Université d'Ottawa • University of Ottawa

Program Understanding Tool for MODSIM Programs

(PUMP)

May 15, 2001

Amarjit Singh Bhullar
Computer Science
Ottawa-Carleton Institute for Computer Science
School of Information Technology and Engineering (SITE)
University of Ottawa
Ottawa, Ontario, Canada

A thesis
Presented to the University of Ottawa
In fulfillment of the thesis requirement for the degree of
***Master** of Computer Science*

To father, mother, my wife and our precious son

Abstract

The focus of software engineering has traditionally been oriented toward new software construction inspite of the fact that very substantial costs arise in the maintenance phase of the software lifecycle. A key element in the maintenance phase is "program understanding" because the individuals assigned this task are typically distinct from those on the development team. Tools to assist with program understanding have, nevertheless, received only modest attention in recent years. In this thesis the topic is explored within the context of the development of a particular program understanding tool called PUMP.

PUMP (Program Understanding Tool for MODSIM Programs) is a tool developed to help understand simulation models written in the object-oriented simulation language called MODSIM programs. It has a simple graphical user interface (GUI) and may be used for both system exploration and for browsing purposes as part of a professional software development tool. It quickly displays and provides views of complex inheritance trees, making it an important tool for understanding object-oriented systems.

Acknowledgements

Most importantly, I would to thank Dr. Lou Birta for his guidance and support during the creation of this thesis. I owe him a lot.

I would thank my parents and my family for their unfailing support. I would to thank my wife for her love, patience, understanding and dealing with me being absent even when I was home. I would like to thank my wife's family for their support.

Detailed Table of Contents

List of Figures

1 Introduction
1.1 Project Overview

Software architecture has traditionally been oriented toward new software construction, not toward software understanding for maintenance and evolution. It is widely accepted that over fifty percent of software evolution work is devoted to program understanding [Cle89, Rob91]. Program understanding is the deductive process of acquiring knowledge about a piece of software through analysis, abstraction and generalization. Increased knowledge facilitates common activities such as performing corrective maintenance, reengineering and redocumentation.

Software maintenance generally absorbs more resources than the original program development task. Nevertheless, it is critical for preserving and utilizing existing software [Corbi89]. It has been estimated that $30 billion is spent annually maintenance cost [Corbi89]. A possible characterization of the increase of maintenance cost with time (base on the writer's own professional experience) is shown in Figure 1.1.

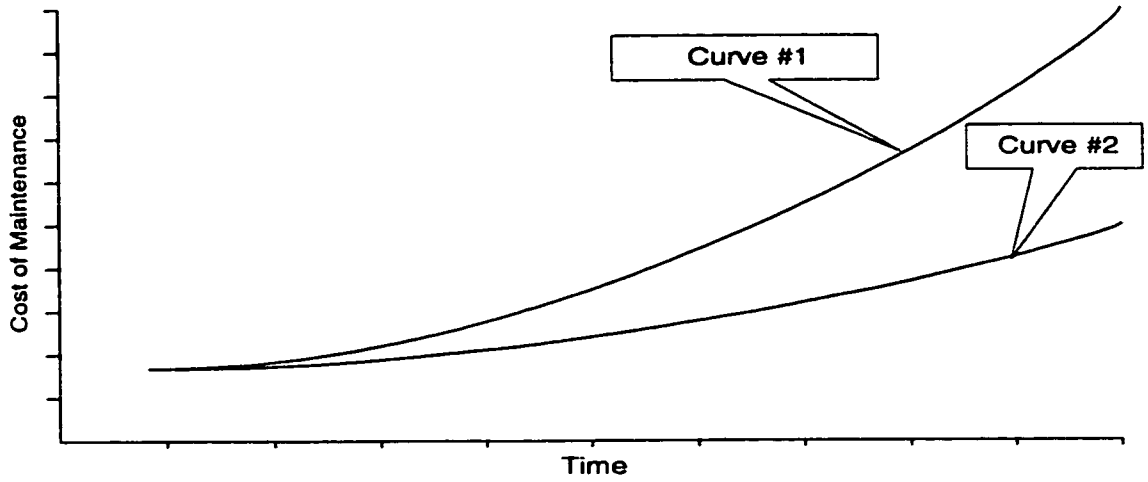


Figure 1.1 Cost of Maintenance after Application is moved to Maintenance

Curve #1 represents the cost of maintenance without using any program understanding tool. The cost of maintaining the application is higher for two reasons:

1.) When the application development is completed, fewer programmers (often novice) are assigned to maintain the application, therefore there is an increasing lost of information about the application.

2.) The quality of documentation is deteriorating. This is often due to unclear, incomplete or lost documentation. Often original developers of the software are often unavailable for consultation about design, organization and even the proper usage of the software (Birta91).

Curve #2 represents the cost of maintenance with the use of a program understanding tool. The cost of maintaining the application still increases with time, but is lower than for case #1 for two reasons:

1.) Even with fewer programmers maintaining the application, the loss of information about the application is reduced. A program understanding tool helps novice programmers to understand the application quickly and therefore to maintain the application more efficiently.

2.) The quality of documentation does not deteriorate since it can be recreated on demand.

Therefore, understanding and automating program understanding are an important components of maintenance tools and methods [Corbi89]. Such tools can substantially reduce the cost of maintenance.

1.2 Background

Program understanding is indispensable for improving the quality of software development. Several development activities such as code reviews, debugging and some testing approaches require programmers to read and understand programs. Maintenance activities cannot be performed without a deep and correct understanding of the component to be maintained. Program understanding is vital to the reuse of code components because they cannot be utilized without a clear understanding of what they do.

Considerable research efforts presenting techniques and tools for analyzing and understanding computer programs have been carried out in recent years. Several of these are reviewed in the discussion below:

a) Algorithmic Approaches

Algorithmic approaches annotate programs according to the formal semantics of a specific model of correctness [Basili91]. As demonstrated by Basili and Mills [Basili82], such formal specification and verification techniques can be useful in documenting and understanding existing complex programs. In an experiment performed by Basili and Mills [Basili82], they try to understand an unfamiliar program of some complexity. Their process consists of reducing the program to be understood to smaller parts and then creating the functions produced by those parts. These functions are then combined at higher and higher levels until a full specification is achieved. A variety of techniques are used to derive the functions of the program parts. For simpler parts of the program, they use direct cognition. In small complex looping parts, they find and verify loop invariants. In the large ones, they organize the effect of major program parts as functions to be determined by additional

analysis. Although this experiment is unautomated, it shows that decomposing programs into their parts can be very effective in their analysis.

Using the functional correctness approach [Mills75], FSQ_2 (Functional Specification Qualifier) supports the derivation of program specifications and the verification of whether or not the program meets those specifications [Basili91]. The FSQ_2 prototype supports a subset of Ada with modifications on the input/output mechanism. It requires the user to provide only the loop function and then a technique is provided to derive the program specification. In a typical session, a user derives the formal specification of a program using step wise abstractions. The user starts by providing trial specifications of every loop in the program as a separate entity. Then, FSQ_2 assists the user in verifying whether or not the loops meet those trial specifications. After finding the actual specifications of all the loops, the correct specification of the whole program is automatically found. This method of step wise abstraction enables the software engineer to concentrate on small pieces of code, one at a time, and thereby simplify the task of specifying the whole program.

Using the axiomatic correctness approach [Hoare69], UNISEX (UNIx-based Symbolic EXecutor) provides mechanical assistance for testing and formally verifying Pascal programs [Kemmerer85]. It consists of two major components: a cross compiler that translates the Pascal program to be symbolically executed into Franz Lisp, and a set of utility routines written in Franz Lisp. The type of correctness being verified is partial correctness [Hoare69]. In other words, if the entry assertion is true when the program is invoked and the program terminates, then the exit assertion will be true when the program halts. However, UNISEX does not provide any assistance in annotating loops with their invariants.

b) Cognitive Studies

Cognitive studies of program understanding provide some insight into how humans understand programs and the factors that affect their understanding[Basili91]. These studies suggest that the understanding process is fundamentally a bottom-up process in which programmers make use of stereotyped solutions to problems in making sophisticated high-level decisions about a program. Instead of perceiving and remembering individual pieces of information, expert programmers process meaningful groups of information. It is also demonstrated that carefully designed high-level semantic information facilitates the understanding of program documentation more than syntactical or structural information.

This view of the process of understanding a program as a fundamentally bottom-up process is supported by many studies [Soloway86]. These studies adopt a theory, which suggests that the programming process is one of constructing mappings from a problem domain, possibly through several intermediate domains, into the programming domain. Consequently, understanding a program involves reconstructing part or all of these mappings. To understand a program, a programmer begins by looking at individual lines of code or at groups of lines and assigning them interpretations. The interpretations are in terms of domains close to that of the program text, such as that of simple algorithms. These interpretations are aggregated to provide higher interpretations for larger and larger segments of code until the entire program is understood [Basili82].

The importance of plans (schemas, chunks) as elementary patterns in perception and thinking is emphasized in these studies. Plans reflect the learned, appropriate, and effective distinctions that people use to make sophisticated high-level decisions.

c) Knowledge-Based Approaches

Several advantages can be gained by using a knowledge-based approach to program understanding [Haberman90]. These approaches provide intelligent assistance for software engineers by utilizing expert-designed plans stored in a knowledge base. In the case where the available knowledge is insufficient for finding an exact solution, they can provide partial solutions. This characteristic makes knowledge-based systems more flexible and user friendly. Expert knowledge is modularized in the form of plans that can be accessed and reused mechanically. By increasing the number of plans in the system, it can be incrementally developed and enhanced.

The approaches reviewed in this section have the common characteristic of utilizing a knowledge base of plans in analyzing programs. They are all implemented, to varying degrees, in automatic analysis systems. However, they differ in following respects:

- The analysis technique (e.g., top-down analysis, bottom-up analysis by graph-parsing, or hybrid analysis).
- The internal notation used to represent programs and, consequently to design the knowledge base plans (e.g., graphs, abstract syntax trees).
- The external notation used to provide the analysis results (e.g., informal natural language text, lambda calculus).

Top-down Analysis. PROUST, which was developed by Johnson and Soloway [Johnson85], uses a top-down analysis strategy to perform automatic analysis and understanding of Pascal programs written by novice programmers. A knowledge base of programming plans and strategies, together with common bugs associated with them, is used

in the understanding and debugging of user programs.

PROUST conducts a matching of a set of functional goals expressed by the user against the program pieces. There is no internal representation of the program being analyzed. A knowledge base plan contains combinations of subgoals and source code statements. The source code statements of a plan are directly matched with the program code. To understand a program, PROUST selects one goal from the goal description. Then, it retrieves a set of plans that implement this goal from its knowledge base and tries to match the individual plans with code. Since a plan may contain subgoals, this process is recursive. The result of the analysis process is a plausible informal explanation of how the given programming intentions have been implemented.

This method of reasoning is convenient when the analysis objective is to confirm a specific hypothesis. It cannot be used to analyze programs in absence of such hypotheses. In addition, a programming task of reasonable complexity can usually be implemented in a variety of ways. In such cases, it becomes difficult to provide all solutions of a problem and supply each of these algorithms with sufficient detail for debugging it.

Bottom-up Analysis. In this subsection, several bottom-up analysis approaches are reviewed and discussed.

Analysis by graph-parsing: The Programmer's Apprentice (PA) project at MIT [Rich81] aims at providing intelligent assistance to programmers for tasks that are required in program development and maintenance. Within this project, the plan calculus is used to represent programs. Here a plan is represented as a graph in which nodes represent operations

and edges show the control and data flows between them. The explicit control and data flow representation abstracts away from the syntactic details of a program. This plan formalism is used in developing a program understanding tool called the Recognizer [Rich90].

The Recognizer [Rich90] is a prototype that automatically finds all occurrences of a given set of commonly used data structures and algorithms, called cliches, in a program. It builds a hierarchical description of the program in terms of the cliches it finds and gives an informal description of the program function. It first translates the program into language independent graphs of control and data flow. The automatic identification of stereotyped fragments is then based on a graph-parsing algorithm described in [Brotsky84]. The advantage of this technique is that it has a flexible, adaptable control structure that can accept advice from external agents. Using this approach, structurally equivalent fragments are represented identically in the data and control flow graphs. However, functionally equivalent fragments do not necessarily have equivalent data and control flow structures.

Heuristic-based object-oriented analysis: The knowledge-based Program Analysis Tool (PAT) designed by Harandi and Ning [Hirandi88] uses an object-oriented framework to represent programming concepts and a heuristic-based recognition mechanism to derive abstract functional concepts from the source code.

Syntactic or semantic program knowledge is expressed in an object-oriented abstract representation, called a program event. Program events are organized in a hierarchy. At the lowest level, there are events representing language constructs like statements and declarations. At higher levels, events can represent standard algorithms and data structures.

Analysis knowledge is represented as a program plan. Knowledge to understand the

events is encoded in a plan's path and test sections. An event set is an instance of a plan if it meets the lexical and control requirements in the path section in addition to any constraints expressed in the test section. Knowledge to generate documentation is stored in a text section and knowledge to perform near-miss debugging is stored in a miss section.

PAT provides a functional description of the program which abstracts away the implementational and structural variations [Kozaczynski89]. However, it does not provide a semantically sound description of the identified program parts. It provides a common-sense explanation of their functional behavior which trades accuracy for simplicity.

Analysis by transformation: Given a functional description of a program, transformational approaches can be used to provide automatic program synthesis which generates executable source code. Transformational approaches to automatic program understanding use the same transformation rules but with their application direction reversed. CPU [Letovsky87], developed by Letovsky, is a tool for analyzing programs by transformation. It uses lambda calculus to represent code, plans and specifications. The understanding process starts by translating the source program into lambda calculus. It then repeatedly transforms the program, using standard plans in a knowledge base, into a more abstract form. The repeated transformations result in a hierarchical structure of program abstractions rooted in the original source code. Higher-level nodes in this hierarchy are specification oriented descriptions.

The Maintainer's Assistant is another tool for analyzing programs by transformation [Ward89]. It expresses both low-level program operations and high-level specifications in a formal language. By utilizing a knowledge base of transformations, the maintenance

programmer can restructure a piece of code or derive its specifications.

The maintenance programmer starts by selecting the piece of code which he/she wants to transform. The tool can automatically derive a suitable set of transformations, from the knowledge base, and apply them to the code. If the transformations are too complicated to be determined automatically, then the maintenance programmer explicitly selects a suitable transformation rule. If the applicability conditions of a transformation rule are not automatically derivable, the maintenance programmer needs to decide whether they are applicable or not.

Even though this approach generates formal specifications, the maintenance programmer is required to perform many tasks ranging from selecting the program part to be transformed to selecting the transformation rule to be applied. This can be very difficult to perform, especially when analyzing unfamiliar programs using a large set of transformation rules.

It seems that a purely transformational approach is not a plausible solution to automatic program understanding [Ning89]. By comparing the input/output relations of automatic program synthesis with those of program understanding, it is clear that automatic program understanding is more difficult. The input to an automatic program synthesis system is often a set of goal specifications emphasizing what is supposed to be performed by the synthesized program. Its output is a subset of all possible implementations. On the other hand, the input to an automatic program understanding system may include all possible implementation instances. Its output is a set of abstractions explaining how the given program achieves its intended functions. In addition, the input to an automatic program synthesis

system is generally more formal and complete than the input to an automatic program understanding system.

Analysis by decomposition. UNPROG, developed by Hartman [Hartman91], recognizes control concepts in unstructured imperative programs. Given an unstructured program, it is first transformed into a language independent representation in the form of a hierarchical control flow/data flow graph. This program representation is then hierarchically decomposed by prospers (single entry/exit control flow subgraphs). The knowledge base plans are represented by similar graphs with added qualifications. Programming concepts are recognized by matching the program parts against the knowledge base plans.

The result of the program analysis yields a general informal description of the kinds of control concepts in the program (e.g., do loop, read-process loop, bounded linear search). No deep analysis of the function of these control concepts is performed. After recognizing the control concepts, a restructuring of the input program can be performed by using some restructuring knowledge associated with the plans.

Another possible decomposition technique is suggested by Hausler et al. [Hausler90]. They suggest the use of program slicing [Weiser84] to decompose loops and to allow the abstraction of loop functions one variable at time. No detailed investigation or discussion of this idea is offered, hence it is not clear how it would affect the size of the knowledge base. Even though the resulting loop slices are independent, each slice must include all the statements affecting the modification of the current variable. This can result in large loop slices which make plan design and identification more difficult. A large knowledge base might be needed to deal with this problem.

Hybrid Analysis. Based on observations of student programmers, Quilici [Quilici93] describes a hybrid approach to program understanding. The procedure begins by producing an abstract syntax tree representation for the program. The components of this tree are matched with the knowledge base plans to deduce their abstract concepts. An indexed, hierarchical organization of the plan knowledge base is used to limit the number of candidate plans considered during program understanding. Each plan has indexing, specialization and implication links to other plans. Indices are used to suggest general candidate plans to match top-down against the code. Specializations are used to refine these general plans once they are recognized. Implications are used to recognize other related plans without doing further matching.

By using hybrid bottom-up/top-down plan recognition, the number of plans that a program understander must try to match against the program, can be reduced. By using indices to make guesses about which plans might actually appear in the code, completeness (the ability to recognize all plan instances in the program) is traded for efficiency (the ability to quickly recognize plans). It has been demonstrated that this approach can efficiently provide informal documentation of small programs [Quilici93]. The plan indices are determined using observations of programmers. This indexing method is time consuming consequently it can only be used in designing plans for small programs.

Another hybrid analysis approach was developed by Bertels [Bertels93]. The understanding of a program is performed bottom-up by first transforming it into an internal representation that is independent of the programming language and the chosen syntactic implementation. This internal representation augments the program instructions with their

equivalent semantic concepts (e.g., increment, sum-instruction). After this transformation, the analysis of the program is performed using the semantic concepts instead of the code itself.

The knowledge base plans contain stereotyped collections of semantic concepts, which are used to present an abstract description of the program and identify some of its faults. To identify context-sensitive faults, similar analysis of a correct teacher program is performed to deduce its abstract description. Comparing the two abstract descriptions, manually, some context-sensitive faults can be detected.

Detecting faults by comparing student programs to the teacher program is similar to PROUST's [Johnson85] top-down analysis. Instead of using goals, Bertels' approach uses a teacher solution. This approach provides informal program descriptions and its underlying recognition process gets very complex even for small programs not exceeding 30-40 lines of code. As pointed out during the discussion of PROUST, a programming task of reasonable complexity usually has a variety of correct implementations. Hence, it becomes difficult to provide all solutions of a problem and supply each of these algorithms with the detail needed for debugging it.

1.3 Goal

The goal of the work carried out in this thesis project has been to design and develop a software tool to provide program understanding support for MODSIM programs. The tool resulting from this work is called PUMP (Program Understanding tool for MODSIM Programs).

A rich collection of languages exists to support simulation model development and simulation studies. MODSIM is a general-purpose, modular, block-structured high-level programming language specifically developed to provide object-oriented programming support for discrete-event simulation. It can be used to build large process-based simulation models through modular and object-oriented development techniques. MODSIM is supported on a variety of machine architectures and operating systems. MODSIM's object-oriented features allow the development of reusable object classes, such as those that are provided in the MODSIM libraries. Inheritance allows objects with a specialized capability (for some particular application) to be derived from other objects having more general capabilities. In order to make use of existing object classes, or to carry out derivations from them, it is necessary to be able to examine their properties. In the case of an object class, which has some complexity in its inheritance ancestry, it can be a tedious task to determine its complete properties since this requires tracing its inheritance through a series of ancestral object definitions.

MODSIM has a browsing tool call the Object Manager. It can be used to explore the properties of objects and other types defined in the MODSIM definition module. This applies equally to those specifics to an application and those supplied in the MODSIM libraries. Object Manager works in conjunction with the MODSIM compiler. When the Object Manager is enabled the compiler scans the definition modules of the application or model, and records information about the declared types and objects in a database. The compiler generates and maintains a database for each separate project.

Although Object Manager provides numerous features, it is not a very user-friendly or efficient tool. Furthermore, its facilities to enhance program understanding are very limited.

1.4 Overview of PUMP

PUMP has been developed as a tool for understanding MODSIM simulation models. There are two main components of PUMP:

1.) An analyzer that extracts and stores all relevant information about a particular project in a database. This analyzer uses a top-down approach. Each statement is broken into several tokens. The analyzer then examines the tokens to determines if it is a reserved word such as FROM, TYPE, OBJECT, etc. These token are then checked against a cross-reference table that contains possible syntax of the reserved word. For example, for FROM, an associated table entry would be FROM id IMPORT id. The program is broken down into several smaller blocks (sometimes called program decomposition), depending on the syntax of the word. For example, FROM GrpMod IMPORT StatQueueObj; is considered to be a block. A knowledge base of the MODSIM objects is used to determine the properties of StatQueueObj, which in this case exist in the GrpMod MODSIM Module. Detail discussion of the analyzer is given in section 5.4.

2.) A user interface that presents project information obtained from the database in a user-friendly manner in response to user queries.

It can be used both for system exploration and for browsing; e.g. it conveniently displays complex inheritance trees (see Figure 1.2). User interaction with PUMP is via a convenient graphical user interface (GUI). PUMP incorporates a powerful editing environment that can be used to examine and edit specified parts of the source code.

Some of the main features of PUMP are outlined in the following sections.

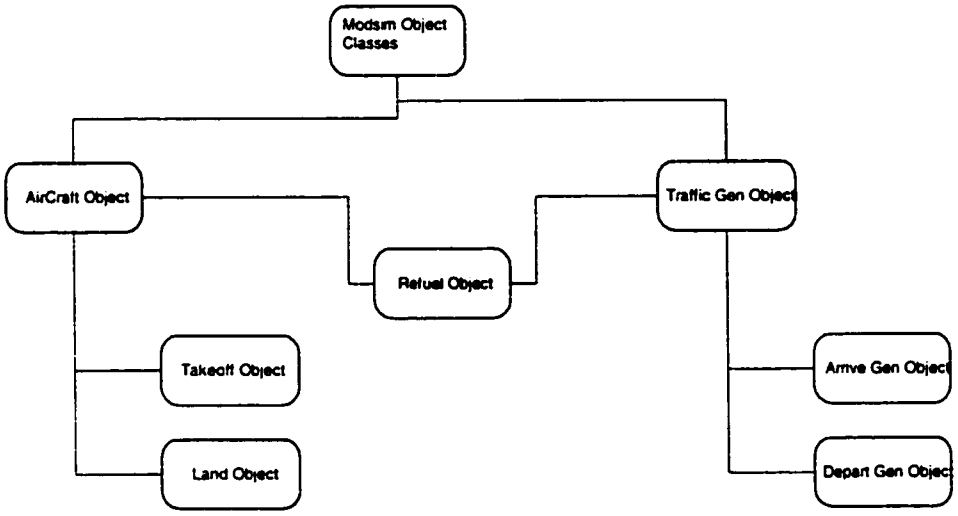


Figure 1.2 Inheritance Tree with Single and Multiple Inheritance

1.4.1 Hierarchy Listings

PUMP presents inheritance trees and is able to accommodate both single and multiple inheritance of an object. Queries that may be made under this feature include:

- show parents
- show children
- show ancestors
- show descendants

1.4.2 Graphical Tree Display

The graphical interface to the browser begins with a compact tree-based view of the set of projects that has thus far been analyzed by PUMP. Any node of this Tree can be readily expanded to reveal aspects of any project in increasing levels of detail. The presentation is highly intuitive so that the user can easily comprehend large inheritance structures. Subtrees may be collapsed or expanded interactively. Scrolling mechanisms are provides with horizontal and vertical scroll bars.

