircraft position in small time increments. This extension could result in a combination of the discrete events philosophy with the continuous system simulation, adding the power of calculating air crash probabilities in the terminal airspace.

2. the weather effect on the operation of the terminal ATC system can be extended; and other effects on the system performance, such as, human factors and wave-off probabilities (i.e.,probability of not landing of an aircraft in its turn), can be included.

3. airport terminal facilities could be modeled in more detail to include *speed exits, taxiways, turn-offs* and *gates* with characteristics of aircrafts, using those facilities.

In addition, several improvements of the existing systems can be studied to cope with the future surge in air travel. These improvements include *optimal*

sequence of aircrafts and *reduced passtimes* with improved equipments. All the improvements are aimed at reducing aircraft delays and increasing the terminal capacity.

Furthermore, better simulation models can be created by extending to include the following statistics:

- traffic distribution by sector, point, air-route and level band,
- distribution of workload at various control positions,
- composition of workloads,
- collision probabilities in the various parts of the airspace,

Although the preceding extensions are not included in the present simulation model, the capability of readily including these extensions by inheritance of the objects and overwriting the virtual methods, indicates the model's versatility.

We have used the Three-Phase Approach to implement our simulation. It is the most suitable world-view for us, because of its basic design. By this approach, the event routines are divided and written into smaller codes. This allows further changes in event routines easily, when it is needed. Although the Three-Phase Approach increases the running time of a program, it is negligible because of its decrement in code development time. Simulation projects are generally, much more costly in their development times. This was also the case in our study, and the program's running time is about 10 seconds in the average for a 24-hour simulation time [18].

To conclude the thesis, we again emphasize that, the object-oriented class declarations, which can be collected into a unit, to make up a class library, can be used for the future simulations, quickly and easily. For the future simulation implementations, object-oriented design is a promising paradigm; since it is very natural for *Distributed Simulations* and *Windows Programming.* In the former case, each of a simulation entities (as an object) can be assigned

to a processor in a multiprocessor environment. We see that the parallelism is often the case in the modeled systems. The latter case is an important point in simulation studies: graphical interfaces can be easily adapted into an object-oriented design, so that the simulation study can be animated. The quality of a program will be improved and the model building efforts will be reduced as simulation framework is equipped with a *user friendly* interface.

Finally, future success of object-orientation depends on programmer acceptance, the establishment of standards, availability of object libraries and object-oriented development environments.

Appendix A

Apsim.pas Program

DELAYOBJECT **object type**

`aclist` : Keeps the list of delay times of aircrafts which exits simulation.

`constructor Initialize`: Initializes the aclist.

`procedure Delaystats`: Collects delay records of aircrafts that exit simulation.

`procedure FindMaxDelay`: Finds and prints maximum delay time among exited aircrafts for statistical purposes (performs garbage collection).

`procedure Done`: Disposes memory allocation.

ACOBJECT **object type**

`delay` : Cumulated delays of aircraft at any time.

`creatt` : Simulation time of introducing this aircraft to the program.

`flight` : Flight type of aircraft (through or terminating).

`typeac` : Body type of aircraft (wide or narrow body).

`turAro` : Declared turn-around-time of aircraft.

`depTime`: Declared departure time from airport.

`ACname` : Name and sign of aircraft, including the flight number.

`constructor Initialize`: Initializes an aircraft variable.

`destructor ExitSim`: Disposes an aircraft variable.

AIRPORT **object type**

`wpark` : stores park counts for the wide aircraft types.

`npark` : stores park counts for the narrow aircraft types.

`procedure Starting`: Initializes Airport Parking counts.

`function Full`: Returns true, if there is no empty place for the next arriving aircrafts to the apron.

`procedure ParkAC`: Increments the counters according to aircraft type.

`procedure UnparkAC`: Decrements the counters according to aircraft type.

EVENTLIST object type

`eventcell` : A node of the Future Event List containing event type, event time and identity of the object that causes the event.

`countcell` : Stores current number of nodes in list for debugging purposes.

`constructor initialize`: Initializes the Future Event List. It is called at the time of initialization phase of the simulation.

`procedure Addevent`: Inserts new node (Event) to Future Event List.

`procedure Getevent`: Retrieves and returns the current top node in the Future Event List, which determines the nearest future event of the simulation.

`procedure Printlist`: Prints contents of the Future Event List for tracing and debugging purpose.