1.4.3 Interactivity

PUMP responds efficiently to queries so that the user maintains a sense of interactivity. This is possible because PUMP accesses the database for all information. Note also that the above comments apply equally to browsing operations because browsing can be regarded as the execution of some predetermined set of queries.

1.4.4 Development Environment

PUMP has been developed using a 32-bit client/server object-oriented development tool called PowerBuilder 5.0.04. The PowerBuilder Foundation Class Libraries have been used in the development process. The database is built using SQLAnywhere ver 5.2. The database runs on an NT Server and PUMP runs on a Windows 95/98 or Windows NT computer. The PowerBuilder Deployment Kit is required to run PUMP on a Windows 95/98 or Windows NT computer. Since the database runs on a server, a SQLAnywhere client must be installed on any computer that runs PUMP. Figure 1.3 shows the PUMP application setup.

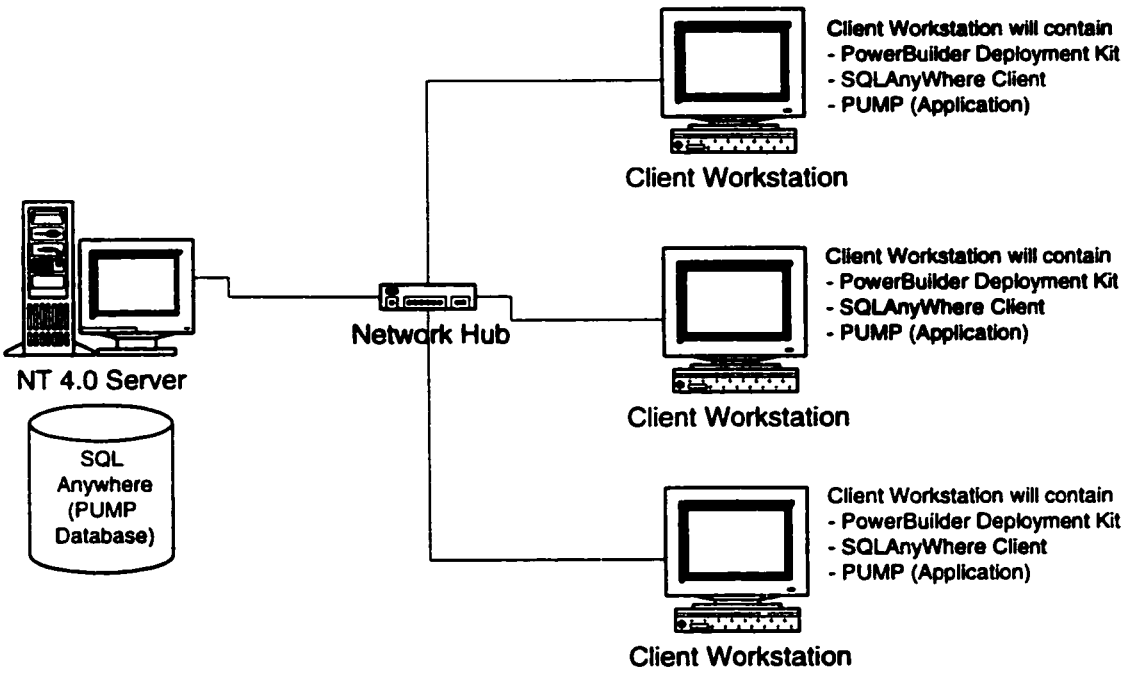


Figure 1.3 PUMP Application Setup

1.5 Thesis Structure

This chapter provides a basic introduction to PUMP. Chapter 2 examines the key features of the object-oriented paradigm and includes a presentation of fundamental object-oriented concepts. Chapter 3 is dedicated to the MODSIM language; MODSIM language syntax and the MODSIM features are presented. In Chapter 4, the PUMP database component is introduced. The database structure is outlined and a detailed description of all the tables in the database is provided. In Chapter 5, the PUMP User-Interface component is introduced. All the components in the PUMP User-Interface component are described. Conclusions and a discussion of possible future work are provided in Chapter 6.

Appendix A provides a declared description of each in the PUMP database. Appendix B presents the script that is used to create the PUMP database. The MODSIM code for an example problem used throughout the body of the thesis, is given in Appendix C. Appendix D

presents some of the key features of the PowerBuilder development tool that has been used to develop PUMP.

2 The Object-Oriented Paradigm

2.1 Introduction

Within the object-oriented paradigm, software development can be viewed as a model building activity that is linked to the problem specification under consideration. As in any problem-solving situation, the first task is to acquire a comprehensive understanding of the problem. An abstract view, or model, of the problem is created by separating relevant details from irrelevant details. This aspect of the modeling process is called abstraction and is illustrated in Figure 2.1.

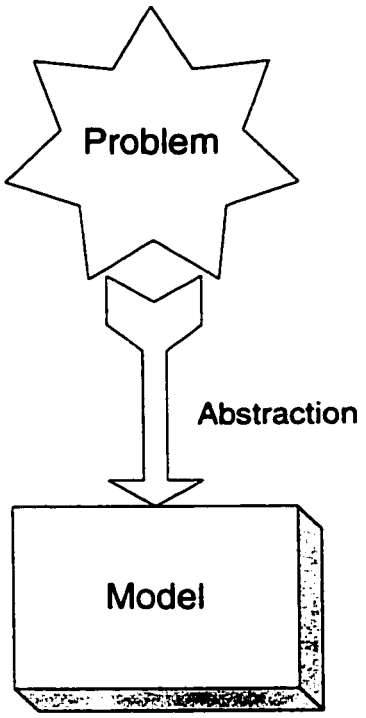


Figure 2.1 Using Abstraction to Create a Model for a Given Problem

A key feature of the model is that it focuses only on the problem's relevant details and helps define properties of the problem. These properties include:

- the data which are manipulated and
- the operations or behavior which are associated with the problem.

As an example consider building a program to simulate an airport. In order to understand this problem, an abstract view, or model, of the problem is created. This model will contain entities or objects that will facilitate the definition of the problem. A natural example of an object is ***Aircraft***.

The ***Aircraft*** object will have certain features such as size, fuel capacity and time required to takeoff and to land. Such features are called the attributes (sometimes called the data structure) of the ***Aircraft*** object and help to formulate a model of a real aircraft. The ***Aircraft*** object will also perform certain tasks or operations such as "refuel", "takeoff" and "land".

The manipulation of the value of an object's attributes is achieved via a set of prescribed external operations. This set of operations is called the interface to the object. Data about the abstract ***Aircraft*** object can only be accessed by using the operations within its interface.

Models that consist of a collection of discrete objects that incorporate both attributes and operations, together with rules that dictate the possible interactions among these objects, are said to fall within the object-oriented paradigm. Abstraction within the object-oriented paradigm corresponds to the structuring of a problem into well-defined entities characterized by their data and operations.

2.2 Characteristics of Objects and Classes

2.2.1 Identity

The state of an object is one of the possible conditions in which it may exist. Identity means that each object is unique, even if its state is identical to that of another object. The ***Aircraft*** object for example can be in one of the three states: *parked, taxing, and flying*.

Objects can be concrete, such as a file in a file system, or conceptual, such as a scheduling policy in a multiprocessing operating system. Each object has its own

inherent identity because each object is given a unique name. Consequently, two objects are distinct even if all their attribute values (for example fuel capacity and size) are identical.

In the real world an object simply exists, but within a programming language each object has a unique name which can uniquely reference it. The naming process may be implemented in various ways, such as an address, array index, or unique value of an attribute. Object references are uniform and independent of the contents of the objects, permitting mixed collections of objects to be created. For example, a file system directory that contains both files and subdirectories can easily be accommodated.

2.2.2 Classification

Classification means that objects with the same attributes and operations are grouped into a class. A class is an abstraction that incorporates only those properties that are relevant to an application. The choice of classes to be associated with an application depends both on the specification of the classes and the nature of the application.

Each class serves as a 'template' or 'blueprint' for a possibly infinite set of individual objects. Each object is said to be an instance of its class. Each instance of the class shares attribute names and operations with other instances of the class and has a (possibly distinct) value for each attribute. This leads to the standard way of thinking about classes: as cookie cutters. Objects are the cookies themselves.

2.2.3 Inheritance

Inheritance is the sharing of attributes and operations among classes based on a hierarchical relationship. A class can be defined broadly and then refined into successively finer subclasses. Each subclass incorporates, or inherits, all of the

properties (attributes and operations) of its superclass (or parent class) and may add its own unique properties. The properties of the superclass need not be repeated in each subclass. For example, "*Takeoff*" and "*Land*" are subclasses of ***Aircraft (See Figure 1.2)***. Both subclasses inherit the properties of ***Aircraft***, such as a "size".

The ability to factor out common properties of several classes into a common subclass and to inherit the properties from the superclasses can greatly reduce repetition within designs and programs and is one of the main advantages of the object-oriented paradigm.

2.2.4 Polymorphism

Polymorphism means that the same generic operation may cause different behavior when invoked within different classes. Recall that an operation is an action or transformation that an object performs or is subject to. "*Refuel*", "*Takeoff*" and "*Land*" are examples of operations. A specific implementation of an operation by a particular class is called a method. The "*Land*" operation, for example, may have an implementation within both the Airport and Traffic Generator classes but the behavior associated with the operation in these two classes may be different. In such a circumstances, the "*Land*" operation is said to be polymorhic.

2.3 Features of Object-oriented Systems.

There are several features inherent in any object-oriented system. Although these features are not unique to such systems, they are particularly well supported in this environment.

2.3.1 Abstraction

Abstraction consists of focusing on the essential, inherent, aspects of an entity. In system development, this means focusing on what an object is and does, before deciding how it should be implemented. Use of abstraction defers decision making as long as possible by avoiding premature commitment to detail. Most modern programming languages provide data abstraction, but the ability to use inheritance and polymorphism provides additional power. The use of abstraction during analysis means that only application-domain concepts need to be addressed, with design and implementation decisions deferred until the problem is fully understood. Proper use of abstraction allows the same model to be used for analysis, high-level design, program structure, database structure, and documentation. A language-independent style of design defers programming details until the final, and relatively mechanical, stage of code development.

2.3.2 Encapsulation

Encapsulation (sometimes also called information hiding) consists of clearly separating the external aspects of an object, which are accessible to other objects, from the internal implementation details of the object, which are hidden from other objects. Encapsulation prevents program modules from becoming so interdependent that a small change in any module has a massive ripple effect. Furthermore, encapsulation allows the implementation of an object to be changed without affecting the applications that use it. One may want to change the implementation of an object to improve performance, fix a bug, consolidate code, or for porting (transfer to another environment). Encapsulation is not unique to object-oriented languages, but the ability to combine data and behavior in a single entity makes encapsulation

cleaner and more powerful than conventional languages where data and behavior are treated separately.

2.3.3 Operation Implementation

The user of an operation need not consider how a given operation is implemented. Operator polymorphism shifts the burden of deciding what implementation to use from the calling code to the class hierarchy. Each object "knows how" to perform its own operations. In an object-oriented programming language, the rules of the implementation language automatically select the correct method associated with an operation based on the name of the operation and the class of the object being operated on. An entity invoking an operation need not be aware of the variety of methods, which may exist to implement a given polymorphic operation.

For example, non-object-oriented code to display the contents of a window must distinguish the type of each figure, such as polygon, circle, or text, and call the appropriate procedure to display it. An object-oriented program would simply invoke the draw operation on each figure; the decision of which procedure to use is made implicitly by each object, based on its class. It is unnecessary to repeat the choice of procedure every time the operation is called in the application program. Maintenance is easier, because the calling code need not be modified when a new class is added.

2.3.4 Sharing

Object-oriented techniques promote sharing at several different levels. Inheritance of both data structure and behavior allows common structure to be shared among several similar subclasses without redundancy. The sharing of code using inheritance is one of the main advantages of object-oriented languages. In fact, more important than the conservation of code, is the conceptual clarity that arises from

recognizing that operations, which may initially be regarded as different, are all really the same thing. This reduces the number of distinct cases that must be understood, analyzed, tested and maintained.

Object-oriented development not only allows information to be shared within an application but also offers the possibility of reusing designs and code in future projects. Object-orientation, however, is not a magic formula for ensuring reusability. Reuse does not just happens even within the object-oriented context. It must be planned by thinking beyond the immediate application and investing extra effort in enhancing the generality of the design to set the stage for better accommodating future unanticipated applications.

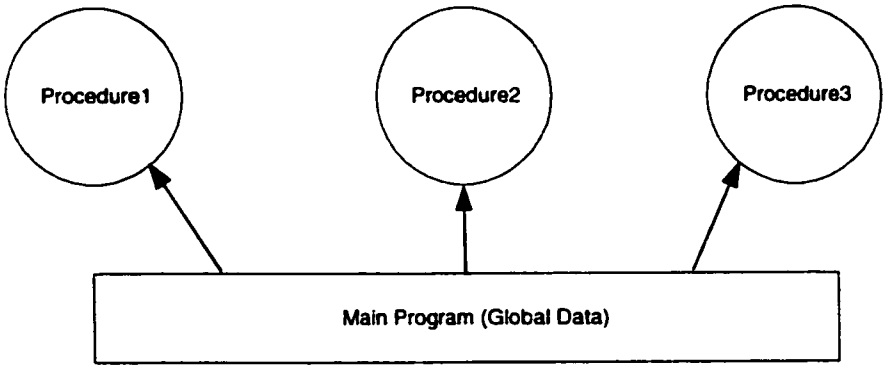
2.3.5 Object Structure vs. Procedure Structure

The object-oriented paradigm focuses on the specification of what objects are, rather than how they are used. The uses of an object depend highly on the details of the application and frequently change during development. As requirements evolve, the features supplied by an object are much more stable than the ways it is used, hence software systems built on object structure are more stable in the long run [Booch86]. Object-oriented development places a greater emphasis on data structure and a lesser emphasis on procedure structure than traditional functional-decomposition methodologies. This fundamental difference in emphasis is shown in Figure 2.2. In part (a) the arrows represents data that is handed over by the main program to the procedures. The main program is the one that coordinates calls to the procedures and hands over appropriate data as parameters. In part (b), the arrows represents messages that are passed by one object to another. Objects of a program

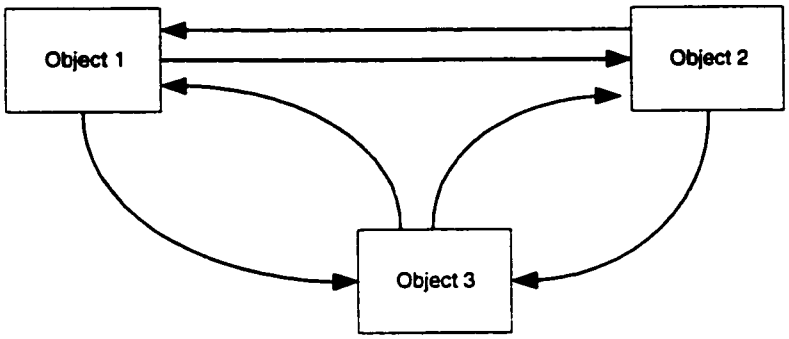
interact by sending messages to each other. A message is a request to an object to invoke one of its methods. It is achieve by using the dot notation "." as shown below:

```
Integer FuelCapacity;
FuelCapacity.setValue(200);
```

In the above example a message is sent to the integer FuelCapacity to invoke the method setValue to set its value to 200.



(a) Procedural Programming



(b) Object-Oriented Programming

Figure 2.2 Procedural vs. Object-Oriented Programming

Object-oriented development is similar to the information modeling techniques used in database design. Object-oriented development, however, adds the concept of class-dependent behavior.

2.3.6 Synergy

Identity, classification, polymorphism, and inheritance characterize mainstream object-oriented languages. Each of these concepts can be used in isolation. The benefits of an object-oriented approach are greater than they might seem at first. The greater emphasis on the essential properties of an object forces the

software developer to think more carefully and more deeply about what an object is and does, with the result that the system is usually cleaner, more general, and more robust than it would be if the emphasis were only on the use of data and operations. According to Thomas, these various features come together to create a different style of programming [Thomas89]. Cox claims that encapsulation is the foundation for the object-oriented approach, shifting emphasis from coding technique to packaging, while inheritance builds on encapsulation to make the reuse of code practical [Cox86].

3 The MODSIM Simulation Language

3.1 Introduction

The object-oriented paradigm provides a particularly effective means for conveniently representing real-world objects in a computer simulation model [CHBKJT93]. Program development with object-oriented languages is, therefore, of particular interest to the simulation community.

MODSIM is a general-purpose, object-oriented, strongly typed, block-structured language, with embedded features tailored to the requirements of simulation model development. More specifically, it has been developed for building large process-based discrete event simulation models through modular and object-oriented development techniques. The modular structure of the language and its syntax is based on Modula-2 [Wirth82], a direct descendent of Pascal.

Figure 3.1 provides a simple example of a MODSIM program to illustrate some basic aspects of its syntactic style. The example computes the average of a sequence of positive numbers, which have been input.

```
MAIN MODULE COMPUTES_AVERAGE;

 VAR
    sum, number : REAL;
    count : INTEGER;
 BEGIN
    OUTPUT("This program computes the average of a sequence of");
    OUTPUT("positive numbers. Enter a sequence of numbers...");
    OUTPUT("Terminate the sequence with a negative number:");
    INPUT(number);
    WHILE number >= 0.0
       INC(count); { increment the count }
       sum := sum + number;
       INPUT(number);
     END WHILE;
     IF count > 0
       OUTPUT(count, " numbers were entered");
       OUTPUT("Average is ", sum / FLOAT(count));
      ELSE
       OUTPUT("Nothing was entered.");
     END IF;
END MODULE.
```

Figure 3.1 An Example of a Simple MODSIM Program

3.2 MODSIM Features

MODSIM has several features that are especially useful for building large process-based discrete event simulation models. These features are summarized in the following discussion.

3.2.1 Modules

MODSIM programs are typically organized as "modules". Each module is stored in a separate file. The advantages of this approach are:

a) Modules may be compiled separately, thereby saving time when only a few modules of a particular group required for a project, are edited.

b) A single module may serve multiple simulation models. In addition modules can import constructs and definitions from each other. The

module concept generalizes the notion of libraries of reusable code.

A MODSIM program consists of a MAIN module and any number of Library modules. Each module in turn consists of two component modules: a **DEFINITION** module and an **IMPLEMENTATION** module. These component modules which are stored in separate files and are compiled separately, are summarized below:

a) A **DEFINITION MODULE** is a named code fragment, which contains declarations for all of the constants, types, variables and methods that are available for import by other modules within the project. Only the headings of procedures and methods are declared in the Definition Module as shown in Figure 3.2.

```
DEFINITION MODULE ControllerMod;
    ControllerObj = OBJECT;
        arriveQ : StatQueueObj;
        departQ : StatQueueObj;
        ASK METHOD LandingClearance(IN plane : LandObj);
        ASK METHOD TakeoffClearance(IN plane : TakeoffObj);
        TELL METHOD ClearOfRunway;
        TELL METHOD ClearOfApproach;
        ASK METHOD ObjInit;
    END OBJECT{ ControllerObj };
END MODULE. { ControllerMod }
```

Figure 3.2 Example of a Definition Module

There is no executable code in a Definition Module. Items declared in a Definition Module are accessible or visible to any other module that may wish to import them. For instance, another module may import a type definition from a Definition Module and then declare variables of that type as shown in Figure 3.3. ASK and TELL Methods are described further in section 3.2.3.

```
FROM ControllerMod IMPORT arriveQ;

VAR
ControllerarriveQ : arriveQ ;
```

Figure 3.3 Example of IMPORT Definition Code

In this example, the variable arriveQ is of type StatQueueObj. This variable is imported from the ControllerMod into another module. A new variable ControllerarriveQ is then defined as of the type arriveQ.

b) An **IMPLEMENTATION MODULE** is a named code fragment which contains the actual code for all procedures and methods that are specified in the associated Definition Module (i.e. the one having the same name) . It may include **CONST**, **TYPE** and **VAR** declarations which are needed solely within that Library Module as shown in Figure 3.4. A variable or data structure which is declared within an Implementation Module is considered global to all procedures and objects in that particular module but is not visible outside of that module.

```
IMPLEMENTATION MODULE ControllerMod;
 OBJECT ControllerObj;
      ASK METHOD LandingClearance(IN AC : LandObj);
        BEGIN
            IF ((arriveQ.numberIn = 0) AND (approachPath = clear))
                  approachPath := inUse;
                  TELL AC TO Land;
            ELSE { AC on go around are put first in arriveQ }
                  IF ((AC.landPriority = goAround) AND (arriveQ.numberIn > 0))
                        ASK arriveQ TO AddBefore(arriveQ.First, AC);
                  ELSE
                        ASK arriveQ TO Add(AC);
                  END IF;
            END IF;
      END METHOD;

      ASK METHOD TakeoffClearance(IN AC : TakeoffObj);
        BEGIN
            IF ((departQ.numberIn = 0) AND (runway = clear) AND (approachPath =
      clear))
                  runway := inUse;
```

```
                TELL AC TO Takeoff;
            ELSE
                ASK departQ TO Add(AC);
            END IF;
        END METHOD;

        TELL METHOD ClearOfRunway;
            { AC which have completed landing rollout or takeoff use this method to
            report that they have cleared the runway. Controller then checks to see if
            an AC is waiting for takeoff }
          VAR
            AC : TakeoffObj;
          BEGIN
            runway := clear;
            WAIT DURATION trafficRanGen.Exponential(sequenceDelay)
            END WAIT;
            IF ((departQ.numberIn > 0) AND (approachPath = clear) AND (runway =
        clear))
                AC := departQ.Remove;
                runway := inUse;
                TELL AC TO Takeoff;
            END IF;
        END METHOD;

        TELL METHOD ClearOfApproach;
            { AC which have cleared the approach corridor use this method to inform
            the controller. The controller then clears the next arriving aircraft to
            land. }
          VAR
            AC : LandObj;
          BEGIN
            IF (arriveQ.numberIn > 0)
                AC := arriveQ.Remove;
                approachPath := inUse;
                TELL AC TO Land;
            ELSE
                approachPath := clear;
            END IF;
        END METHOD;

        ASK METHOD ObjInit;
          BEGIN
            NEW(arriveQ);
            ASK arriveQ TO SetDelayStats(TRUE); { turn ON stats collecting }
            NEW(departQ);
            ASK departQ TO SetDelayStats(TRUE);
        END METHOD;
    END OBJECT { ControllerObj };
END MODULE. { ControllerMod }
```

Figure 3.4 Example of an Implementation Module

3.2.2 Object-orientation

An object is an encapsulation of a data record, which describes the state of some entity. MODSIM support two kinds of procedures, ordinary procedures and a special kind of procedure called methods. They interact through a clearly defined messages. Methods can only be used within an object and are invoked by sending a message to the object asking it to perform a specific method. A new object type (sometimes called class) can inherit the attributes of an existing object type and elaborate on the attributes and methods of its ancestor type (see section 2.2.3)

An object is defined through the use of the TYPE statement as shown in Figure 3.5.

```
TYPE
        VehicleObject = OBJECT
                course : [ 0 .. 359 ];
                speed : INTEGER;
                position : PositType;
    END OBJECT;
```

Figure 3.5 Example of TYPE Statement

The new object identified as VehicleObject is declared here with three attributes: course, position and speed. Note the similarity with the syntax of the Pascal record structure. One of the attributes, position is of a user-defined type called PositType, which is declared elsewhere within the scope of this object (i.e., within the same module) as shown in Figure 3.6.

```
MAIN MODULE Airport;
  VAR
       PositType: REAL;
  TYPE
       VehicleObject = OBJECT
              course : [ 0 .. 359 ];
              speed : INTEGER;
              position : PositType;
    END OBJECT;
.
.
END MODULE. { Main }
```

Figure 3.6 Declaration of the Variable: PositType

MODSIM has two types of methods: namely, ASK and TELL. VehicleObject can be expanded to include an ASK and a TELL method as shown in Figure 3.7:

```
TYPE
       VehicleObject = OBJECT
              course : [ 0 .. 359 ];
              speed : INTEGER;
              position : PositType;
              ASK METHOD GoTo(IN destination : PositType);
              TELL METHOD Stop;
       END OBJECT;
```

Figure 3.7 Example of an ASK and a TELL Method.

VehicleObject now has two methods, an ASK method, GoTo, which passes a variable destination which is of type PositType and a TELL method, Stop. All object declarations appear in a DEFINITION module.

An associated IMPLEMENTATION module contains the code for the methods that are declared in a DEFINITION module. The VehicleObject object within the IMPLEMENTATION module would have a structure shown in Figure 3.8.

```
OBJECT VehicleObject;
        ASK METHOD GoTo(IN destination : PositType);
        BEGIN
                ...
                executable code goes here
                ...
        END METHOD;
        TELL METHOD Stop;
        BEGIN
                ...
                executable code goes here
                ...
        END METHOD;
END OBJECT;
```

Figure 3.8 Example of an IMPLEMENTATION Module

3.2.2.1 Encapsulation

Encapsulation is supported at several levels within MODSIM. The separation of interface specification (DEFINITION) and code implementation (IMPLEMENTATION) permits information hiding and facilitates software reusability through separate compilation. An advantage of separate compilation is that source code for the DEFINITION can be supplied to the software user while the IMPLEMENTATION is supplied only in compiled code. The integration of objects into this scheme is quite natural, with the object declaration being contained in the DEFINITION module and the code for object behaviors in the IMPLEMENTATION module.

A second level of Encapsulation is in the scope rules for object visibility. The scope rules of MODSIM restrict access of attributes and behaviors to those specified in the DEFINITION modules. This provides the data and procedure encapsulation required of objects. An additional information hiding capability is the PRIVATE statement. It is used to restrict access of an object's data and behaviors to methods within the object definition itself. An example of such Encapsulation is shown in Figure 3.9.

```
TYPE
        AircraftObj = OBJECT( VehicleObject )
                altitude : INTEGER;
                TELL METHOD ClimbTo(IN height: REAL);
                TELL METHOD Circle;
                ASK METHOD FindTarget(IN enemy: VehicleObject);
                PRIVATE
                        liftCoefficient : REAL;
                        ASK METHOD CalcLiftCoeff;
                        WAITFOR METHOD DeployLandingGear;
                OVERRIDE
                        TELL METHOD Stop;
        END OBJECT;
```

Figure 3.9 An Example of Encapsulation

The PRIVATE section lists all of the fields and methods, which are part of the object declaration, but which cannot be accessed from outside of the object. Therefore, AircraftObj can be imported into a module and declared as reference variable of that type called plane as shown in Figure 3.10.

```
FROM AircraftMod IMPORT AircraftObj
VAR
        Plane : AircraftObj;
```

Figure 3.10 AircraftObj Declared A Reference Variable

According to the second level of Encapsulation the following code is valid:

```
TELL plane TO ClimbTo(2500.0);
ASK plane TO FindTarget(tank);
```

Since liftCoefficient is a private field and CalcLiftCoeff is a private method of type AircraftObj, therefore, the following code is invalid:

```
ASK plane TO CalcLiftCoeff;
coeffOfLift := ASK plane liftCoefficient;
```

3.2.2.2 Abstraction

Abstraction is implemented by the TYPE and VAR statements as in Modula-2 and Pascal. The object TYPE declaration takes place in the DEFINITION module as shown in Figure 3.11.

```
TYPE
    TakeoffObj = OBJECT(AircraftObj);
        TELL METHOD Takeoff;
        ASK METHOD ObjInit; { set takeoff performance attributes }
        ASK METHOD ObjTerminate; { report statistics bef. DISPOSing }
    END OBJECT;
```

Figure 3.11 TYPE Declaration in DEFINITION Module

Object instances are realized as variables of the associated object TYPE by declaration in the VAR section of an IMPLEMENTATION module as shown in Figure 3.12.

```
OBJECT TrafficGenObj;
    TELL METHOD GenTraffic(IN interarrivalRate : REAL;
                           IN kindOfAC : trafficType);
    VAR
        planeTO : TakeoffObj;
        planeLand : LandObj;
        ...
        executable code goes here
        ...
    END METHOD;
END OBJECT;
```

Figure 3.12 VAR Declaration in IMPLEMENTATION Module

3.2.2.3 Inheritance

It is through the TYPE construct that MODSIM implements inheritance. Once an object type has been defined, new types can be defined based on the existing types. Each descendant in the hierarchy can add its own fields and methods definitions to those of its ancestry.

An example of inheritance in MODSIM is shown in Figure 3.13

```
TYPE
        AircraftObj = OBJECT;
                ovhdTime : REAL;
                taskTime : REAL;
                startTime : REAL;
                ASK METHOD
        FuelCapacity;
        END OBJECT;

        TakeoffObj = OBJECT(AircraftObj);
                TELL METHOD Takeoff;
                ASK METHOD ObjInit;
                ASK METHOD ObjTerminate;
        END OBJECT;

        LandObj = OBJECT(AircraftObj);
                landPriority : priorityType;
                TELL METHOD Land;
                TELL METHOD GoAround;
                ASK METHOD ObjInit;
                ASK METHOD ObjTerminate;
        END OBJECT;
```

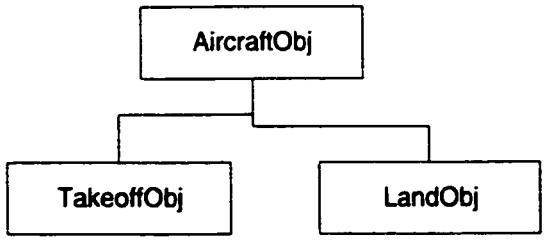


Figure 3.13 An Example of Inheritance in MODSIM

Notice the use of AircraftObj in the TYPE declaration as a parameter to OBJECT. This specifies that the new objects TakeoffObj and LandObj will inherit all of the attributes and methods of the object AircraftObj. A further refinement with the addition of the a new attribute called landPriority (of TYPE priorityType) in LandObj is also shown. In MODSIM, complex derived objects can be constructed through use of multiple inheritance as shown in Figure 3.14.

```
DEFINITION MODULE ...

    ...
    ComputingObj = OBJECT
            ASK METHOD FindTarget(IN enemy: VehicleObj);
    END OBJECT;
    ...
    AircraftObj = OBJECT(VehicleObj, ComputingObj)
    ...
    END OBJECT;

    MissileObj = OBJECT(AircraftObj, WeaponObj)
    ...
    END OBJECT;

    WeaponObj = OBJECT(ComputingObj)
    ...
    END OBJECT;

END MODULE.
```

Figure 3.14 An Example of Multiple Inheritance

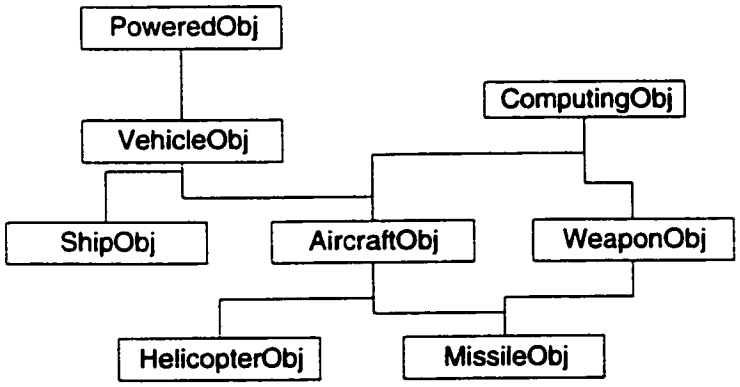


Figure 3.15 Shows a Complex Inheritance Tree

In Figure 3.14, it is shown that MissileObj is derived from an AircraftObj and a WeaponObj. Also it is shown that the AircraftObj is derived from a VehicleObj and a ComputingObj. The inheritance tree is shown in Figure 3.15. Objects defined in this way inherit all of the attributes and methods of the parent objects. This opens the possibility for ambiguity in attribute and method names. MODSIM resolves these conflicts by

requiring direct reference to multiple defined attributes and through use of the OVERRIDE statement for ambiguous methods as shown in Figure 3.16.

```
DEFINITION MODULE ...

    ...
    ComputingObj = OBJECT
        ASK METHOD FindTarget(IN enemy: VehicleObj);
    END OBJECT;
    ...
    AircraftObj = OBJECT(VehicleObj, ComputingObj)
    ...
        OVERRIDE
            ASK METHOD FindTarget(IN enemy: VehicleObj);
    END OBJECT;

    MissileObj = OBJECT(AircraftObj, WeaponObj)
    ...
        OVERRIDE
            ASK METHOD FindTarget(IN enemy: VehicleObj);
    END OBJECT;

    WeaponObj = OBJECT(ComputingObj)
    ...
        OVERRIDE
            ASK METHOD FindTarget(IN enemy: VehicleObj);
    END OBJECT;

END MODULE.


IMPLEMENTATION MODULE ...

    OBJECT MissileObj;
        ASK METHOD FindTarget(IN enemy: VehicleObj);
        BEGIN
        ...
            INHERITED FROM AircraftObj FindTarget(enemy);
        ...
        END METHOD;
    END OBJECT;

END MODULE.
```

Figure 3.16 Illustration of the OVERRIDE Statement

In Figure 3.16, it is shown that FindTarget is a method inherited from the ComputingObj by AircraftObj and WeaponObj. Because MissileObj is derived from AircraftObj and WeaponObj, MissileObj inherits the method FindTarget. To resolve the

conflict between the FindTarget method from the AircraftObj and FindTarget method from the WeaponObj, an OVERRIDE statement is used in the definition of the MissileObj. In the implementation of the MissileObj, the FindTarget method execution code is defined where only the FindTarget method from the AircraftObj is executed during runtime. The INHERITED FROM statement is used to explicitly call the FindTarget method from the AircraftObj.

Instances of a declared object type are created via the use of the VAR statement. Objects are created and destroyed dynamically, at run-time, by the built-in procedures NEW and DISPOSE. Each object, which is created by NEW, according to the type specification for that object type, is called an object instance and multiple instances of objects of that type can be dynamically created as shown in Figure 3.17.

```
OBJECT TrafficGenObj;
    TELL METHOD GenTraffic(IN interarrivalRate : REAL;
                           IN kindOfAC : trafficType);
    VAR
        planeTO : TakeoffObj;
        planeLand : LandObj;
    BEGIN
        WHILE (SimTime <= stopTime)
        WAIT DURATION ranGen.Exponential(interarrivalRate);
        END WAIT;
        INC(numACgen);
          CASE (kindOfAC)
              WHEN arrive:
                  NEW(planeLand);
                  ASK Controller LandingClearance(planeLand);
              WHEN depart:
                  NEW(planeTO);
                  ASK Controller TakeoffClearance(planeTO);
          END CASE;
        END WHILE;
    END METHOD;

END OBJECT;
```

Figure 3.17 Illustration of the MODSIM Built-In Function, NEW

In Figure 3.17 it is shown that a new instances of the object LandObj (var planeLand) and object TakeoffObj (var planeTO) are created using the function, NEW.

3.2.3 Constructs for Supporting Simulation

There are two schemes for organizing discrete-event simulation models; namely, event-oriented and process-oriented [BPBFLS83]. The event-oriented approach is based on the sequencing of a dynamic event list. Each event has an associated routine that determines when other events are placed on the event list. During the execution of an event routine simulation time does not advance. The event list manager advances time when the next scheduled event occurs. A process in the process-oriented approach, is a sequence of logically related activities ordered in time. The routine implementing the process contains all of its related activities. Each process maintains its own activity list. The system maintains a master activity list containing the next activity from each process' activity list.

Process-oriented simulation in MODSIM is supported through a natural extension of its object hierarchy to include the special object SimControlObj together with time-elapsing methods. Objects descended from SimControlObj can have multiple concurrent activities. There are provisions for activities to operate synchronously or asynchronously and to interrupt activities within the same object or in other objects. Simulated time is a REAL (floating-point) value maintained by the MODSIM run-time manager. It is dimensionless and can be used to represent any time resolution desired in a simulation. It is accessed by the function SimTime(). Figure 3.18 illustrate the use and

characteristics of the object SimControlObj. The DEFINITION module of CombatUnit is written below using SimControlObj.

```
        FROM SimMod IMPORT SimControlObj;

TYPE
        CombatUnit = OBJECT(SimControlObj )
                Personnel : INTEGER;
                Location, Destination : Coordinate;
                Speed : INTEGER;
                ASK METHOD Status;
                TELL METHOD MoveTo( IN : NewDestination : Coordinate);
        END OBJECT;
```

Figure 3.18 Illustrates The Use Of SimControlObj Object

The first line of the example above shows the use of the IMPORT statement. The MODSIM module SimMod contains the object declarations of a large variety of constructs necessary to support simulation, including SimControlObj. This IMPORT statement makes the DEFINITION of SimControlObj visible within the scope of the newly defined TYPE CombatUnit.

Notice also the addition of a new method, MoveTo, which is a TELL METHOD. TELL METHODs can only appear in objects derived from SimControlObj. The operation of TELL METHODs in a MODSIM program differs from that of ASK METHODS. ASK METHODs perform like procedure calls in most programming languages. When an ASK METHOD is encountered, the program waits for the ASK to complete before undertaking the execution of the next statement. However, when a TELL METHOD is encountered, the program does not wait for its completion. Instead, the next statement is executed immediately. It is use of the TELL METHOD combined with the WAIT statement that causes simulation time to pass. WAIT statements can only appear in TELL METHODs. A basic form of the WAIT statement is shown in Figure 3.19.

```
WAIT DURATION real-valued-expression
        statement sequence
        [ ON INTERRUPT statement sequence ]
END WAIT;
```

Figure 3.19 Basic Form of WAIT Statement

When this form of the WAIT statement is encountered, the statements after the WAIT will be executed when the specified simulation time has elapsed. An optional INTERRUPT clause is provided to permit other objects to stop the WAIT as shown in Figure 3.19 (the statement between [...] is an optional statement). If the WAIT is interrupted, the statements after the INTERRUPT will be executed.

With respect to the current example, the TELL METHOD, MoveTo, would contain a WAIT DURATION statement where the 'time to wait' is calculated based on the IN parameter, Coordinate.

Processes can be combined to operate synchronously through the use of another form of the WAIT, i.e. the WAIT FOR statement as shown in Figure 3.20:

```
WAIT FOR object TO tell-method( parameter )
        statement sequence
        [ ON INTERRUPT statement sequence]
END WAIT;
```

Figure 3.20 The WAIT FOR Statement

With the use of this form of the WAIT statement, objects can synchronize their activities. This permits the direct expression of logical and physical time dependencies in a natural manner. An example of the WAIT FOR statement is shown in Figure 3.21:

```
WAIT FOR ArmorPlatoon TO MoveTo( 2435 )
```

Figure 3.21 An Example of the WAIT FOR Statement

In example 3.21, it is shown the effect of WAIT FOR statement within a TELL METHOD of CombatUnit is to synchronize its activities with those of ArmorPlatoon.

WAIT statements provide for elapsing simulated time and can be used to synchronize activities. There are, however, situations where processes depend on the occurrence of specific conditions. MODSIM provides the object, TriggerObj, which is used with the WAIT to allow a TELL METHOD wait for some arbitrary conditions to be met as shown in Figure 3.22:

```
WAIT FOR trigger-object TO Fire  (Syntax for TriggerObj )

        landedSignal : TriggerObj;
        ...
        IF flying
                WAIT FOR landedSignal TO Fire
                END WAIT;
        END IF;
```

Figure 3.22 Syntax and an Example of TriggerObj

Here trigger-object is an instance of TriggerObj and thus contains a method call Trigger. This method is typically invoked (fired) by some other method. Its firing permits all methods that are waiting on it to continue.

There are two constructs provided by MODSIM to stop methods that have invoked WAITs: INTERRUPT and TERMINATE. The INTERRUPT method is called from another method to stop a WAIT statement and invoke its ON INTERRUPT clause. For example, if the ArmorPlatoon's MoveTo method had an ON INTERRUPT clause in its WAIT: INTERRUPT ("MoveTo") would cause the statements in that clause to execute. The TERMINATE statement is called from within a process object's method to stop execution immediately.

3.2.4 MODSIM Module and File Naming Conventions

The names for MODSIM modules must contains only letters and numbers and must start with a letter. Furthermore, the MODSIM compiler and associated utilities expect a certain naming convention to be used for files which contain modules. All MODSIM source code files are expected to have the extension **.mod**. The file name for a MAIN module must begin with M, for a DEFINITION module the file name must begin with D, and for an IMPLEMENTATION module the file name must begin with an I. Apart from this first character requirement, the file name must match the module name exactly, and is case sensitive. The following examples illustrate these naming conventions:.

MAIN MODULE Airport → MAirport.mod

IMPLEMENTATION MODULE Aircraft → IAircraft.mod

DEFINITION MODULE Aircraft → DAircraft.mod

4 The PUMP Database Component
4.1 Introduction

The database component of PUMP provides the means for organizing and storing all relevant information about any particular MODSIM project. The are three elements of PUMP database component:

1) a schema for the database,
2) a set of routines to load the database with data, and
3) a set of routines to extract from the database the appropriate information to satisfy high level queries originating from the user interface component.

The schema uses an entity-relationship (ER) model. The main advantage of this model is that it provides a level of data abstraction. It also provides a set of concepts that are useful for describing the structure of a database and details of how data is stored. Relevant features include the data types of the attributes, of the objects under consideration, relationships among the objects, and constraints that should hold on the data. An attribute is a property that describes some aspect or feature of an object. Note that entity-relationship data models are sometimes referred to an object-based models because they essentially describe objects and their interrelationships.

A database that is derived from an entity-relationship model is called a relational database. All data in a relational database are stored in tables. Each table is devoted to a particular category of entity and consists of rows and columns. Each column is associated with some attribute and thus carries a particular kind of information (e.g. a phone number, a name), and each row represents an instance of the entity type that corresponds to the table. Each row in a relational database table is uniquely identifiable by the values in some prescribed set of attributes. This special set of attributes is called

the primary key for the table. Tables can be inter-related via foreign keys. An attribute set, A, in table X is a foreign key if A is a primary key for another table Y. With respect to attribute set A, table X and Y are generally referred to as the foreign and primary tables respectively. The foreign key links the information in the foreign table to that in the primary table.

Figure 4.1 shows the entity-relationship diagram for the PUMP database. This diagram identifies primary keys and foreign keys for each table and relationships between tables. The SQL (Structure Query Language) script to create an instance of the PUMP database, which runs at a particular installation, is provided in Appendix B.

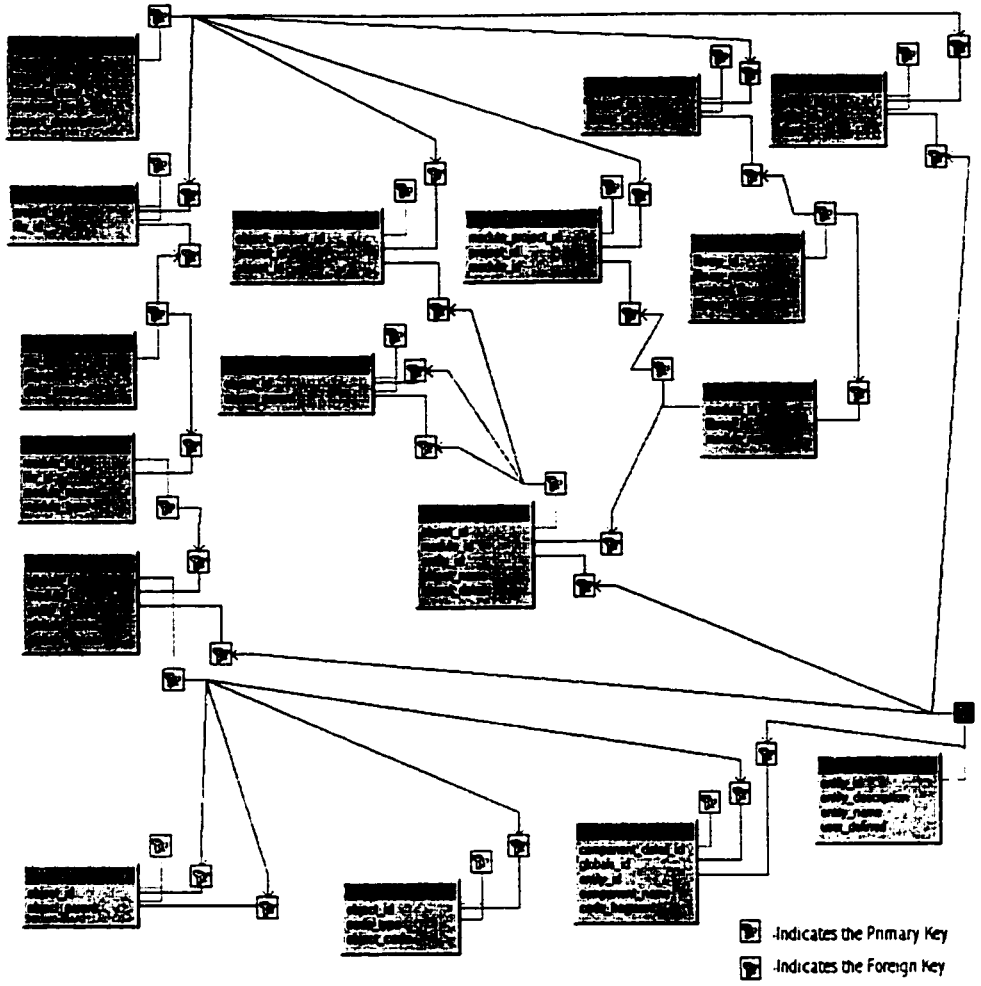


Figure 4.1 Entity-Relationship Diagram for the PUMP Database.