CLOCKOBJECT object type

`clockrec` : Stores the simulation clock in the *hour:minute* format.

`constructor Initialize` : Initializes the simulation clock in the beginning of the simulation.

`procedure Advancetime` : Increments the simulation clock.

`procedure Printtime` : Prints the current simulation clock.

QUEUE object type

`que` : Stores a linked list of the objects which are queueing in a sector.

`length` : Keeps instantaneous queue length.

`maxLeng` : The maximum queue length of the simulation, until current time.

`totDelay` : Cumulative delays in this queue.

`AveWait` : Average waiting times of aircrafts in this queue.

`acDeld` : Number of aircrafts that are delayed in this queue.

`acIn` : Number of aircrafts come into this queue.

`acOut` : Number of aircrafts go out of this queue.

`Cumul` : Stores a statistical data to calculate the time averaged number of aircrafts in the queue.

`totWait` : Stores the total delay times plus the service times of passed aircrafts.($totdelay + totalpasstime$)

`aveW` : Average waiting time in this queue.

`lastPass`: Last aircraft's, -which leaves the sector empty-, exiting time of this queue.

`idleTime`: The total idle times of the queue up to the time.

`maxwait` : Maximum waited aircraft's waiting time in this queue.

`passtime` : An aircraft's passing time for this sector.

`status` : Keeps current status of this sector, either empty or not.

`lastone` : Last aircraft's departing time, for calculating the required spacing time for the succeding aircraft.

`sepMin` : Spacing in minutes that is enforced in this sector.

`hourlyWait`: Stores total delays in one-hour time intervals.

`hourlyACcount`: Stores number of aircrafts entered sector in one-hour intervals.

`nextsect` : Points to sector after this one.

`constructor Initialize`: Initializes the queue with the input data.

`procedure Insert`: Inserts arriving aircrafts into the queue.

`procedure Remove`: Removes an aircraft from the queue which leaves Sector and returns delay of the aircraft.

`procedure Quedep`: Calls Remove method and updates the queue after the removal of the aircraft.

`procedure Average`: Calculates average waiting time in queue.

`procedure Little`: Calculates the statsitics to verify Little's law of *Conservation Equations* for verification purpose.

`procedure FindW`: Finds average aircraft waiting time in this queue.

`function Empty`: Returns true if the queue is empty, and false otherwise.

`procedure Print`: Prints contents of the queue for debugging.

`procedure Stopping`: Reports statistics of the queue at the finishing time of the simulation.

`procedure PrintHourly`: Prints delays in one-hour intervals.

SECOBJECT object type

`sectname` : Keeps the name of the sector.

`Inque,Outque` : Pointers to the arrival and departure route portions of the sector.

`constructor Setdata`: Initializes the sector, sets the links among the sectors.

`procedure CheckQueue`: Checks one of the two queues of the sector, for readiness for departure.

INIAPP object type

`procedure Exiting`: Disposes aircrafts which leave Simulation and/or terminal airspace.

`procedure ArriveSect`: Inserts an aircraft into this sector.

FINAPP object type

`procedure CheckQueue`: Overwritten CheckQueue method of sector object.

`procedure ArriveSect`: Inserts an aircraft into this sector.

APRON object type

`procedure ArriveSect`: Inserts aircraft into this sector.

`procedure SortQue`: Performs Insertion-Sort to sort the aircraft queue waiting for departure, in their departure times.

Appendix B

Object-Oriented Turbo Pascal

In 1989, with version 5.5, BORLAND brings the object-oriented paradigm to Turbo Pascal. Turbo Pascal has been influenced by the AT&T C++ programming language in this paradigm. In fact, these languages are not accepted as *pure* object-oriented languages, but their wide separate usage, makes them very popular object-oriented program development tools.

The new object-oriented features of the Turbo Pascal language will be briefly described in the following sections [4] [9].

Object types

The type declaration for an object is just like that of a record; but in object type declaration, objects can have functions and procedures (or methods) as their "fields". Referencing to these methods is in the same way as referencing to any fields of a classical record type. That is, the dot notation is used to show, a method belongs to an object type variable. This provides with the encapsulation of an object.

The inheritance principle is implemented very easily in the type declaration of a new object; by typing the ancestor type's name, enclosed within a paranthesis after the keyword `object`. (Note that, Turbo Pascal does not allow

Multiple Inheritance, that is, any child object type can have only one ancestor object type.)