4.2 The PUMP Database

The tables presented in the PUMP entity-relationship diagram shown in Figure 4.1 are examined in detail in this section. Various relevant features of each table shown in Figure 4.1 are outlined. The list of attributes for each table (together with type information) is provided in Appendix A.

4.2.1 The Project Table

This is the highest level table of the PUMP database. It stores information about the project that is being analyzed. The user, via the User Interface (UI) component of

PUMP enters some of this information. The user-entered data is the project name (*project_name*) and a project description (*project_description*). PUMP assigns a unique project identification number (*project_id*) for each project and enters the user name of the individual carrying out the analysis (*analyzed_by*) and the analysis date (*analysis_date*). The *project_id* is the primary key for this table. An example of the two partial data records (i.e. rows) for this table is given in Figure 4.2.

project_id	*project_name*	*analysis_date*	*analyzed_by*
1	Airport	1999/04/26	BhullarA
2	Combat	1999/05/12	BhullarA

Figure 4.2 Sample Contents of the Project Table

There are two projects called Airport and Combat shown in Figure 4.2 having *project_id* values of 1 and 2 respectively.

4.2.2 The Source_file Table

This table stores information about the various source code files associated with each project. Source code files contain MODSIM code and hence have MOD as the file extension. Using the UI component of PUMP, the user enters the names of the various source code files associated with a particular project. Each of the source code files is then assigned a unique file identification number (*file_id*), which is the primary key for the Source_file table.

An example of the data contents for this table is shown in Figure 4.3. In this example, four source code file names (*file_name)* including their paths and their creation date (*date_created*) have been entered.

File_id	*file_name*	*date_created*
1	c:\modsim\icontrollerobj.mod	1999/04/28
2	c:\modsim\mairport.mod	1999/04/28
3	c:\modsim\dcontrollerobj.mod	1999/03/02
4	d:\combat\mcombatunit.mod	1999/03/03

Figure 4.3 Sample Contents of the Source_file Table

4.2.3 The Project_source_files Table

Source files may be shared across projects and it is the Project_source_files Table that maintains such data. This table is called a relational table because it is used to create an association between the Project Table and the Source_file Table as shown in Figure 4.4. This diagram shows that the relationship between the Project Table and Project_source_files Table is one-to-many. Therefore, with any row in the Project Table, there can be associated multiple rows in the Project_source_files Table. The same is true for the relationship between the Source_file Table and the Project_source_files Table. The maximum number of rows in Project_source_files Table is the product of the number of rows in the Project Table and the number in the Source_file Table.

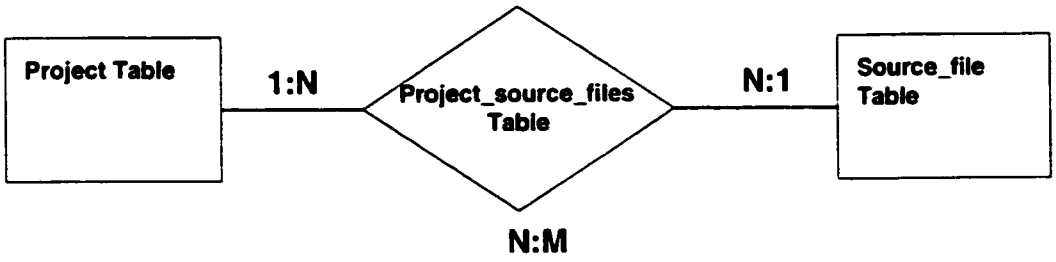


Figure 4.4 ER Diagram to Illustrate the Association between Project Table and Source_file Table

Both the Project_source_files Table and the Project Table have an attribute *project_id*. This attribute is the primary key for the Project Table and is a foreign key for Project_source_files Table. The primary key for Project_source_files Table is the tuple (*project_id, file_id*). Figure 4.5 shows typical content for the Project_source_files Table.

project_id	*file_id*
1	1
1	2
1	3
2	1
2	3
2	4

Figure 4.5 Sample Contents of Project_source_files Table

Figure 4.5 shows that the project whose *project_name* is Airport (*project_id* = 1) has three source code files associated with it. Their names are icontrollerobj.mod (*file_id* = 1), dcontrollerobj.mod (*file_id* = 3) and mairport.mod (*file_id* = 2) (see Figure 4.3). Figure 4.5 also shows that the project whose *project_name* is Combat (*project_id* = 2) has four source code files associated to it. The source code file names are icontrollerobj.mod (*file_id* = 1), dcontrollerobj.mod (*file_id* = 3) and icombatunit.mod (*file_id* = 4). Note that source code files icontrollerobj.mod and dcontrollerobj.mod are being shared by both projects; i.e., Airport and Combat.

4.2.4 The Library_information Table

This table stores information about the various libraries that are used in the project that is being analyzed. The user, using the UI component of PUMP, enters the library name (*library_name*), a library description (*library_description*) and a flag (*modsim_library*) indicating whether the library is a MODSIM library. PUMP assigns a

unique identification number (*library_id*) for each library. Figure 4.6 shows typical contents of the Library_information Table.

library_id	*library_name*	*modsim_library*	*library_description*
1	Standard Library	Y	Contains all constants, variables, types, procedures and objects defined by the standard modsim library modules
2	Airport Library	N	Contains all constants, variables, types, procedures and objects defined for an airport module

Figure 4.6 Sample Contents of the Library_information Table

Figure 4.6 shows that there are two libraries (called Standard and Airport) which have been given library identification numbers 1 and 2 respectively. The Standard Library is a MODSIM library and the Airport Library has been created by the program developer.

4.2.5 The Project_libraries Table

This table is a relational table and is used to create an association between the Project Table and the Library_information Table as shown in Figure 4.7. This diagram shows that the relationship between the Project Table and Project_libraries Table is one-to-many. Therefore, with any row in the Project Table, there can be associated multiple rows in the Project_libraries Table. The same is true for the relationship between the Library_information Table and Project_libraries Table.

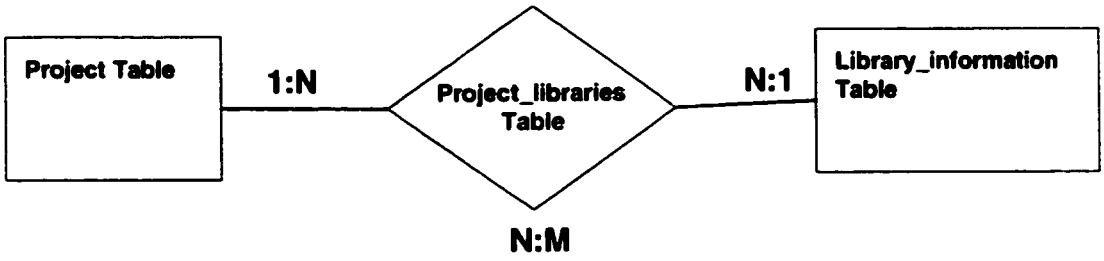


Figure 4.7 ER Diagram to Illustrate the Association between Project Table and Library_information Table

Some typical contents of the Project_libraries Table are shown in Figure 4.8.

project_id	*File_id*
1	1
1	2
2	1
2	2

Figure 4.8 Sample Contents of Project_libraries Table

The *project_id* in the Project_libraries Table is a foreign key because *project_id* is the primary key (*project_id*) for Project Table. The *library_id* in the Project_libraries Table is the foreign key to the primary key (*library_id*) in the Library_information Table. The primary key for the Project _libraries Table is the tuple (*project_id, library_id*).

Figure 4.8 shows that project Airport (*project_id* = 1) has two libraries associated with it; namely, Standard Library (*library_id* = 1) and Airport Library (*library_id* = 2). Figure 4.8 also shows that, project Combat (*project_id* = 2) has the same two libraries associated with. In other words, both the Airport project and the Combat project are sharing these two libraries.

4.2.6 The Module_information Table

This table contains information about all modules associated with source files. Figure 4.9 shows typical contents of the Module_information Table. This data arises from the analysis of the source files for project Airport.

module_id	*file_id*	*module_name*	*module_type*
1	1	ControllerMod	I
2	2	Airport	M
3	3	ControllerMod	D

Figure 4.9 Sample Contents of the Module_information Table

Figure 4.10 shows that there are three different modules associated with project Airport. Each of the modules listed in the table is extracted from the source code files that are associated with project Airport.

4.2.7 The Entity_type Table

This table contains all the entity_types that have thus far been encountered by PUMP. This table initially contains a list of all the entity_types used in the MODSIM language. Each entity_type is assigned a unique identification number. The MODSIM entity_types are INTEGER, REAL, BOOLEAN, CHAR, STRING, ENUMERATION, SUBRANGE, PROCEDURE, FUNCTION, ASK METHOD, TELL METHOD and OBJECT. The primary key for this table is the *entity_id*. User defined entity_types are also entered in this table. Figure 4.10 shows typical contents of the Entity_type Table.

entity_id	*entity_name*	*user_defined*
1	ArrayType	N
2	Constant	N
3	Enumeration Type	N
4	Object	N
5	TrafficGenObj	Y
6	ControllerObj	Y
7	Procedure	N
8	Ask Method	N
9	Tell Method	N
10	Variable	N

Figure 4.10 Sample Contents of the Entity_type Table

When PUMP encounters an entity_type that is not a MODSIM type during the analysis phase, it inserts a new record in the Entity_type Table for this entity_type and assigns a YES flag in the *user_defined* column. The attribute *entity_id* is a foreign key for several tables. These tables are Library_objects, Globals_detail, Component_detail and Project_stats.

Figure 4.10 shows the various entity_types that are used in the MODSIM program that has been analyzed. There are two user-defined types in the table i.e. TrafficGenObj (*entity_id* =5) and ControllerObj (*entity_id* =6). These user-defined types arise following the analysis of the project Airport. Figure 4.11 shows part of the source code that was analyzed by PUMP from the c:\modsim\mairport.mod source code file.

```
32      TrafficGenObj = OBJECT
33              numACgen : INTEGER; { number of AC generated }
34              numACcomp : INTEGER; { number of AC completed landing/takeoff }
35              totTimeSpent : REAL; { total time spent by AC completing task }
37              ranGen : RandomObj; { random number gen. used by this obj }
38              TELL METHOD GenTraffic(IN interarrivalRate : REAL; IN kindOfAC : traffic-Type);
39              ASK METHOD LogCompletion(IN whenStarted: REAL); { when done }
40              ASK METHOD ObjInit;
41      END OBJECT;
42
43      VAR
```

```
44          runway : statusType;
45          approachPath : statusType; { approach corridor }
46          ArriveGen : TrafficGenObj;
47          DepartGen : TrafficGenObj;
48          Controller : ControllerObj;
49          randSeed : INTEGER; { each new generator uses a new seed }
50          trafficRanGen : RandomObj; { used by aircraft to set their fields }
51          goAroundCount : INTEGER;
52          interRate : REAL; { interarrival rate }
```

Figure 4.11 Part of the project Airport source code

Note that on line 32 TrafficGenObj is defined as a user-defined object. It is a user-defined object and is used as a variable type on line 46 and 47.

4.2.8 The Project_stats Table

This table contains some basic statistical data about the entity_types found in the various projects analyzed by PUMP. As PUMP analyzes source code files of a project, it inserts a record for each entity_types found in the project. It then inserts a count of the entity_types found in the project. Because this table creates an association between the Project table and the Entity_type table, it is a relational table (see Figure 4.12). The relationship between the Project Table and Project_stats Table is one-to-many. Therefore, with any row in the Project Table, there can be associated multiple rows in the Project_stats Table. The same is true for the relationship between the Entity_type Table and the Project_stats Table.

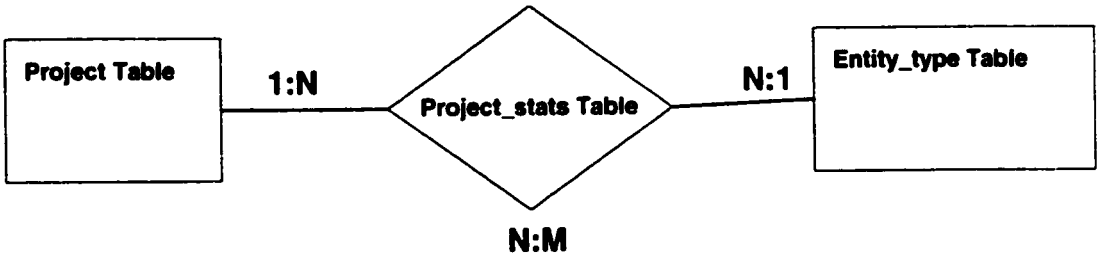


Figure 4.12 ER Diagram for the Association between the Project Table and the Entity_type Table.

The *project_id* in the Project_stats Table is a foreign key for Project Table. The *entity_id* in Project_stats Table is a foreign key for Entity_type Table. The primary key for Project_stats is the tuple (*project_id* , *entity_id*). Figure 4.13 shows typical contents for the Project_stats Table.

project_id	*entity_id*	*count*
1	2	2
1	3	3
1	4	5
1	5	2
1	6	1
1	7	1
1	8	9
1	9	6

Figure 4.13 Sample Contents of the Project_stats Table

In Figure 4.13 indicates that the entity_types used within the Airport project are as listed below with number of instances of each category as indicated.

Constants	2
Enumeration Types	3
Object Types	5
TrafficGenObj	2
ControllerObj	1
Procedure	1
Ask Methods	9
Tell Methods	6
Variables	14

4.2.9 The Globals_detail Table

This table contains information about the items that were defined as global to the entire project. These global items can be variables, constant type, enumeration type, array types and procedures that are accessible to other constructs in the project (e.g. objects, procedures) in the project. All of these globals are defined in the Main Module of a project. This table will also contain information about every user-defined object in the project. PUMP assigns a unique identification number to every global or object found in a project. The primary key for this table is *globals_id.*. Several other tables have attributes, which are foreign keys for Globals_detail; namely, the attribute *object_id* in the Object_inheritance Table, the attribute *object_parent* in the Object_inheritance Table, the attribute *object_id* in the Object_code Table and the attribute *globals_id* in the Component_detail Table.

The attribute *module_id* in this table is a foreign key to Module_information Table. The *entity_id* in the table is a foreign key to Entity_type Table.

Figure 4.14 shows typical content for the Globals_detail Table.

globals_id	*module_id*	*entity_id*	*globals_name*	*globals_detail*
2	2	12	StatQueueObj	FROM GrpMod IMPORT StatQueueObj;
3	2	12	StartSimulation	FROM SimMod IMPORT StartSimulation;
4	2	12	SimTime	FROM SimMod IMPORT SimTime;
5	2	12	RandomObj	FROM RandMod IMPORT RandomObj;
6	2	12	ReadKey	FROM IOMod IMPORT ReadKey;
7	2	3	trafficType	TrafficType = (arrive, depart);
8	2	3	statusType	statusType = (clear, inUse);
9	2	3	priorityType	PriorityType = (normal,goAround);
10	2	4	AircraftObj	AircraftObj = OBJECT; { virtual aircraft type with common attributes }
11	2	4	TakeoffObj	TakeoffObj= OBJECT(AircraftObj);
12	2	4	LandObj	LandObj=OBJECT(AircraftObj) ;
13	2	4	TrafficGenObj	TrafficGenObj = OBJECT
14	3	4	ControllerObj	ControllerObj = OBJECT;
15	2	2	stopTime	stopTime = 1440.0; { minutes }
16	2	2	sequenceDelay	sequenceDelay = 1.0; { interval between departing AC }

Figure 4.14 Sample Contents of the Globals Detail Table

Figure 4.14 shows a list of the globals found in the project Airport. All the entries in this table except entry with *globals_id* = 14, are from the module Airport (*module_id* = 2) which is the Main Module. The entry with *globals_id* = 14 is from the module Controller (*module_id*=3) which is the Definition Module. Every entry in this table has a *entity_id*, which is defined in the Entity_type Table. Figure 4.15 shows the part of the source code listing from the c:\modsim\mairport.mod (*file_id* = 2) source file and c:\modsim\dcontrollerobj.mod (*file_id* =3) source file. The values of the attribute global_detail in Figure 4.14 are marked in bold in Figure 4.15.

```
1       FROM GrpMod IMPORT StatQueueObj;
2       FROM SimMod IMPORT StartSimulation, SimTime;
3       FROM RandMod IMPORT RandomObj;
4       FROM IOMod IMPORT ReadKey;
5
6
7       TYPE
8       trafficType = ( arrive, depart );
9       statusType = ( clear, inUse );
10      priorityType = ( normal, goAround );
11
12      AircraftObj = OBJECT; { virtual aircraft type with common attributes }
17
18      TakeoffObj = OBJECT(AircraftObj);
23
24      LandObj = OBJECT(AircraftObj);
31
32      TrafficGenObj = OBJECT
42
43      VAR
44              runway : statusType;
45              approachPath : statusType; { approach corridor }
46              ArriveGen : TrafficGenObj;
47              DepartGen : TrafficGenObj;
48              Controller : ControllerObj;
49              randSeed : INTEGER; { each new generator uses a new seed }
50              trafficRanGen : RandomObj; { used by aircraft to set their fields }
51              goAroundCount : INTEGER;
52              interRate : REAL; { interarrival rate }
53              ch : CHAR;
54
55      CONST
56              stopTime = 1440.0; { minutes }
57              sequenceDelay = 1.0; { interval between departing AC }
58
```

This is the content of c:\modsim\dcontrollerobj.mod source code file.

```
1       DEFINITION MODULE ControllerObj;
2               ControllerObj = OBJECT;
```

Figure 4.15 Part of the code listing from c:\modsim\mairport.mod (file_id = 2) source file and c:\modsim\dcontrollerobj.mod (file_id =3) source file

4.2.10 The Library_objects Table

This table contains information on the objects that are from the Standard MODSIM library. This table is initialized with the objects within the MODSIM standard library It is similar to the Globals_detail Table. The primary key for this table is *object_id*. Several other tables have attributes, which are foreign keys for Library_objects; namely, the attribute *object_id* in the Library_object_inheritance Table, the attribute *object_parent* in the Library_object_inheritance Table and the attribute *object_id* in the Project_library_objects Table. The attribute *module_id* in this table is the foreign key to Library_modules Table. The attribute *entity_id* in the table is the foreign key to Entity_type Table.

4.2.11 The Object_inheritance Table

This table contains information on the object inheritance. The primary key for this table is the tuple (*object_id*, *object_parent*). The attribute *object_id* is the foreign key to Globals_detail Table. The attribute *object_parent* is the foreign key to Globals_detail Table.

4.2.12 The Library_object_inheritance Table

This table contains information on the object inheritance of MODSIM objects. This table is initialized with the objects within the MODSIM standard library. It is similar to the Object_inheritance Table. The primary key for this table is the tuple (*object_id*, *object_parent*). The attribute *object_id* is the foreign key to Library_objects Table. The attribute *object_parent* is the foreign key to Library_objects Table..

4.2.13 The Object_code Table

This table contains information about the definition and implementation code for a particular object. The primary key for this table is the tuple (*object_id*, *code_type*). The attribute *object_id* is the foreign key to Globals_detail Table.

4.2.14 The Component_detail Table

This table contains information about components that are associated to a particular object. The primary key for this table is the attribute *component_detail_id*. The attribute *globals_id* is the foreign key to Globals_detail Table. The attribute *entity_id* in the table is the foreign key to Entity_type Table.

4.2.15 The Library_modules Table

This table contains information about the all the module from libraries thus far analyzed by PUMP. This table is initialized with the modules within the MODSIM standard library. It is similar to the Module_information Table. The primary key for this table is the attribute *module_id*. Several other tables have attributes, which are foreign keys for Library_modules; namely, the attribute *module_id* in the Library_objects Table and the attribute *module_id* in the Project_library_modules Table. The attribute *library_id* is the foreign key to Library_information Table.

4.2.16 The Project_library_modules Table

This table contains information about the all the modules from libraries thus far analyzed by PUMP and is associated to a particular project. The primary key for this

table is the attribute *module_project_id*. The attribute *project_id* is the foreign key to Project Table. The attribute *module_id* is the foreign key to Library_modules Table.

4.2.17 The Project_library_objects Table

This table contains information about the all the objects from libraries thus far analyzed by PUMP and is associated to a particular project. The primary key for this table is the attribute *object_project_id*. The attribute *project_id* is the foreign key to Project Table. The attribute *object_id* is the foreign key to Library_objects Table.

5 Architecture of PUMP
5.1 Introduction

The eight major components of PUMP provide the means for analyzing, storing and retrieving in a user-friendly manner, all relevant information about any particular MODSIM project. These eight components are listed below and their inter-relationship are shown in Figure 5.1:

1) Project Verification
2) Module Verification
3) Source Code Analyzer (accesses the cross-reference table and reserved word table)
4) Parser/U (carries out parsing of user source code)
5) Parser/L (carries out parsing of components taken from MODSIM Libraries and accesses the MODSIM object knowledge base)
6) Data Storage Process
7) Report Process
8) Query Process

Only three components of PUMP require user interaction in order to carry out their functions. These components are the Project Verification component, the Report Process and the Query Process. The remaining components are internal to PUMP and are transparent to the user.

It should be emphasized that PUMP is not a compiler for MODSIM programs. It is, therefore, not able to detect any syntax or programming errors. All the source code files that are being analyzed by PUMP are assumed to syntactically correct and executable.

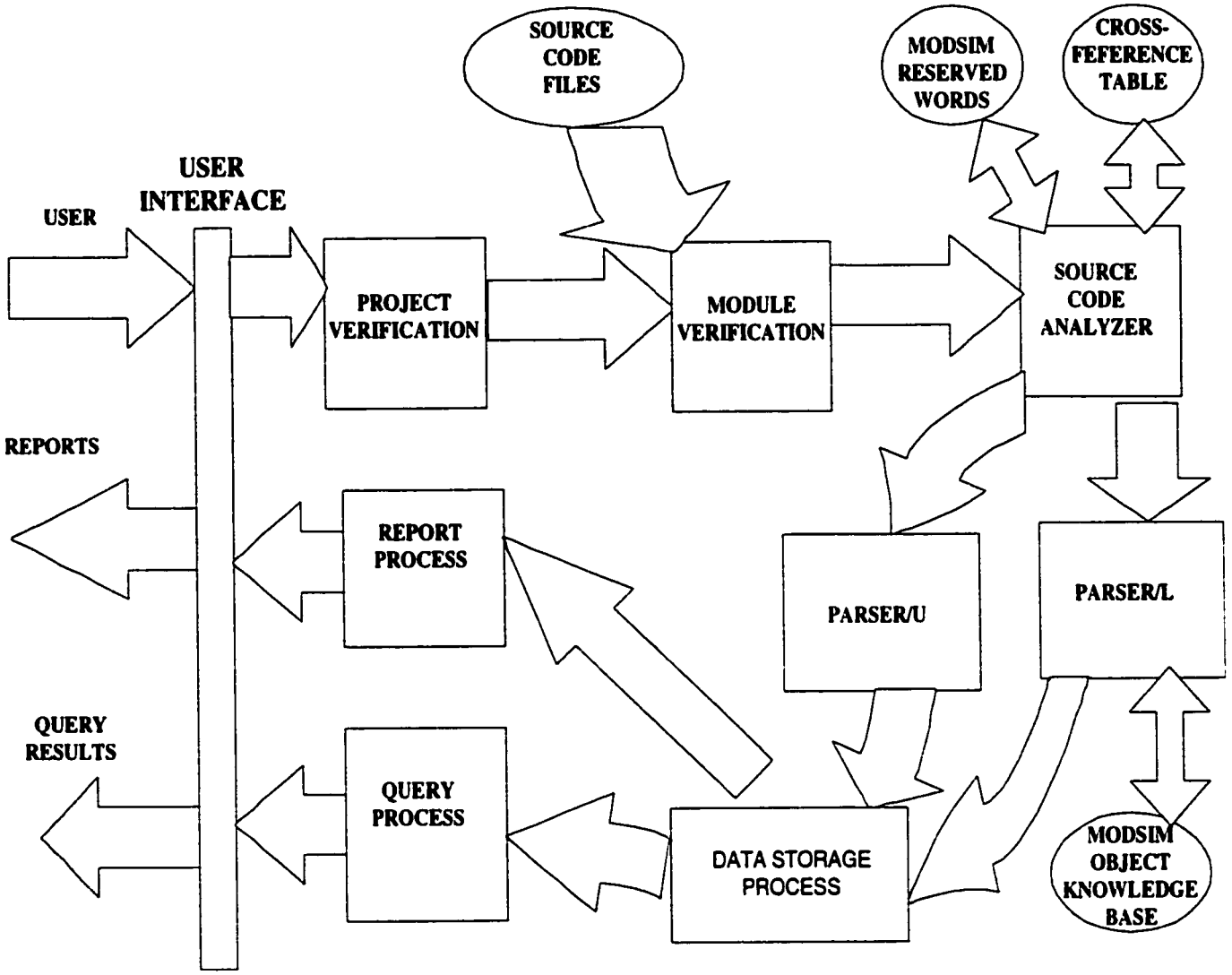


Figure 5.1 Components of PUMP

5.2 Project Verification

This component requires user interaction to carry out some preliminary tasks required prior to the initiation of the project analysis process. The user must enter the project name, the project description and the names of all associated source code files for this project. When entering information about the source code files, the user has the

option of searching for a source code file that already exists in the database. The user enters the filename of source code files, which includes the location of the source code file. When all of this information has been entered by the user and saved, PUMP assigns a unique identification number to this project and enters this information in the Project Table. The project identification number is then assigned to a variable *ll_project_id* and can be used by all the other components. The list of the source code files is then passed to the Source Code File Verification Process, which carries out a phase of the project verification.

5.2.1 Source Code File Verification Process

In this process, several important functions are performed to validate the list of source code files entered by the user. These functions are listed below:

1.) Locate Source Code File: - PUMP locates each of source code files listed by the user. If any of these files is not found, PUMP returns an appropriate error message.

2.) Main Module Source Code File: - As indicated in Chapter 3, a MODSIM program must have a Main Module, which must reside in a file whose filename starts with **'M'**. PUMP searches for a filename that starts with **'M'** in the list of source code files entered by the user. If this search is unsuccessful, an appropriate error message is displayed.

3.) Definition and Implementation Source Code File: - Definition and Implementation Modules may be included in the source code file that contains the Main Module. Alternately these Modules may exist in

independent source code files. However, these modules must either both be included with the Main Module or must separately exist in individual source code files. This requirement is verified in this phase.

4.) Corrupted Source Code File: - PUMP makes sure that none of the source code files listed is corrupted. This task is carried out by reading each source code file line by line. This is an extremely important task to perform because it avoids entering corrupted data in the PUMP database. If any of the source code files is not readable or is otherwise corrupted then an appropriate error message is displayed.

Once all the source code files have been verified, PUMP assigns a unique identification number for each of them and enters this information in the Source_file Table. PUMP then creates an association between the project and each of the source code files and enters this information in the Project_source_files Table. The list of source code file names is then passed to the Module Verification Component.

5.2.2 Modification Process

The Modification Process assists the user in documenting and managing some limited modifications in the project. Modifications can only be performed on projects that have already being analyzed. Two categories of modifications can be accommodated:

1.) Project Description Modification: - The user can update the description of a particular project using the modification window. In this window, the user select the project name from the list of projects that exists in the database. PUMP then searches the Project Table and extract the Project Description for the table and displays it in the modification window. The

user can then update the Project Description. Upon completion of the changes, PUMP updates the record of this project in Project Table with the new Project Description.

2.) Source Code File List Modification: - The user can update the list of source code files for a particular project. In the modification window, user can add or delete a source code file name from the list, for a particular project. This type of modification will trigger the Source Code File Verification Process listed in Section 5.2.1.

5.3 Module Verification

When the Source Code File Verification Process is completed, PUMP passes the list of source code files to the Module Verification Component. This component contains three important verification processes, which are listed below:

1.) Main Module Identification Process: - This process first locates the Main Module in the source code file whose filename begins with 'M' (i.e. the file that should contain the Main Module). This is done by reading the source code file line by line and locating the line that contains the keywords '**Main Module**'. The string between 'Main Module' and the line terminating semi-colon is the name of the Main Module and is extracted. This name is then compared against the filename of the source code file. If there is a difference between the Main Module name and filename, then an error message is displayed and processing stops. If there

is an agreement between the Main Module name and the filename, then appropriate entries are made in the Module_information Table.

If PUMP can't locate the string 'Main Module' in the source code file, then an error message is displayed and the processing stops.

2.) Definition Module Verification Process: - This process is activated when there is a filename beginning with 'D' in the list of source code files associated with the project. The process is entirely analogous to the Main Module Identification process discussed above. The focus in this case is with identifying the Definition Module (in the source code filename that begins with 'D') and again making appropriate entries in the Module_information Table.

3.) Implementation Module Verification Process: - This process is activated when there is a filename beginning with 'I' in the list of source code files associated with the project and is again analogous to the Main Module Identification process discussed above. The focus in this case in with identifying the Implementation Module (in the source code filename that begins with 'I') and making appropriate entries in the Module_information Table.

The role of the Module Verification process is partially redundant in view of the underlying assumption that the MODSIM program files that are being analyzed by PUMP are executable. The main purpose of this phase is to populate the Module_information Table with necessary data. Each module is assigned a unique identification number by PUMP.

5.4 Source Code Analyzer

This component reads each of the source code files that are passed to it by the Module Verification Component. The source code file containing the Main Module is passed to the Source Code Analyzer first, followed by the source code files containing the Definition Modules and then the source code files containing the Implementation Modules. Source code files are read line by line from the top of the program to the bottom. The Source Code Analyzer uses a top-down approach. In this lexical analysis phase each statement is broken into lexical units or tokens and a symbol table is constructed. The analyzer then examines the tokens to determine if it is a reserved word; e.g., FROM, TYPE, OBJECT, etc. This is done by scanning the Reserved Word Table. Tokens that are reserved words are then checked against the cross-reference table that contains templates for the possible syntax of the reserved words. For example, FROM may occur in the form FROM id IMPORT id.

The program is broken down into smaller blocks derived from the reserved word (sometimes called program decomposition) The specific form of these blocks depends on the syntax of the reserved word; for example, FROM GrpMod IMPORT StatQueueObj; is a block. The Source Code Analyzer passes code in blocks to either Parser/U or Parser/L. A block of code begins with a MODSIM keyword and ends at the occurrence of the next MODSIM keyword.

Consider the MODSIM code fragment shown in Figure 5.2. Line # 1 is the first block of code. It starts with a MODSIM keyword 'FROM' and ends with the start of the next MODSIM keyword 'FROM'. This block is sent to the Parser/L to be parsed and analyzed further. The reason the block is sent to the Parser/L is that the block contains the

MODSIM keyword 'IMPORT', which implies the import of entities from library modules. Figure 5.2 shows the first 5 blocks of the code from Figure 5.3.

```
1    FROM GrpMod IMPORT StatQueueObj;                  → Block # 1
2    FROM SimMod IMPORT StartSimulation, SimTime;      → Block # 2
3    FROM RandMod IMPORT RandomObj;                    → Block # 3
4    FROM IOMod IMPORT ReadKey;                        → Block # 4
5    FROM ControllerMod IMPORT ControllerObj;          → Block # 5
```

Figure 5.2 First 5 Blocks of Code

With reference to Figure 5.3, the next block (block # 6) begins at line # 7 and ends at line # 42. This block is a TYPE segment where all user-defined TYPEs are declared: it is, therefore, passed to Parser/U. The next block (block #7) begins at line # 43

```
1   FROM GrpMod IMPORT StatQueueObj;
2   FROM SimMod IMPORT StartSimulation, SimTime;
3   FROM RandMod IMPORT RandomObj;
4   FROM IOMod IMPORT ReadKey;
5   FROM ControllerMod IMPORT ControllerObj;
6
7   TYPE
8   trafficType = ( arrive, depart );
9   statusType = ( clear, inUse );
10  priorityType = ( normal, goAround );
11
12  AircraftObj = OBJECT; { virtual aircraft type with common attributes }
13          ovhdTime : REAL; { overhead time: taxi onto runway for TO or roll out after landing
14          taskTime : REAL; { time required to takeoff or fly approach }
15          startTime : REAL; { sim time at which AC starts Land or TO }
16  END OBJECT;
17
18  TakeoffObj = OBJECT(AircraftObj);
19          TELL METHOD Takeoff;
20          ASK METHOD ObjInit; { set takeoff performance attributes }
21          ASK METHOD ObjTerminate; { report statistics before DISPOSing }
22  END OBJECT;
23
24  LandObj = OBJECT(AircraftObj);
25          landPriority : priorityType; { normal or goAround which is higher }
26          TELL METHOD Land;
27          TELL METHOD GoAround;
28          ASK METHOD ObjInit; { set landing performance attributes }
29          ASK METHOD ObjTerminate; { report statistics before DISPOSing }
30  END OBJECT;
31
32  TrafficGenObj = OBJECT
33          numACgen : INTEGER; { number of AC generated }
34          numACcomp : INTEGER; { number of AC completed landing/takeoff }
35          totTimeSpent : REAL; { total time spent by AC completing task }
37          ranGen : RandomObj; { random number gen. used by this obj }
38          TELL METHOD GenTraffic(IN interarrivalRate : REAL; IN kindOfAC : traffic-Type);
39          ASK METHOD LogCompletion(IN whenStarted: REAL); { when done }
40          ASK METHOD ObjInit;
41  END OBJECT;
42
43  VAR
44          runway : statusType;
45          approachPath : statusType; { approach corridor }
46          ArriveGen : TrafficGenObj;
47          DepartGen : TrafficGenObj;
48          Controller : ControllerObj;
49          randSeed : INTEGER; { each new generator uses a new seed }
50          trafficRanGen : RandomObj; { used by aircraft to set their fields }
51          goAroundCount : INTEGER;
52          interRate : REAL; { interarrival rate }
53          ch : CHAR;
54
55  CONST
56          stopTime = 1440.0; { minutes }
57          sequenceDelay = 1.0; { interval between departing AC }
58
```

Figure 5.3 A MODSIM Code Fragment

and ends at line # 54. This block is a VAR segment (where all the user-defined VARs are declared) and is also sent to Parser/U. The next block (block #8) begins at line #55 and ends at line # 58. This block is a CONSTANT segment and contains the specification of user constants. The block is, therefore, again passed to Parser/U.

Each block may broken into smaller blocks (if necessary) by each parser; i.e., Parser/L and Parser/U, to be analyzed further.

5.5 Parser/L

This component analyzes the blocks of code that are identified, by the Source Code Analyzer, to be Modules, Objects or Methods taken from either the MODSIM Standard Library of from user Libraries. The goal of the Parser/L is to populate the following tables: Project_libraries , Project_library_modules and Project_library_objects. The process of populating these tables is as follows:

1.) Parse each line of code from the block into tokens, and if necessary create sub-blocks.

2.) Execute a set of pre-defined SQL Queries to and obtain the appropriate identification numbers from the Library_modules Table and Library_objects Table. These two table are the knowledge base component of Parser/L

3.) Insert appropriate data into the Project_libraries Table, Project_library_modules Table or Project_library_objects Table.

To illustrate the process further, consider block # 1 passed to Parser/L as shown below:

1 FROM GrpMod IMPORT StatQueueObj; → Block # 1

When this block is passed to the parser, the various constituent tokens are identified and extracted. The second token; i.e., the string 'GrpMod', is passed to the pre-defined SQL Query shown below:

```
SELECT MODULE_ID, LIBRARY_ID
INTO :ll_module_id, :ll_library_id
FROM LIBRARY_MODULES
WHERE MODULE_NAME = 'GrpMod' ;
```

This query returns the *library_id* (in the variable *ll_library_id*) and *module_id* (in the variable *ll_module_id*) that are associated with the 'GrpMod' MODSIM Module. In this case *ll_library_id* has the value 12 and the *ll_module_id* has the value 34. The analyzer passes the fourth token, i.e., the string 'StatQueueObj' and *ll_module_id* to the predefined SQL Query shown below:

```
SELECT OBJECT_ID
INTO :ll_object_id
FROM LIBRARY_OBJECTS
WHERE MODULE_ID = :ll_module_id AND OBJECT_NAME = 'StatQueueObj' ;
```

This query returns the *object_id* (in the variable *ll_object_id*) that is associated with the 'StatQueueObj' MODSIM Object. In this case the value of *ll_object_id* is 65. The analyzer passes the variable *ll_library_id* and the variable *ll_project_id* which contains the *project_id* of the Project being analyzed (assigned by the Project Verification component) to a pre-defined SQL Query to insert a new record in the Project_libraries Table as shown below:

```
INSERT INTO PROJECT_LIBRARIES VALUES (:ll_project_id, :ll_library_id);
```

The analyzer assigns a unique identification number to the *module_project_id*. The following SQL Query is the constructed from a pre-defined template:

```
INSERT INTO PROJECT_LIBRARY_MODULES
    VALUES (:module_project_id, :ll_project_id, :ll_module_id);
```

The analyzer assigns a unique identification number to the *object_project_id*. The following SQL Query is the constructed from a pre-defined template:

```
INSERT INTO PROJECT_LIBRARY_OBJECTS
    VALUES (:object_project_id, :ll_project_id, :ll_object_id);
```

All the INSERT SQL Queries are passed to the Data Storage Process Component which inserts data into the PUMP database.

5.6 Parser/U

This component analyzes any block of code that relate to user-defined TYPEs, VARs, CONSTANTs, OBJECTs and METHODs. The goal of Parser/U is to populate the following tables: Globals_detail, Component_detail, Object_code and Object_inheritance. The process of populating these tables is as follows:

1.) Parse each line of code from the block into tokens, and if necessary create sub-blocks.

2.) Execute a set of pre-defined SQL Queries to obtain certain identification numbers from the Globals_detail Table and Component_detail Table.

3.) Insert appropriate data into the Globals_detail Table, Component_detail Table, Object_code Table and Object_inheritance .

To illustrate the process, consider block # 6 passed to Parser/U as shown in Figure 5.4. When the block is passed to the parser, the identification of sub blocks is carried out as shown in Figure 5.4.

```
7   TYPE
8   trafficType = ( arrive, depart );                                   Block # 6a
9   statusType = ( clear, inUse );
10  priorityType = ( normal, goAround );
11
12  AircraftObj = OBJECT; { virtual aircraft type with common attributes }
13          ovhdTime : REAL; { overhead time: taxi onto runway for TO or roll out after landing   Block # 6b
14          taskTime : REAL; { time required to takeoff or fly approach }
15          startTime : REAL; { sim time at which AC starts Land or TO }
16  END OBJECT;
17
18  TakeoffObj = OBJECT(AircraftObj);
19          TELL METHOD Takeoff;
20          ASK METHOD ObjInit; { set takeoff performance attributes }          Block # 6c
21          ASK METHOD ObjTerminate; { report statistics before DISPOSing }
22  END OBJECT;
23
24  LandObj = OBJECT(AircraftObj);
25          landPriority : priorityType; { normal or goAround which is higher }
26          TELL METHOD Land;
27          TELL METHOD GoAround;                                       Block # 6d
28          ASK METHOD ObjInit; { set landing performance attributes }
29          ASK METHOD ObjTerminate; { report statistics before DISPOSing }
30  END OBJECT;
31
32  TrafficGenObj = OBJECT
33          numACgen : INTEGER; { number of AC generated }
34          numACcomp : INTEGER; { number of AC completed landing/takeoff }
35          totTimeSpent : REAL; { total time spent by AC completing task }
37          ranGen : RandomObj; { random number gen. used by this obj }         Block # 6e
38          TELL METHOD GenTraffic(IN interarrivalRate : REAL; IN kindOfAC : traffic-Type);
39          ASK METHOD LogCompletion(IN whenStarted: REAL); { when done }
40          ASK METHOD ObjInit;
41  END OBJECT;
42
```

Figure 5.4 Sub Blocks of Block # 6

Consider block # 6b. The specification of the user-defined object called AircraftObj is first detected. It has three variables: ovhdTime, taskTime, and startTime, all of type REAL. Parser/U assigns a unique identification number to this object and populates the Globals_detail Table with the appropriate data. The three variables are components of object AircraftObj, therefore, the Component_detail Table is populated with three records, one for each variable.

Consider block #6c. It is first detemined that it specifies the user-defined object called TakeoffObj. It has three methods: one ASK METHOD and two TELL METHODs. Parser/U assigns a unique identification number to this object and populates the

Globals_detail Table with the appropriate data. The three methods are components of the TakeoffObj. Therefore, the Component_detail Table is populated with three records, one for each method.

The difference between block #6b and block #6c is that, the TakeoffObj is a subclass of AircraftObj object. Therefore the Object_inheritance Table is populated with the appropriate data where AircraftObj is set as the parent of TakeoffObj.

All the INSERT SQL Queries are passed to the Data Storage Process Component. This component then saves the record into the PUMP database.

5.7 Data Storage Process

The Project Verification, Module Verification, Parser/U and Parser/L components all use this process to save data. The main purpose of this process is to maintain the integrity of data in the PUMP database. If any duplication of data occurs during the saving of a record, this process alerts the user with an appropriate message.

5.8 Report Process

This process enables the user to execute several pre-defined reports for a selected project. The reports are listed below:

1) Project Stats Report – This report lists basic statistics relating to entities used in a project.
2) Project Source Files Report – This report lists all the source files associated with a project.
3) Project Modules Report – This report lists all the modules in a source file associated with a project.

4) Project Globals Report - This report lists all the Globals in a module that is a particular source file associated with a project.

5) Project Component Names Report – This report lists all the components that are associated with a project.

6) Project Component Details Report – This report lists all the details of components that are associated with a project.

The reports listed above are generated using the PowerBuilder Datawindow and their associated pre-defined SQL Queries. The process is shown in Figure 5.5

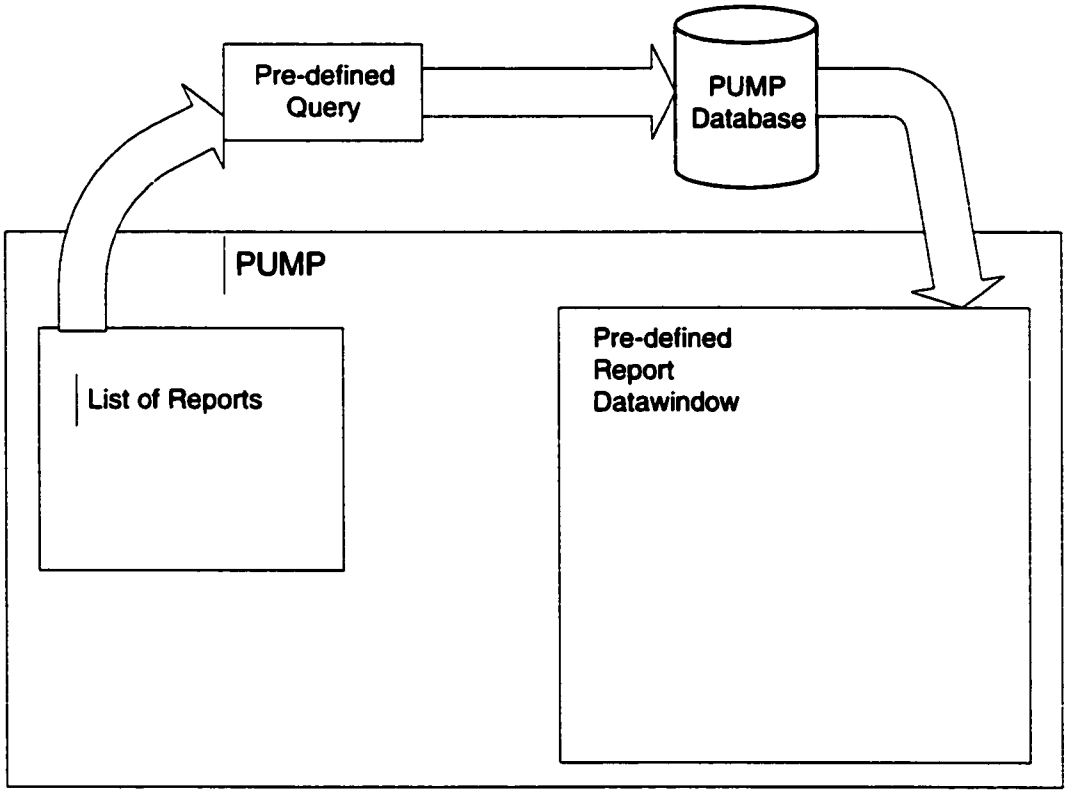


Figure 5.5 Executing the Pre-defined Reports

5.9 Query Process

This process enables the user to access all the information about a particular project. The approach taken to display the information for a particular project emulates

the presentation style used in the Windows 95 Explorer. Recall that in Windows Explorer, user can see both the hierarchy of folders on a selected storage medium (hard disk, floppy disk or a tape) and all the files and folders in each selected folder. By double clicking on a selected folder, the user can expand the Tree and view all the files and folders for that folder. By double-clicking on the same folder again, the user can collapse the Tree. The process of expanding and collapsing the Tree is extremely complex. A recursive function was written to display the Tree View and a particular challenge here was the problem of maintaining the position of each row in the Tree View during the recursive calls to this recursive function.