Object types can be defined only in the outermost scope of a program or a unit. Object type definitions within procedures and functions are not allowed.

Virtual methods

The polymorphism is implemented by the *virtual* methods of the objects. Virtual methods are different than the classical procedures or functions. In fact, in the declaration section, the only difference is to add a `virtual` keyword after the method declaration. However, the Turbo Pascal compiler handles virtual methods in a much different way. Briefly, the compiler does not link the virtual methods to any other methods, in the compile time; but in the run time. So that the linking is delayed. Therefore, this process is called *the late binding.* In *the early binding* of the static methods, the caller and the callee are connected (bound) at the earliest opportunity, that is, at the compile time. In the late binding, these methods are put into a place to bind the two later on, when the call is actually made. This is done by using a *virtual method table* (VMT) for each of the objects that has a virtual method in their declarations. VMT is created in the data segment of the main memory and every object instance of that object type have a link to it. Every call to a virtual method must pass through the VMT. This extra job slows down the execution speed, but brings the flexibility, by allowing different method code insertions of child objects with the same method name. This is an important principle of the object-oriented programming: the same call to different objects will be responded differently.

Once an ancestor object type declares a method as virtual, all its descendant types that implement a method of that name must declare that method as virtual, as well.

Constructors

Turbo Pascal uses *constructors* to initialize the VMTs for the object types. They are also methods, declared by substituting the keyword `constructor` for `procedure`. A constructor must be declared and called before any virtual methods are called. Constructor methods must be static.

Dynamic objects

Objects can be allocated on the heap and manipulated by the pointers, just as the records. Objects can be allocated as pointer referents with the `new` statement, which assigns enough space for that object type and returns the address of that space, in the pointer. Turbo Pascal 5.5 extends the traditional use of the `new` and `dispose` standard procedures, to allow to take two parameters as their arguments: a dynamic object variable and a procedure name. Therefore, in one statement more work is performed by these procedures. These new uses reinforce the safety. Moreover, they allow to do more work than just allocating or deallocating of heap space, when needed; second arguments of the procedures can do related works such as, initializing or cleaning up the object type.

Destructors

Turbo Pascal 5.5 provides a special type of method called a `destructor`, for cleaning up and disposing of dynamically allocated objects. This procedure helps the `dispose` standard procedure to decide how much memory to release. Destructors use the VMT which contains the size of the variable. Turbo Pascal allows destructors with no code at all, but they still work as well, since they serving a link to the VMT automatically.

Debugging object-oriented Turbo Pascal

Together with the new extensions creating object-oriented Pascal, Borland has also extended the integrated debugger to support object-oriented debugging within the *Integerated Development Environment (IDE).*

For stand-alone testing, version 1.5 (or later) of the Turbo Debugger also supports object-oriented debugging, including several special features that allow access and examination of object-oriented elements. The debugger has new window items such as *The Object Hierarchy Window, The object type Inspector Window, The Modul Window* and *The Object Instance Inspector Window.*

Appendix C

Glossary

Aerodrome Synonym for airport.

Air-carrier Commercial passenger aircraft.

Airline Commercial air-transportation company.

Air traffic control service A service provided for the purpose of preventing collisions between aircraft, and on the manoeuvring area between aircraft and obstructions and expediting and maintaining an orderly flow of air traffic. This service is provided by *area control centre, approach control office* or *aerodrome control tower.*

Air traffic service A generic term meaning flight information service, alerting service, air traffic advisory service, air traffic control service, area control service, approach control service or aerodrome control service.

Airway A control area or portion thereof established in the form of a corridor equipped with radio navigational aids.

Approach control office A unit established to provide air traffic control service to controlled flights arriving at, or departing from , one or more aerodromes.

Apron A defined area, on a land aerodrome, intended to accommodate aircraft for purposes of loading or unloading passengers, mail or cargo, refuelling, parking or maintenance.

Control area A controlled airspace extending upwards from a specified limit above the earth.

Controlled airspace An airspace of defined dimensions within which air traffic control service is provided to controlled flights.

Controlled Flight Any flight which is provided with air traffic control service.

Controlled zone A controlled airspace extending upwards from the surface of the earth to a specified upper limit.

EUROCONTROL European Organization for the Safety of Air Navigation.