Windows Explorer presentation is divided into two panes, the left pane displays a hierarchical list of all folders and the right displays the contents of the selected folder. In PUMP, a similar presentation style is used to present the analysis results of a MODSIM project. The presentation window has two panes and is called the Tree View Window. The left pane (the Tree View DataWindow) displays the hierarchical list of all projects and their appropriate source code files, module names, globals names, component names and component details. The right panel of the Presentation Window displays detailed information of the selected items on the left pane, using an appropriate datawindow depending on the item selected. A 'Go To Source' command button is provided to generate a view of that portion of the source code file that contains each of the item selected in the Source Code Viewer window. The user can also scroll through the contents of the source code file or search for any text string by using the 'Find' command button (an elaboration is provided in the following sections).

5.9.1 Project Information

The user can see both the information about the project in the PUMP database and all the related information for each selected project. By selecting the Tree View Menu option from the menu, the user can view all the projects that are currently in the PUMP database. Once a user has selected a project to view, there are several levels of information that are displayed by expanding the tree. The project name is considered to be level one information. Each of the levels has its own datawindow associated with it and is tailored to the information at that level. The Tree View datawindow is considered to be the root datawindow to which are connected a sequence of child datawindows. Figure 5.6 shows the hierarchical view of the levels.

The top six levels of information that can be viewed are listed below.

Project Names	(Level 1)
Source Code Filenames	(Level 2)
Module Names	(Level 3)
Globals Names	(Level 4)
Component Names	(Level 5)
Component Details	(Level 6)
.	.
.	.
.	.
.	.

Figure 5.6 Hierarchical View of All the Levels

Any level below level six has the same functionality as level six because a recursive call is made to the functions used in level six. Therefore the levels displayed can be extended to any level after level six. This approach was taken because of the inheritance (including multiple inheritance) feature supported by MODSIM.

5.9.1.1 Project Names

The Project Name is the level one information. This information is displayed by executing the SQL Query shown below:

```
SELECT PROJECT_ID, PROJECT _NAME
FROM PROJECT
ORDER BY PROJECT_NAME
```

This query obtains all project names (*project_name*) and project identification numbers (*project_id*) from the Project Table and displays them in the Tree View Datawindow, (Datawindows are discussed in Appendix D) which is on the Presentation Window as shown in Figure 5.7.

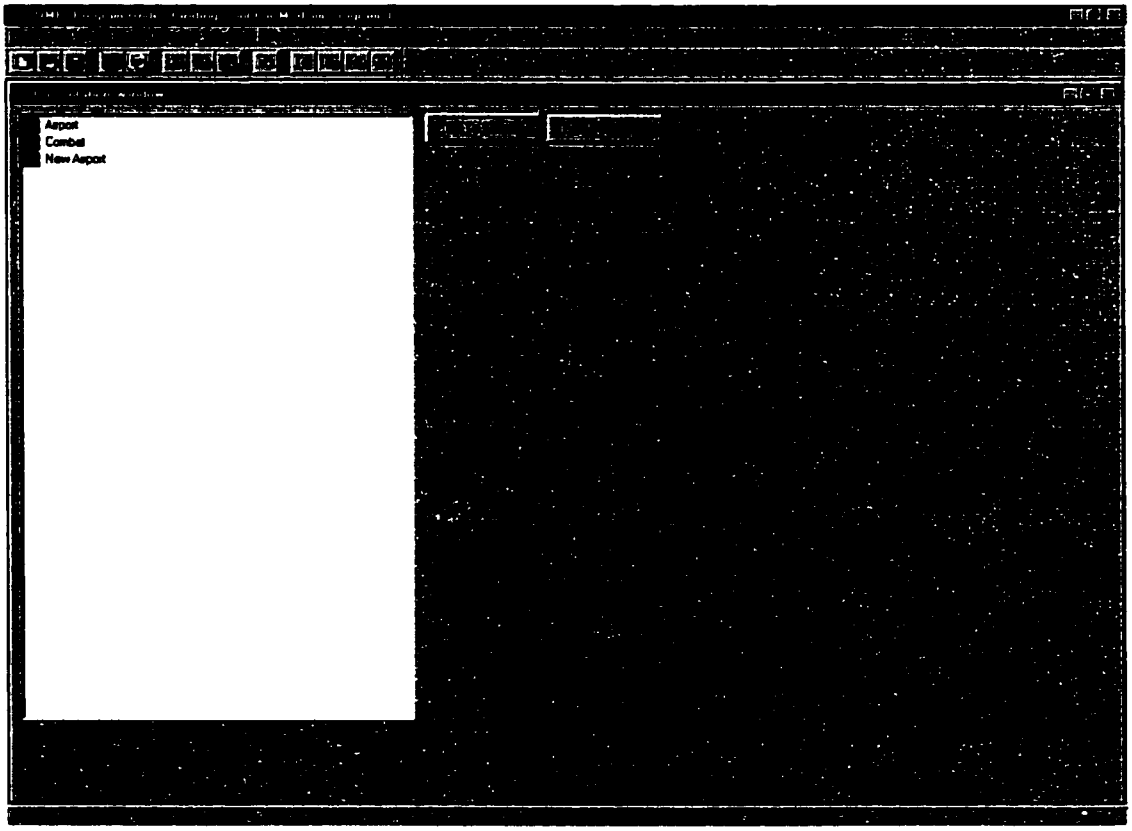


Figure 5.7 List of Projects from the PUMP Database

When the user clicks on a project name in the Tree View datawindow, detailed information about the project is displayed in Project Detail datawindow. This is achieved by obtaining the project identification number (*project_id*) from the selected project name and passing it to the Project Detail datawindow. The datawindow then executes the following PUMP generated SQL Query to obtain the relevant information about the project. The SQL Query is shown below:

```
SELECT PROJECT_NAME, ANALYSIS_DATE, ANALYZED_BY,
        PROJECT _DESCRIPTION
FROM PROJECT
WHERE PROJECT_ID = 11
```

In the query above the project identification number (*project_id*) passed is 11 which corresponds to the Airport project which was selected in the Tree View datawindow. The specific information that is presented is shown in Figure 5.8 (see also Appendix C).

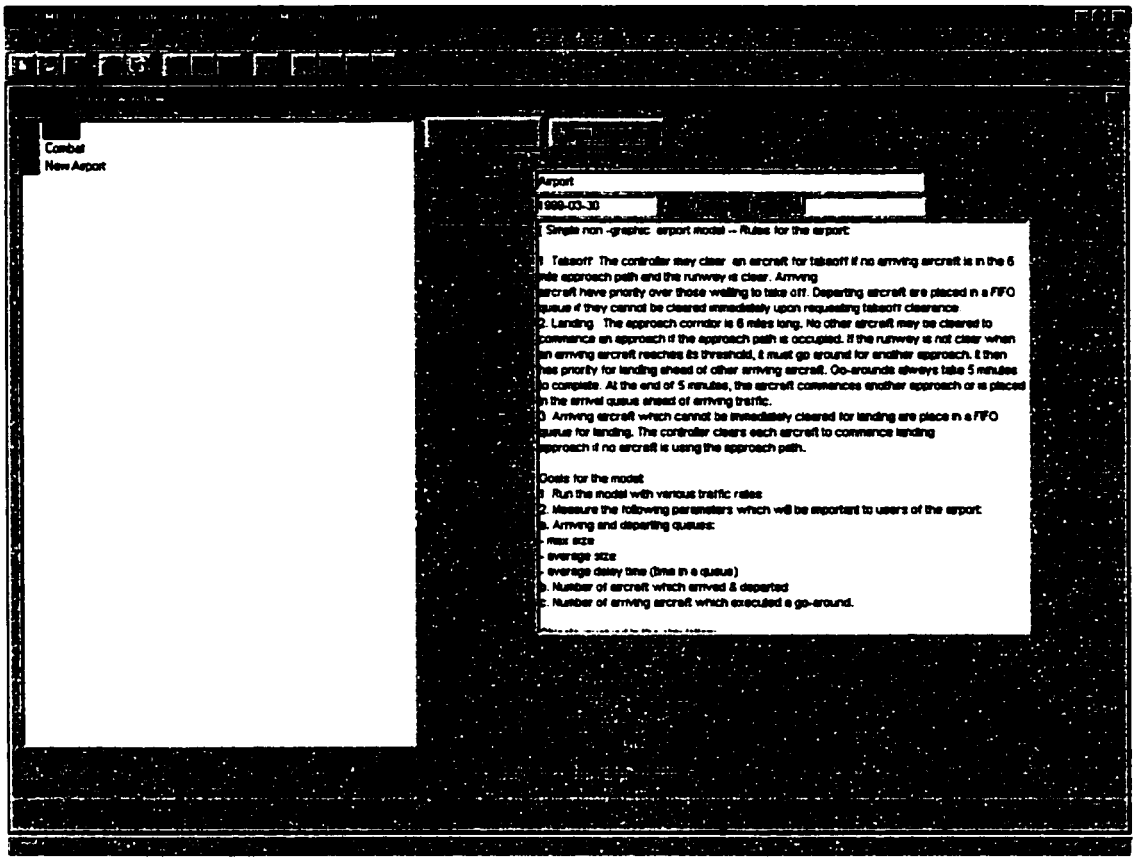


Figure 5.8 Project Information Displayed in Tree View Window

5.9.1.2 Source Code Filenames

This is the level two information of a MODSIM Project. When the user double-clicks on the project name, PUMP displays all the source code filenames that are associated with this project. This is achieved by obtaining the project identification number (*project_id*) of the selected project and passing it to the to Source Code Filenames datawindow. This datawindow returns a list of associated source code filenames in alphabetical order by executing a pre-defined SQL Query as shown below:

```
SELECT SOURCE_FILE.FILE_ID, SOURCE_FILE.FILENAME,
        PROJECT_SOURCE_FILES.PROJECT_ID
FROM SOURCE_FILE, PROJECT_SOURCE_FILES
WHERE SOURCE_FILE.FILE_ID = PROJECT_SOURCE_FILES.FILE_ID AND
        PROJECT_SOURCE_FILES.PROJECT_ID = 11
        ORDER BY SOURCE_FILE.FILENAME
```

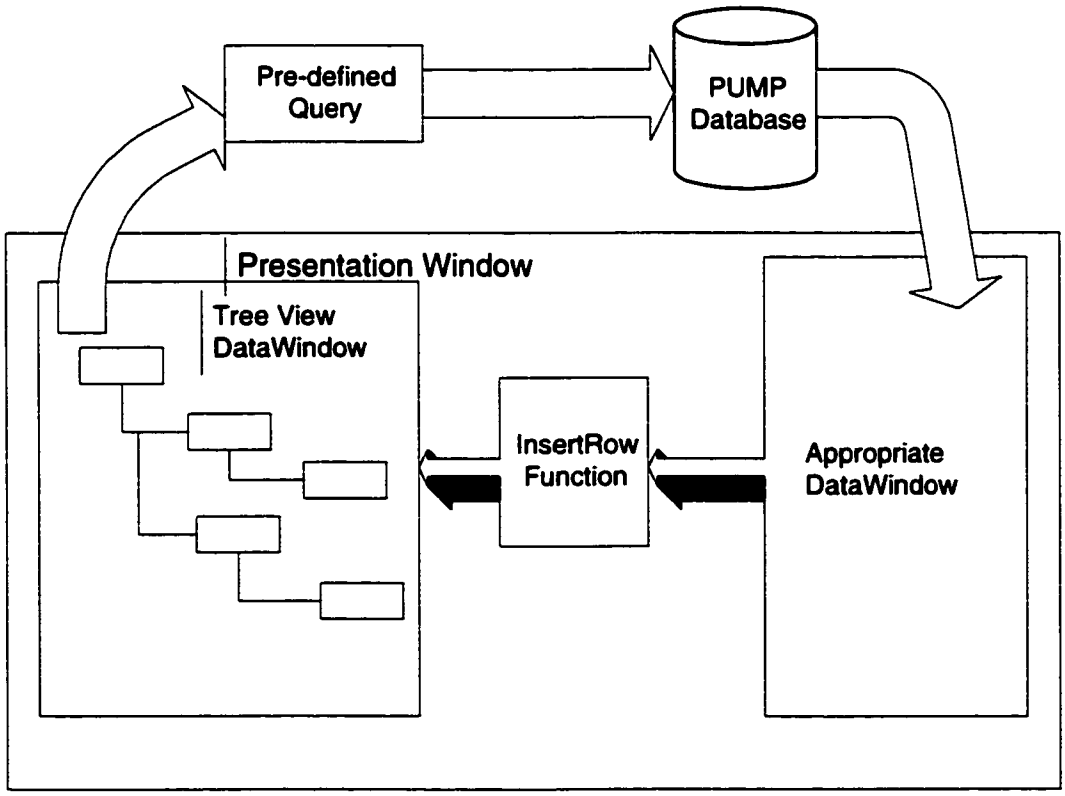


Figure 5.9 Expanding Process in the Tree View Datawindow

In the query above the project identification number passed was 11 which is the Airport project. The list of source code filenames is obtained by joining the two tables: Source_file Table and Project_source_file Table on the attribute *file_id*. This list is then passed to the Tree View datawindow. The *InsertRow* function then adds a new row to the Tree View datawindow for each of the source code filename as shown in Figure 5.9. This process then expands the Tree. The source code filename are indented to show the hierarchical view. When the user double-clicks on the project name again, the Tree is collapsed. This process is achieved passing the *project_id* and file_id from the Source File datawindow to the Tree View datawindow DeleteRow function. This function then finds the corresponding rows and deletes them from the Tree View Datawindow.

When the user clicks on the source code filename in the Tree View datawindow, detailed information about the source code file is displayed in Source Code Detail datawindow. This is achieved by obtaining the source code file identification number (*file_id*) from the selected source code filename and passing it to the Source Code Detail datawindow. The datawindow then executes the following SQL Query to obtain the relevant information:

```
SELECT FILE_ID, FILENAME, DATE_CREATED
FROM SOURCE_FILE
WHERE FILE_ID = 3
```

In the query above the source code identification number (*file_id*) passed was 3 which is the identification number assigned to the Mairport.mod source code file. The result of the query is shown in Figure 5.10. The process is shown in Figure 5.11.

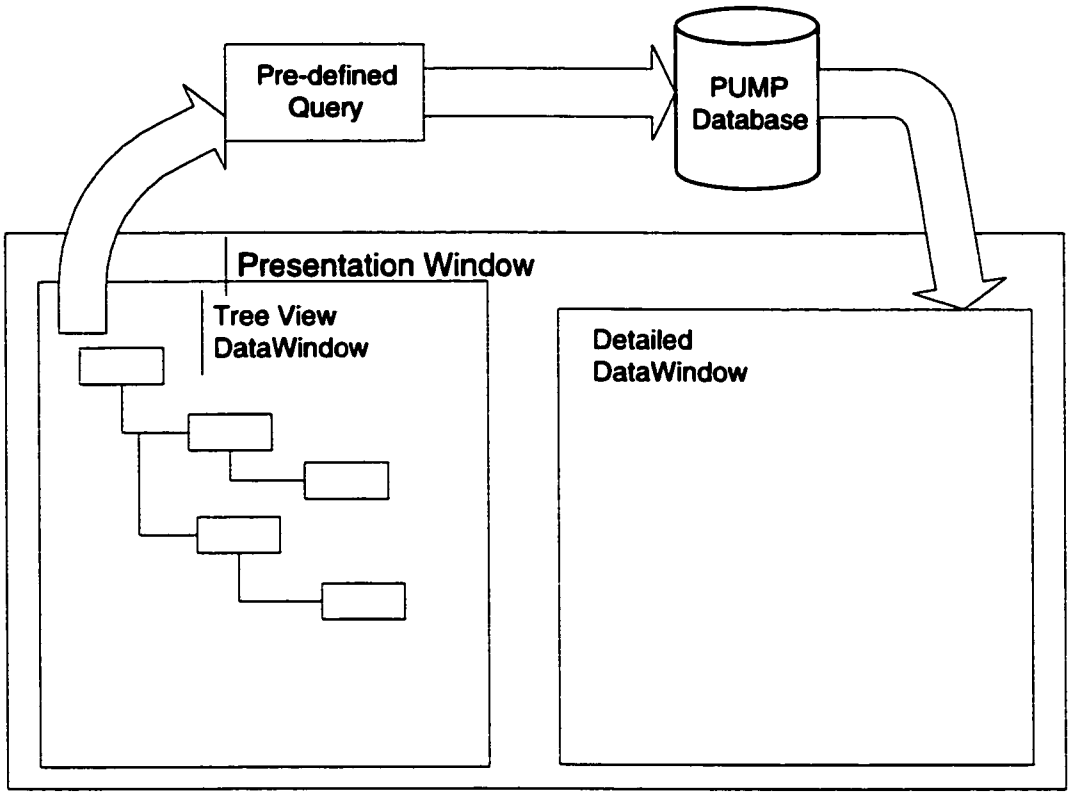


Figure 5.10 Process of Accessing the Detailed Information

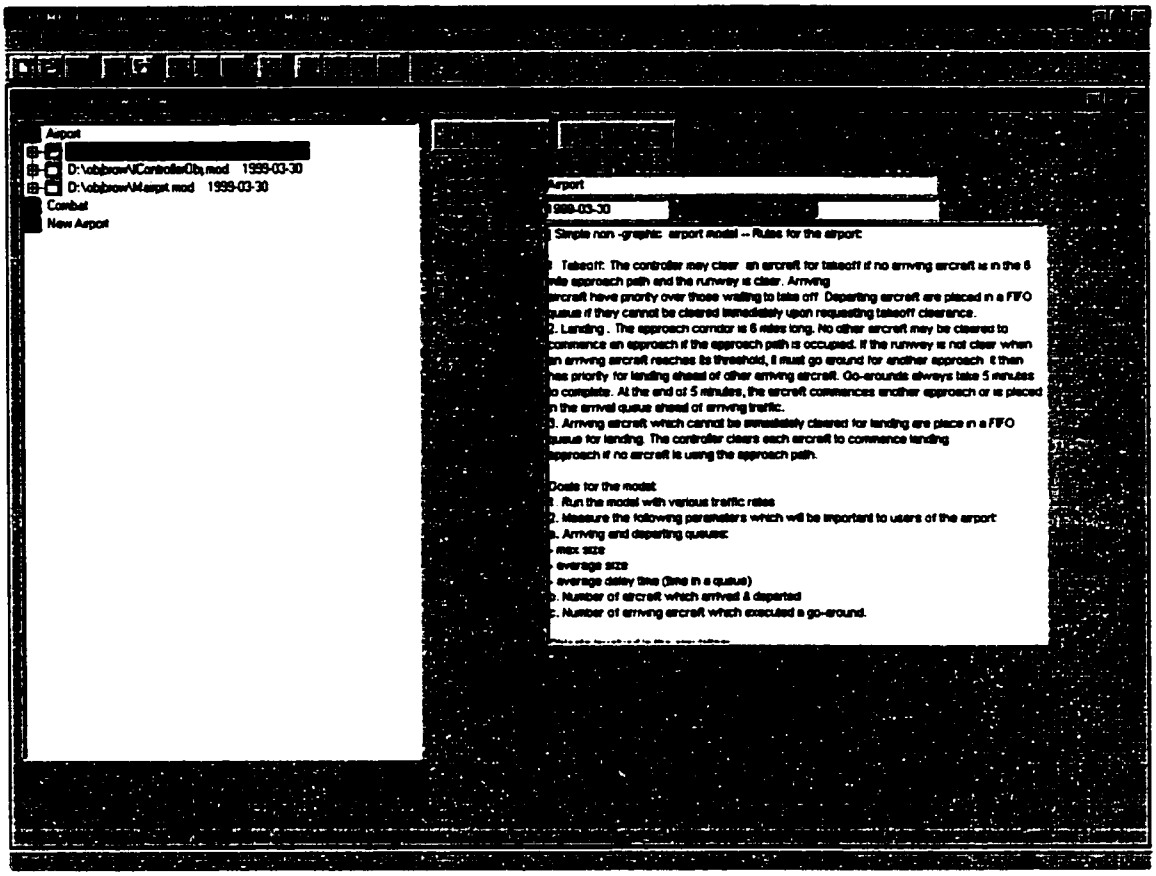


Figure 5.11 Source Code File Detail Information Displayed in Tree View Window

The user has the option of viewing the contents of the selected source code file by clicking on the 'Go To Source' command button which appears in the window. When the user clicks on this command button, the Source Code Viewer window is executed and the source code filename is passed to the window. The process is shown in Figure 5.12. The content of the source code file is displayed in the window as shown in Figure 5.13.

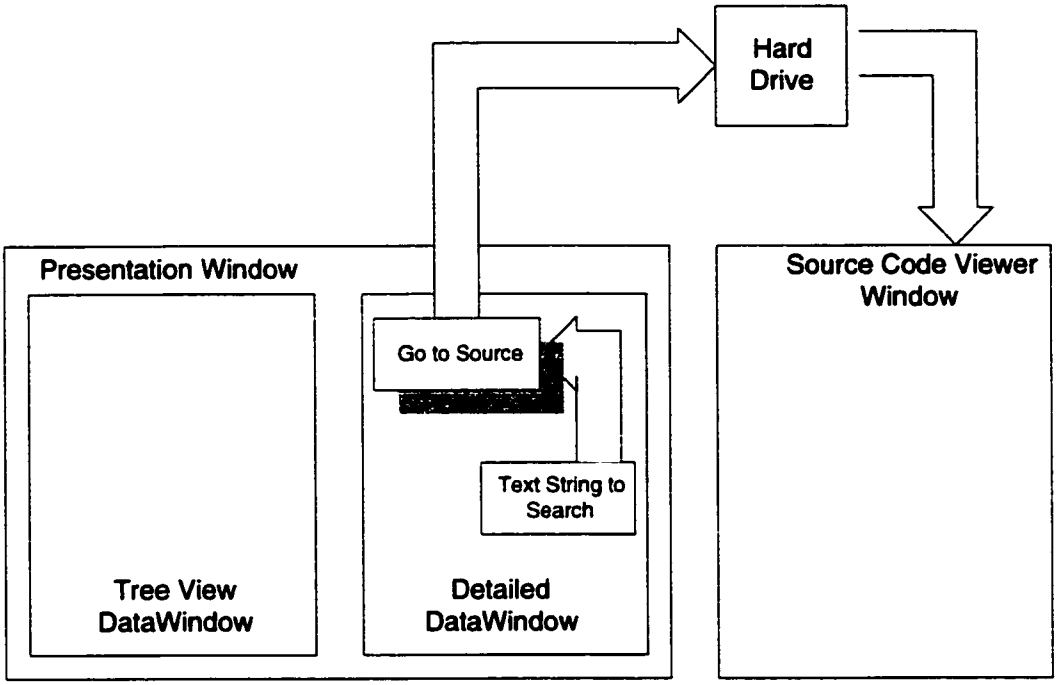


Figure 5.12 Process of viewing the Source Code

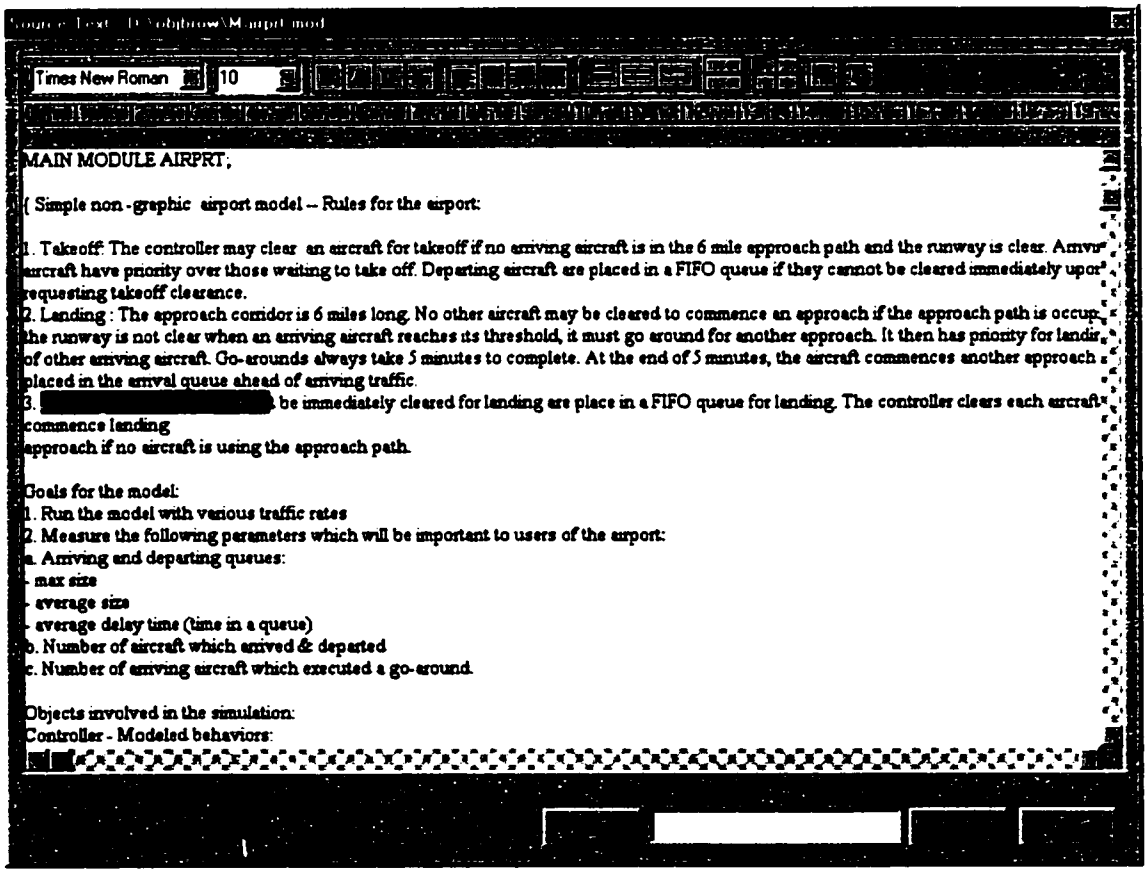


Figure 5.13 Contents of Source Code File Displayed in Source Code Viewer Window

5.9.1.3 Module Names

This is the level three information for a MODSIM Project. When the user double-clicks on the source code filename, PUMP displays all the modules that are in the file. This is achieved by obtaining the file identification number (*file_id*) of the selected source code file and passing it to the Module datawindow. This datawindow returns a list of module names in alphabetical order by executing the SQL Query shown below:

```
SELECT MODULE_INFORMATION.MODULE_ID,
       MODULE_INFORMATION.MODULE_NAME,
       MODULE_INFORMATION.MAIN_MODULE,
       SOURCE_FILES.PROJECT_ID
FROM MODULE_INFORMATION, SOURCE_FILE
WHERE MODULE_INFORMATION.FILE_ID = SOURCE_FILE.FILE_ID AND
       SOURCE_FILES.FILE_ID  = 3
       ORDER BY MODULE_INFORMATION.MODULE_NAME
```

In the query above, the file identification number (*file_id*) passed was 3 which is the Mairport.mod source code file. The list of module names is obtained by joining the Module_information Table and the Source_file Table on the *file_id* attribute. This list is the passed to the Tree View datawindow. The *InsertRow* function then adds a new row below the source code filename, for each of the Module names in the Tree View datawindow. This process expands the Tree. The Module names are indented to show the hierarchical view. The Main Module is indicated by displaying the word 'Main Module' beside the module name. When the user double-clicks on the source code filename again, the Tree is collapsed. This process is carried out by passing the *module_id* and *file_id* from the Module datawindow to the Tree View datawindow *DeleteRow* function. This function then finds the corresponding rows and deletes them from the Tree View Datawindow.

When the user clicks on the module name in the Tree View datawindow, detailed information about the module is displayed in Module Detail datawindow. This is achieved by obtaining the module identification number (*module_id*) from the selected module name and passing it to the Module Detail datawindow. The datawindow then executes the following a SQL Query to obtain all the relevant information about the module.

```
SELECT MODULE NAME, MODULE_TYPE
FROM MODULE_INFORMATION
WHERE MODULE_ID = 1
```

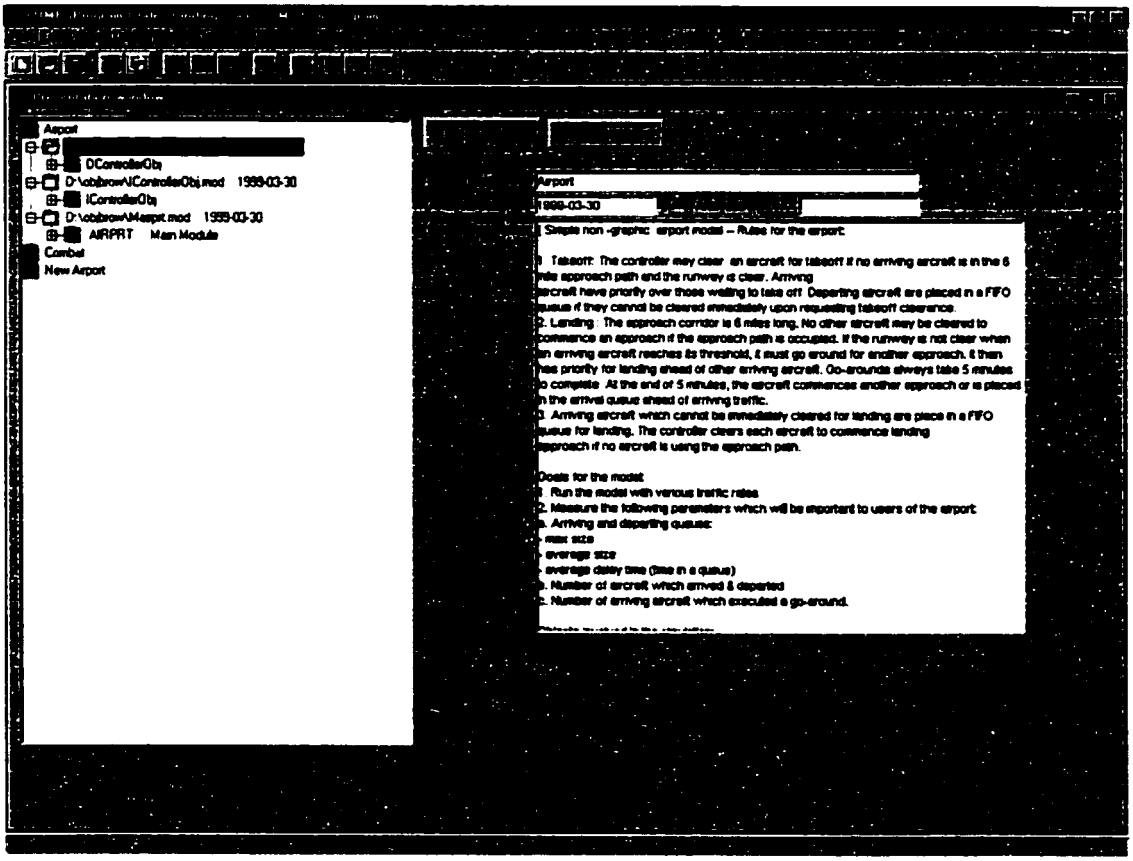


Figure 5.14 Module Detail Information Displayed in Tree View Window

In the query above the module identification number (*module_id*) passed was 1 which is the Airport Main Module. The information which is presented is shown in Figure 5.14.

The user can find the beginning of the Airport Main Module in the source code file by clicking on the 'Go To Source' command button. When the user clicks on this command button, the Source Code Viewer window is executed and the source code filename and the module name are passed to the window. The contents of the source code file are displayed in the window and the beginning of Airport Main Module is highlighted as shown in Figure 5.15.

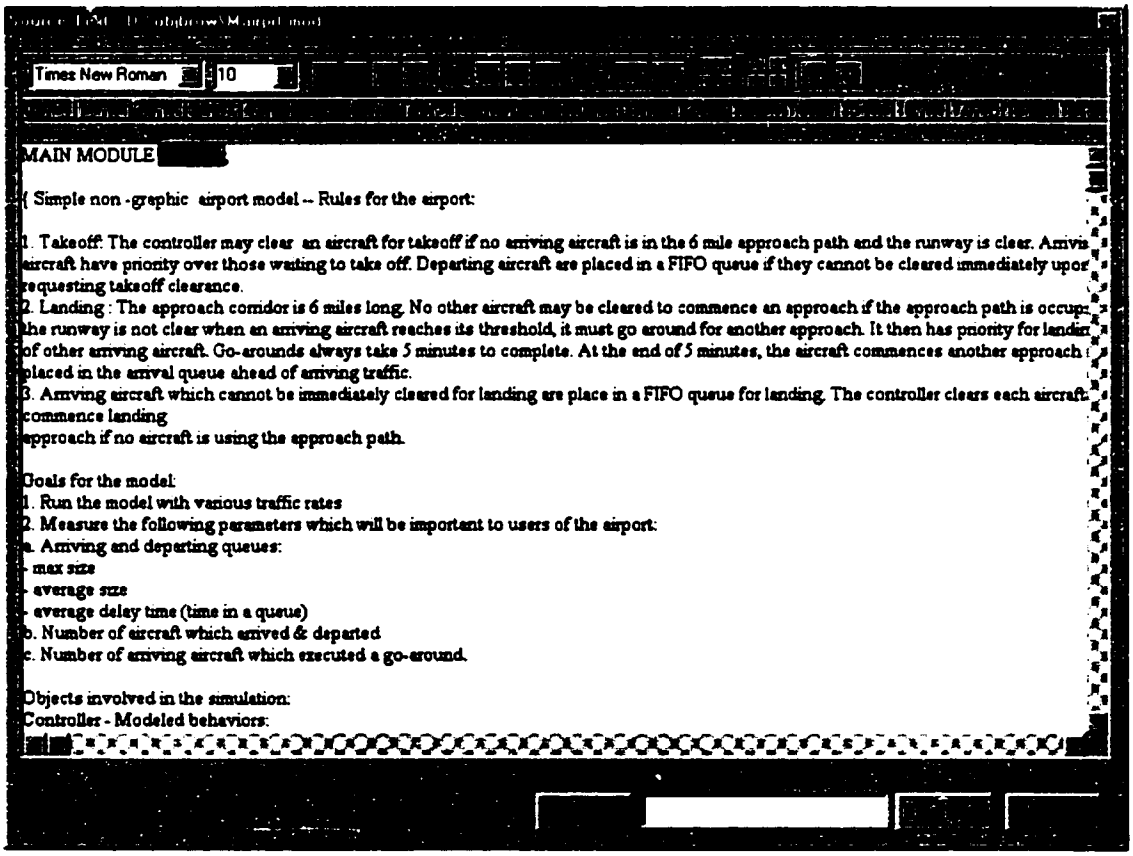


Figure 5.15 Airport Main Module Displayed in Source Code Viewer Window

5.9.1.4 Globals Names

This is the level four information for a MODSIM Project. When the user double-clicks on the module name, PUMP displays all the Globals that are in selected module. This is achieved by obtaining the module identification number (*module_id*) for the selected module and passing it to the to Globals datawindow. This datawindow returns a list of globals names, arranged in alphabetical order, by executing the SQL Query shown below:

```
SELECT GLOBALS_DETAIL.GLOBALS_ID,
        GLOBALS_DETAIL.GLOBALS_NAME,
        GLOBALS_DETAIL.GLOBALS_DETAILS, GLOBALS_DETAIL.ENTITY_ID,
        MODULE_INFORMATION.MODULE_ID, ENTITY_TYPE.ENTITY_NAME
FROM GLOBALS_DETAIL, MODULE_INFORMATION, ENTITY_TYPE
WHERE GLOBALS_DETAIL.MODULE_ID = MODULE_INFORMATION.MODULE_ID AND
        GLOBALS_DETAIL.ENTITY_ID = ENTITY_TYPE.ENTITY_ID AND
        GLOBALS_DETAIL.MODULE_ID  = 1 AND
        GLOBALS_DETAIL.GLOBALS_ID NOT IN (SELECT
OBJECT_INHERITANCE.OBJECT_ID FROM OBJECT_INHERITANCE)
        ORDER BY GLOBALS_DETAIL.GLOBALS_NAME
```

In the query above, the module identification number (*module_id*) passed was 1 which is the Airport Main Module. The list of globals names is obtained by creating two table joins:

1.) Globals_detail Table and Module_information Table on the *module_id* attribute

2.) Globals_detail Table and Entity_type Table on the *entity_id* attribute. This join is created to obtain the entity name of each globals from the Entity_type Table.

Also in the query above, a subquery is required to make sure that the list of globals only contains globals that are not inherited by other user-defined objects. This list is the passed to the Tree View datawindow. The *InsertRow* function adds a new row below the module

name, for each of the Globals names in the Tree View datawindow and this process expands the Tree. The Globals names are indented to show the hierarchical view and the type of each Global is indicated. When the user double-clicks on the source code filename a second time the Tree is collapsed. This process is carried out by passing the *globals_id* and *module_id* from the Globals datawindow to the Tree View datawindow *DeleteRow* function. This function then finds the corresponding rows and deletes them from the Tree View datawindow.

When the user clicks on the globals name in the Tree View datawindow, detailed information about the Globals is displayed in Globals Detail datawindow. This is achieved by obtaining the globals identification number (*globals_id*) for the selected globals name and passing it to the Globals Detail datawindow. The datawindow then executes the following a SQL Query to obtain the relevant information:

```
SELECT GLOBALS_DETAIL.GLOBALS_NAME,
        GLOBALS_DETAIL.GLOBALS_DETAILS, GLOBALS_DETAIL.ENTITY_ID,
        ENTITY_TYPE.ENTITY_NAME
FROM GLOBALS_DETAIL, ENTITY_TYPE
WHERE GLOBALS_DETAIL.ENTITY_ID = ENTITY_TYPE.ENTITY_ID AND
        GLOBALS_DETAIL.GLOBALS_ID  = 22
```

In the query above the globals identification number (*globals_id*) passed is 22 which is the AircraftObj Object. The information presented to the user is shown in Figure 5.16.

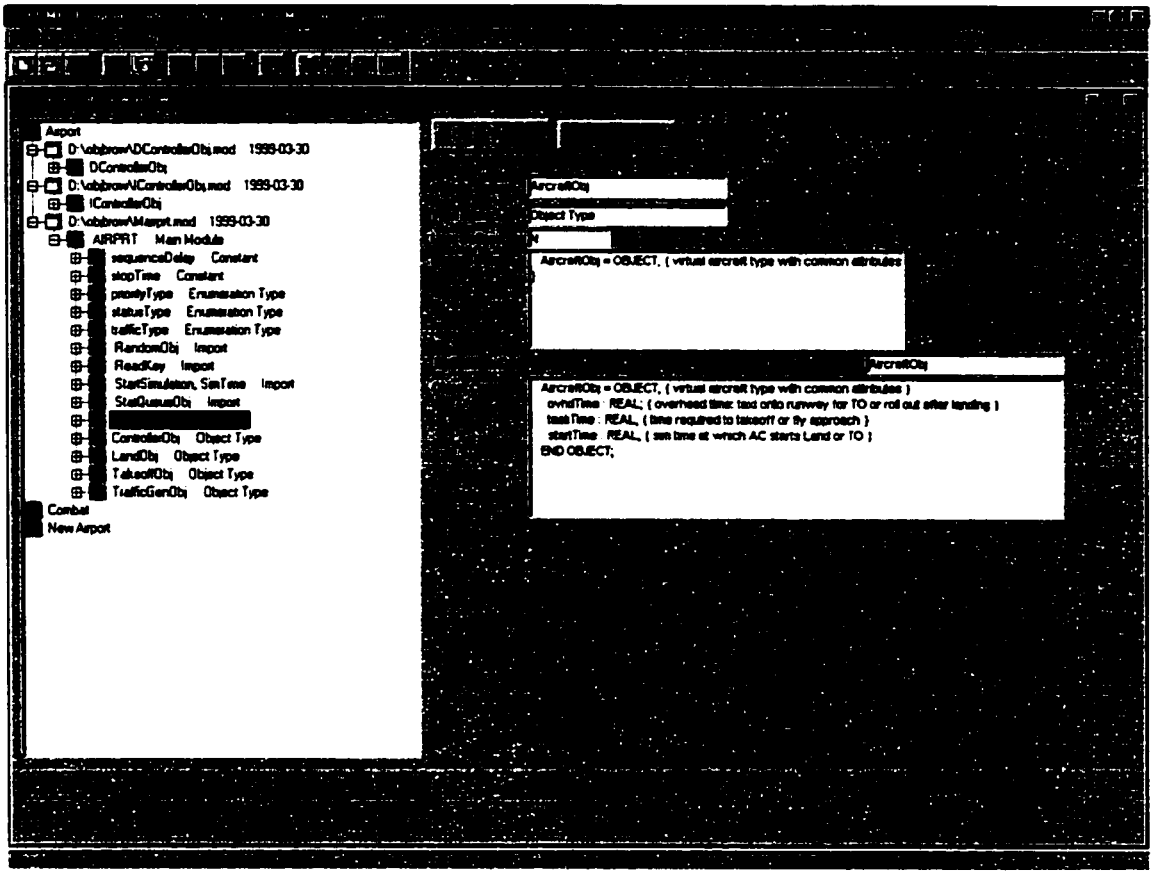


Figure 5.16 Globals Detail Information Displayed in Tree View Window

The user can find the first occurrence of the AircraftObj in the source code file by clicking on the 'Go To Source' command button. When the user clicks on this command button, the Source Code Viewer window is executed and the source code filename and the globals name are passed to the window.

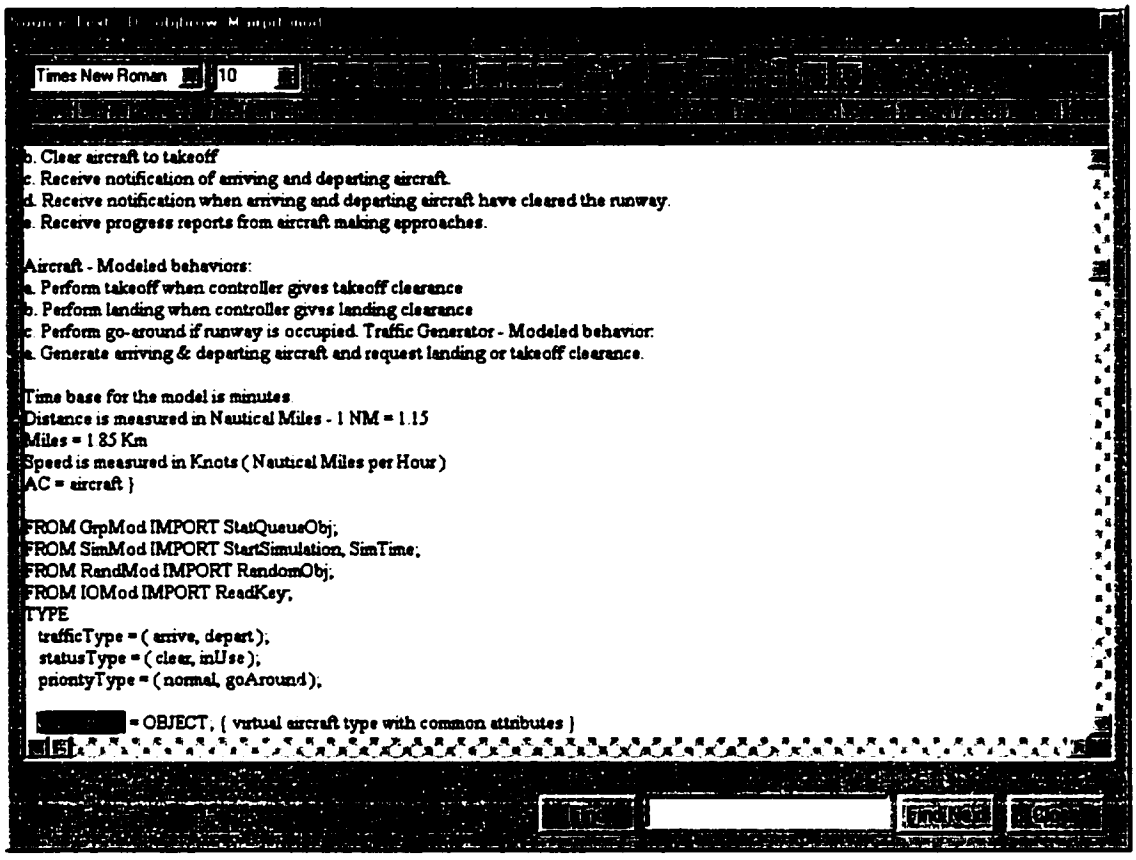


Figure 5.17 AircraftObj Displayed in Source Code Viewer Window

The contents of the source code file are displayed in the window and the first occurrence of AircraftObj is highlighted, as shown in Figure 5.17. To find the subsequent occurrences of AircraftObj, the user can click on the 'Find Next' command button.