Final approach That part of an instrument approach procedure from the time the aircraft has completed the last procedure turn or base turn; or crossed a specified fix; or intercepted the last track specified for the procedure; until it has crossed a point in the vicinity of an aerodrome from which a landing can be made; or a missed approach procedure is initiated.

GMT *(Z-time)* Greenwich Mean Time, which is used as standard time in aviation.

Holding Stack *(H.S.)* A portion of a controlled airspace, where waiting aircrafts kept orderly, before proceeding. Holding pattern.

ICAO International Civil Aviation Organization.

IFR *(Instrument Flight Rules)* Describes the condition that an aircraft is subject to fly according to the directions and help of air traffic control centers located on the ground.

Knot A unit for measuring air speed. (Nautical miles per hour).

Nautical Mile A length measure, equal to 1,852 meters.

Runway A defined rectangular area on a land aerodrome prepared for the landing and take-off of aircraft.

Runway-in-use One of the runways on the aerodrome which allows aircraft movements currently.

Speed Exit A Turn-Off, where the taxiway intersect the runway at an acute angle clockwise.

Taxiway The strips on an airport, except runways, for the aircraft movements.

Terminating Flight *(Originating Flight)* A flight which departs with a different flight number that it has arrived.

Through Flight A flight that arrives and departs with the same flight number.

Turn-Off A region on an airport where the taxiways intersect runways.

VFR *(Visual Flight Rules)* Describes the conditions that the pilot of aircraft takes all responsibilities of the flight.

Bibliography

[1] *ICAO Annex 11: Air Traffic Services*, 1978.

[2] A European planning strategy for air traffic to the year 2010. Technical report, Georgia, 1990.

[3] G. Booch. *Object-Oriented Design with Applications.* Benjamin/Cummings Pub. Co., California, 1991.

[4] Borland International, Inc., California. *Turbo Pascal 5.5 Object-Oriented Programming Guide*, 1989.

[5] A. E. Brant and P. J. McAward. Evaluation of airfield performance by simulation. *Transportation Engineering*, 105:505–522, 1984.

[6] J. R. Clymer. *Systems Analysis Using Simulation and Markov Chains.* Prentice-Hall International, Inc., California, 1990.

[7] P. D. Corey and J. R. Clymer. Discrete event simulation of object movement and interactions. *Simulation*, 56:167–174, 1991.

[8] R. M. Davies and R. M. O'Keefe. *Simulation Modeling with Pascal.* Prentice Hall, Hertfordshire, 1989.

[9] B. Ezzell. *Object-Oriented Programming in Turbo Pascal 5.5.* Addison-Wesley, Massachussetts, 1989.

[10] S. T. Gantt. Analysis of airport/airline operations using simulation. Proceedings of the 1992 Winter Simulation Conference, 1992.

[11] K. E. Gorlen. An object-oriented class library for C++ programs. *Software Practice and Experience*, 17:899–922, 1987.

[12] S. Hall. Air traffic simulation; one and two runway airports. *Transportation Engineering*, 106:756–784, 1985.

[13] O. Işıklı. SIMLIB: A class library for object-oriented simulation. Master's thesis, Bilkent University, 1993.

[14] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis.* McGraw-Hill, Inc, New York, 1991.

[15] D. E. Low. Use of simulation in airport planning and design. *Transportation Engineering*, 95:985–996, 1974.

[16] J. J. Luna. Hierarchical, modular concepts applied to an object-oriented simulation model development environment. Proceedings of the 1992 Winter Simulation Conference, 1992.

[17] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, Hertfordshire, 1988.

[18] M. Pidd. *Computer Simulation in Management Science.* John Wiley & Sons, Chichester, 1992.

[19] H. D. Sherali, A. G. Hobeika, A. A. Trani, and B. J. Kim. An integrated simulation and dynamic programming approach for determining optimal runway exit locations. *Management Science*, 38:1049–1063, 1992.

[20] F. Sokhar, A. Harjanto, and S. V. Nelson. Examination of air traffic flow at a major airport. Proceedings of the 1990 Winter Simulation Conference, 1990.

[21] A. L. Winblad, S. D. Edwards, and D. R. King. *Object-Oriented Software.* Addison-Wesley, Massachusetts, 1990.

[22] J. C. Yu, W. E. Wilhelm, and S. A. Akhand. GASP simulation of terminal air traffic system. *Transportation Engineering*, 95:593–609, 1974.