5.9.1.5 Component Names

This is the level five information for a MODSIM Project. When the user double-clicks on the globals name, PUMP displays all the components that are in selected global. This is achieved by obtaining the globals identification number (*globals_id*) of the selected module and passing it to the Component datawindow. Only user-defined objects will contain components. All of these components have been declared locally within the

selected global. In some occasion, the selected global may contain objects that has been inherited from the global. When globals_id is passed to the Component datawindow, it returns a list of component names in alphabetical order by executing SQL Query as shown below:

```
SELECT COMPONENT_DETAIL.COMPONENT_DETAIL_ID,
       COMPONENT_DETAIL.COMPONENT_NAME COMPONENT_NAME,
       COMPONENT_DETAIL.ENTITY_ID,
       ENTITY_TYPE.ENTITY_NAME
FROM GLOBALS_DETAIL, COMPONENT_DETAIL, ENTITY_TYPE
WHERE COMPONENT_DETAIL.GLOBALS_ID = GLOBAL_DETAILS.GLOBALS_ID
       COMPONENT_DETAIL.ENTITY_ID = ENTITY_TYPE.ENTITY_ID AND
       GLOBAL_DETAILS.GLOBALS_ID  = 1
UNION
SELECT OBJECT_INHERITANCE.OBJECT_ID,
       GLOBALS_DETAIL.GLOBAL_NAME COMPONENT_NAME,
       GLOBALS_DETAIL.ENTITY_ID
       ENTITY_TYPE.ENTITY_NAME
FROM GLOBALS_DETAIL, COMPONENT_DETAIL, ENTITY_TYPE
WHERE OBJECT_INHERITANCE.OBJECT_ID = GLOBAL_DETAILS.GLOBALS_ID
       GLOBALS_DETAIL.ENTITY_ID = ENTITY_TYPE.ENTITY_ID AND
       OBJECT_INHERITANCE.OBJECT_PARENT  = 1
ORDER BY GLOBAL_DETAIL.COMPONENT_NAME
```

In the query above, the globals identification number (globals_id) passed was 1 which is the AircraftObj. The list of component names is obtained by using a union of two queries. The first query will return a list of components name from the Component Detail Table. Two joins are used in this query:

1.) Globals_detail Table and Component_detail Table are joined using the *globals_id* attribute.

2.) Component_detail Table and Entity_type Table are joined using the *entity_id* attribute. This join is created to obtain the entity name of each component.

The second query will return a list of object names from the Global_detail Table that are inherited from the selected global. Two joins are used in this query:

1.) Globals_detail Table and Component_detail Table are joined using the *globals_id* attribute.

2.) Component_detail Table and Entity_type Table are joined using the *entity_id* attribute. This join is created to obtain the entity name of each component.

This list is the passed to the Tree View datawindow. The *InsertRow* function then adds a new row below the globals name, for each of the Component names in the Tree View datawindow. This process then expands the Tree. The Component names are indented to show the hierarchical view and the type of each component is indicated. When the user double-clicks on the source code filename again, the Tree is collapsed. This process is achieved passing the *component_detail_id* and *globals_id* from the Component datawindow to the Tree View datawindow *DeleteRow* function. This function then finds the corresponding rows and deletes them from the Tree View datawindow.

When the user clicks on the component name in the Tree View datawindow, detail information of the component is displayed in Component datawindow. This is achieved by obtaining the component detail identification number (*component_detail_id*) or the global identification number (*globals_id*) from the selected component name and passing it to the Component datawindow. The datawindow then executes the following a SQL Query to obtain all the relevant information on the component. The SQL Query is shown below:

```
SELECT COMPONENT_DETAIL.COMPONENT_NAME,
        COMPONENT_DETAIL.CODE_FRAGMENT, COMPONENT_DETAIL.ENTITY_ID
        ENTITY_TYPE.ENTITY_NAME
FROM COMPONENT_DETAIL, ENTITY_TYPE
WHERE COMPONENT_DETAIL.ENTITY_ID = ENTITY_TYPE.ENTITY_ID AND
        COMPONENT_DETAIL.COMPONENT_DETAIL_ID  = 34
UNION
SELECT GLOBALS_DETAIL.GLOBALS_NAME,
        GLOBALS_DETAIL.GLOBALS_DETAILS,GLOBALS_DETAIL.ENTITY_ID,
        ENTITY_TYPE.ENTITY_NAME
FROM GLOBALS_DETAIL, ENTITY_TYPE
WHERE GLOBALS_DETAIL.ENTITY_ID = ENTITY_TYPE.ENTITY_ID AND
        GLOBALS_DETAIL.GLOBALS_ID  = 34
```

Again, in the query above, a union of two queries are used. The component detail identification number (*component_detail_id*) or global identification number (*globals_id*) passed was 34 which is TakeoffObj. This object has been inherited from AircraftObj. The detail information is shown in figure 5.18. This includes the Definition and the Implementation Code for a particular user-defined object.

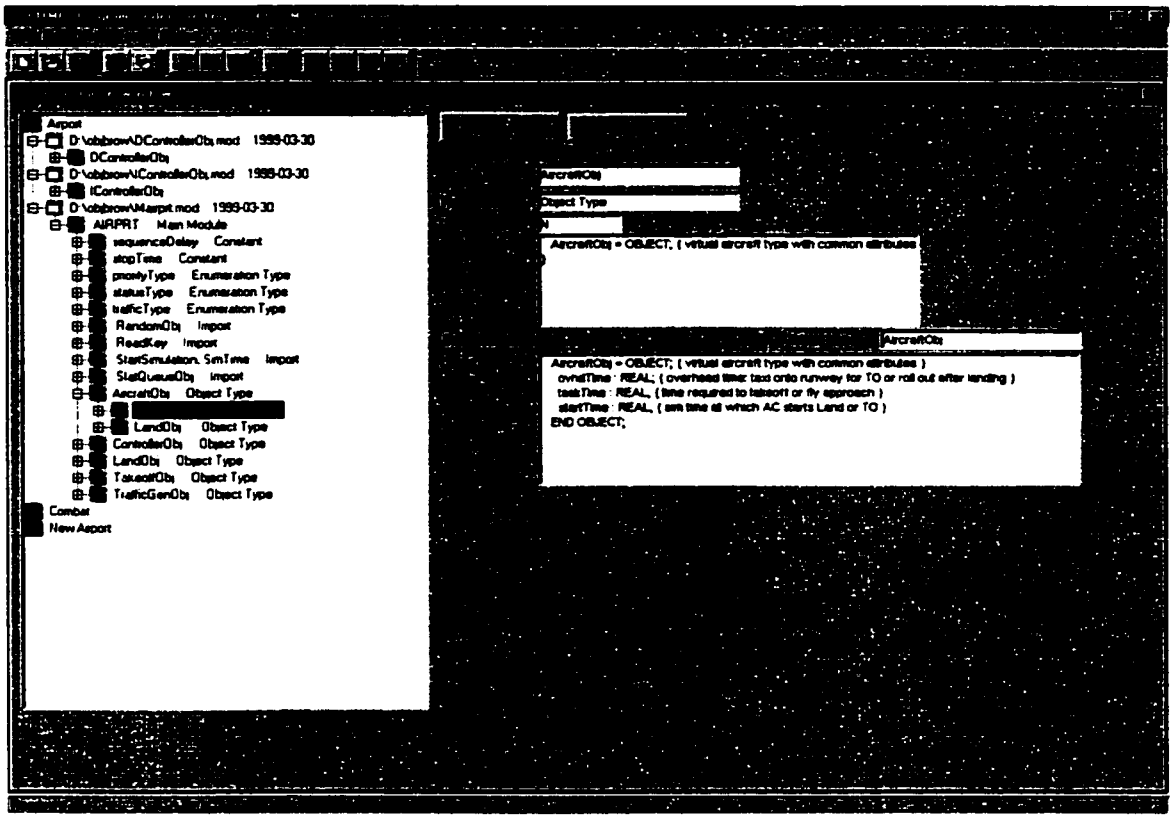


Figure 5.18 Component Detail Information Displayed in Tree View Window

The user can find the first occurrence of the string 'TakeoffObj' in the source code file by clicking on the 'Go To Source' command button. When the user clicks on this command button, the Source Code Viewer window is executed and the source code filename and the component name are passed to the window. The content of the source code file is displayed on the window and the occurrence of 'TakeOffObj' is highlighted as shown in figure 5.19. To find the subsequent occurrences of 'TakeOffObj', the user can click on the 'Find Next' command button.

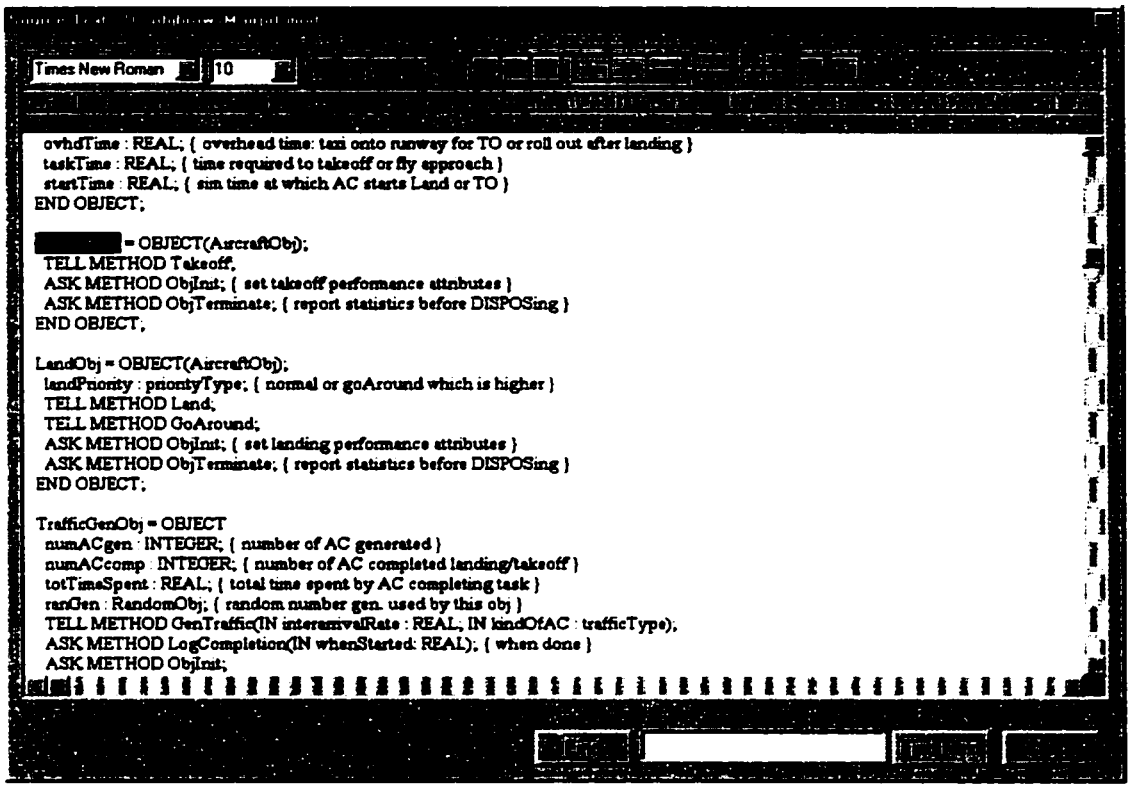


Figure 5.19 'TakeoffObj' Displayed in Source Code Viewer Window

5.9.1.6 Component Details

This is the level six information for a MODSIM Project. When the user double-clicks on the component name, PUMP displays all the components that are in selected component. This is achieved by obtaining the component detail identification number (*component_detail_id*) or global identification number (*globals_id*) of the selected component and passing it to the Component Detail datawindow. This datawindow returns a list of component names in alphabetical order by executing SQL Query as shown below:

```
SELECT COMPONENT_DETAIL.COMPONENT_DETAIL_ID,
       COMPONENT_DETAIL.COMPONENT_NAME COMPONENT_NAME,
       COMPONENT_DETAIL.ENTITY_ID,
       ENTITY_TYPE.ENTITY_NAME
FROM GLOBALS_DETAIL, COMPONENT_DETAIL, ENTITY_TYPE
WHERE COMPONENT_DETAIL.GLOBALS_ID = GLOBAL_DETAILS.GLOBALS_ID
       COMPONENT_DETAIL.ENTITY_ID = ENTITY_TYPE.ENTITY_ID AND
       GLOBAL_DETAILS.GLOBALS_ID  = 34
UNION
SELECT OBJECT_INHERITANCE.OBJECT_ID,
       GLOBALS_DETAIL.GLOBAL_NAME COMPONENT_NAME,
       GLOBALS_DETAIL.ENTITY_ID
       ENTITY_TYPE.ENTITY_NAME
FROM GLOBALS_DETAIL, COMPONENT_DETAIL, ENTITY_TYPE
WHERE OBJECT_INHERITANCE.OBJECT_ID = GLOBAL_DETAILS.GLOBALS_ID
       GLOBALS_DETAIL.ENTITY_ID = ENTITY_TYPE.ENTITY_ID AND
       OBJECT_INHERITANCE.OBJECT_PARENT  = 34
       ORDER BY GLOBAL_DETAIL.COMPONENT_NAME
```

In the query above, the globals identification number (globals_id) passed was 34 which is the TakeoffObj. The list of component names is obtained by using a union of two queries. The first query will return a list of components name from the Component Detail Table. Two joins are used in this query:

1.) Globals_detail Table and Component_detail Table are joined using the *globals_id* attribute.

2.) Component_detail Table and Entity_type Table are joined using the *entity_id* attribute. This join is created to obtain the entity name of each component.

The second query will return a list of object names from the Global_detail Table that are inherited from the selected global. Two joins are used in this query:

1.) Globals_detail Table and Component_detail Table are joined using the *globals_id* attribute.

2.) Component_detail Table and Entity_type Table are joined using the *entity_id* attribute. This join is created to obtain the entity name of each component.

This list is the passed to the Tree View datawindow. The *InsertRow* function then adds a new row below the globals name, for each of the Component names in the Tree View datawindow. This process then expands the Tree. The Component names are indented to show the hierarchical view and the type of each component is indicated. When the user double-clicks on the source code filename again, the Tree is collapsed. This process is achieved passing the *component_detail_id* and *globals_id* from the Component Detail datawindow to the Tree View datawindow *DeleteRow* function. This function then finds the corresponding rows and deletes them from the Tree View datawindow. Figure 5.20 shows the Component Details List.

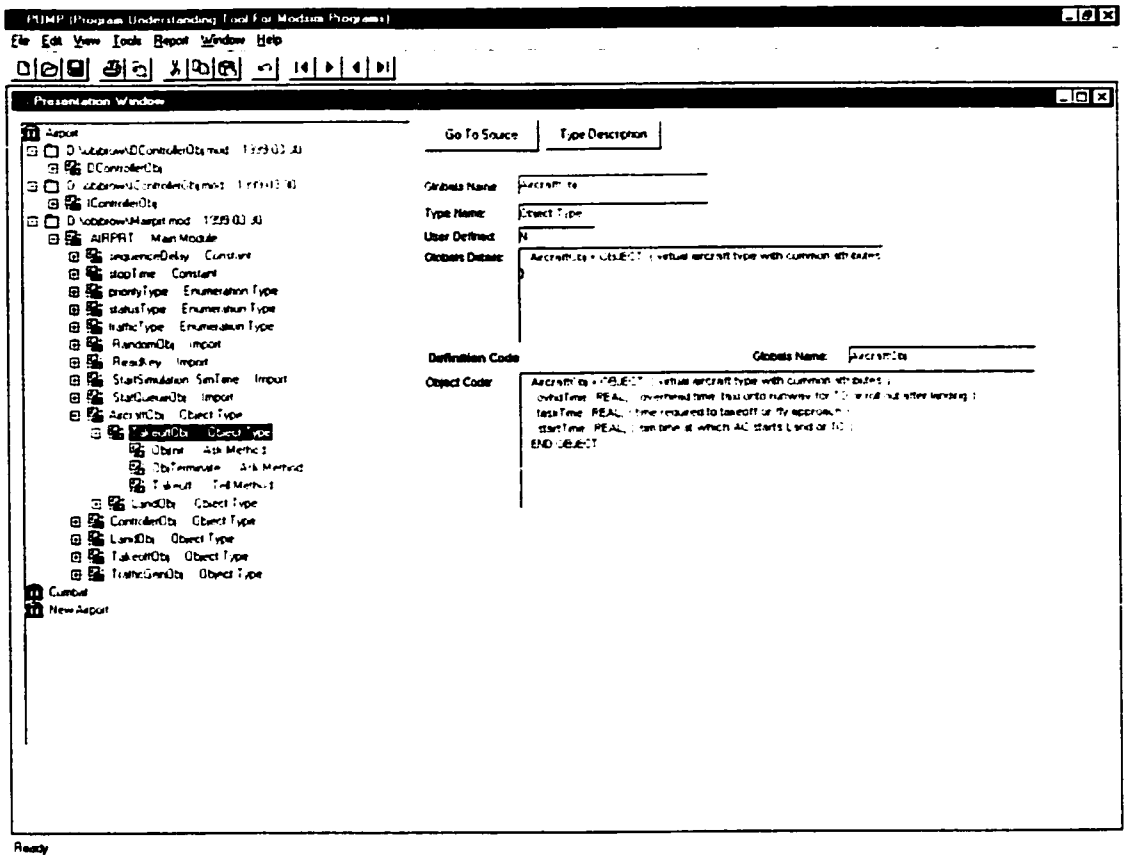


Figure 5.20 Example of the Component Details List

The rest of the functionality for this level is similar to level five functionality. This includes the Component Detail and the 'Go To Source'. Any levels below this level will have the same functionality as level 6. The same datawindows are used to retrieve information relevant to the appropriate level. Recursive calls are made to extend the levels to the level of the last level of inheritance for a particular user-defined object.

MODSIM allows an object to be inherited from more that one object. This is called multiple inheritance. PUMP will indicate to the user if an object have been inherited from more than one object. When the user selects a component name at level 5 and below and if the component have been inherited from more that one object, then

PUMP will display a list of the parent objects in Component Detail datawindow in the right pane of the Presentation window. PUMP will also indicate if a method has been overridden in the component selected. This is shown in Figure 5.21.

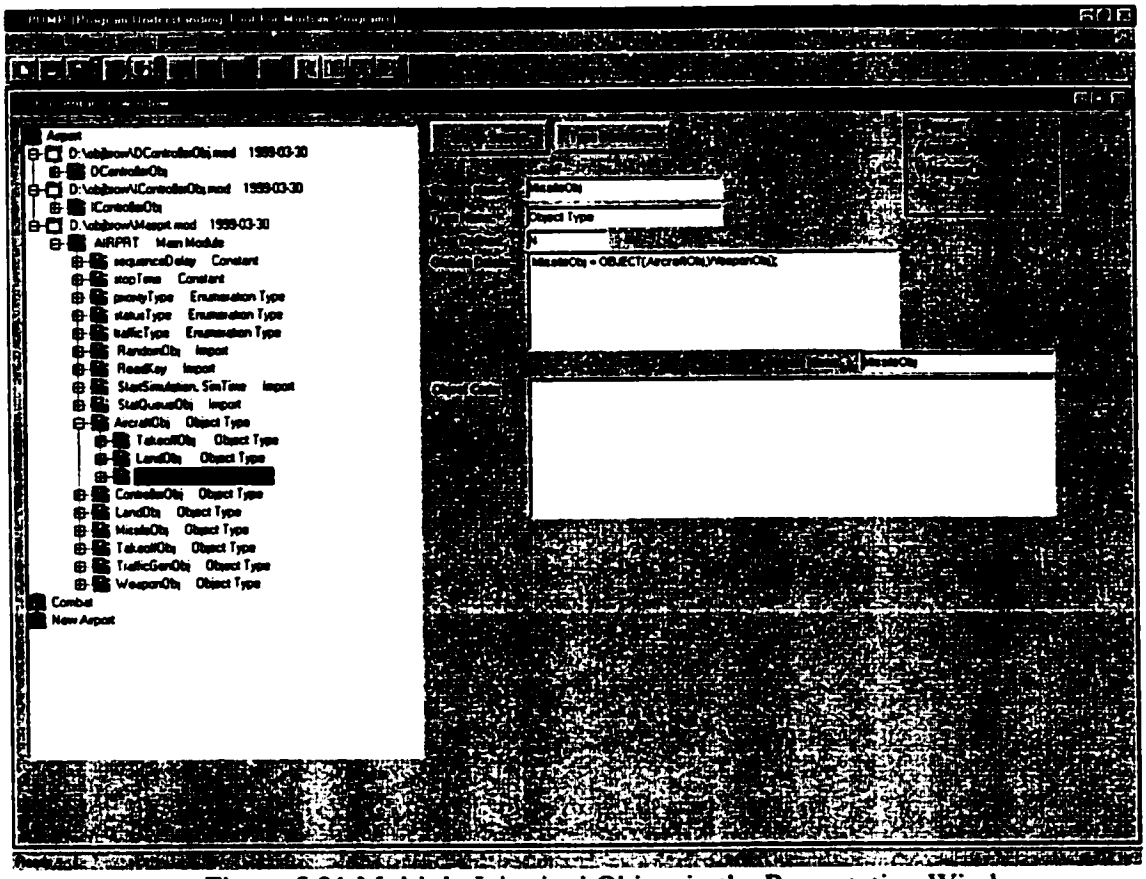


Figure 5.21 Multiple Inherited Object in the Presentation Window

It is shown that MissileObj is inherited from AircraftObj and WeaponObj.

5.9.2 Export Function

PUMP allows a user to export code for a particular object. This feature will help the sharing of objects with other developed projects. This is process is carried out by the selecting an object from the Tree View datawindow in the Presentation Window and clicking on the 'Export' command button on the right pane of the Presentation Window.

When the user selects an object, PUMP will search for the object in the PUMP database and export the object into a text file.

5.10 Evaluation of PUMP

In this section, two examples are provided to illustrate the nature of the support that PUMP can provide within the context of MODSIM program maintenance.

a.) A simple maintenance task is the modification of the value of some program constant. We consider here increasing the value of the constant *stopTime* from 1440.0 to 2440.0 . The following steps are performed to carry out this change.

1.) Search for the constant *stopTime* in the Tree View datawindow using the Find feature provided by PUMP. Once located the details for *stopTime* are presented in the Detailed datawindow as shown in Figure 5.22.

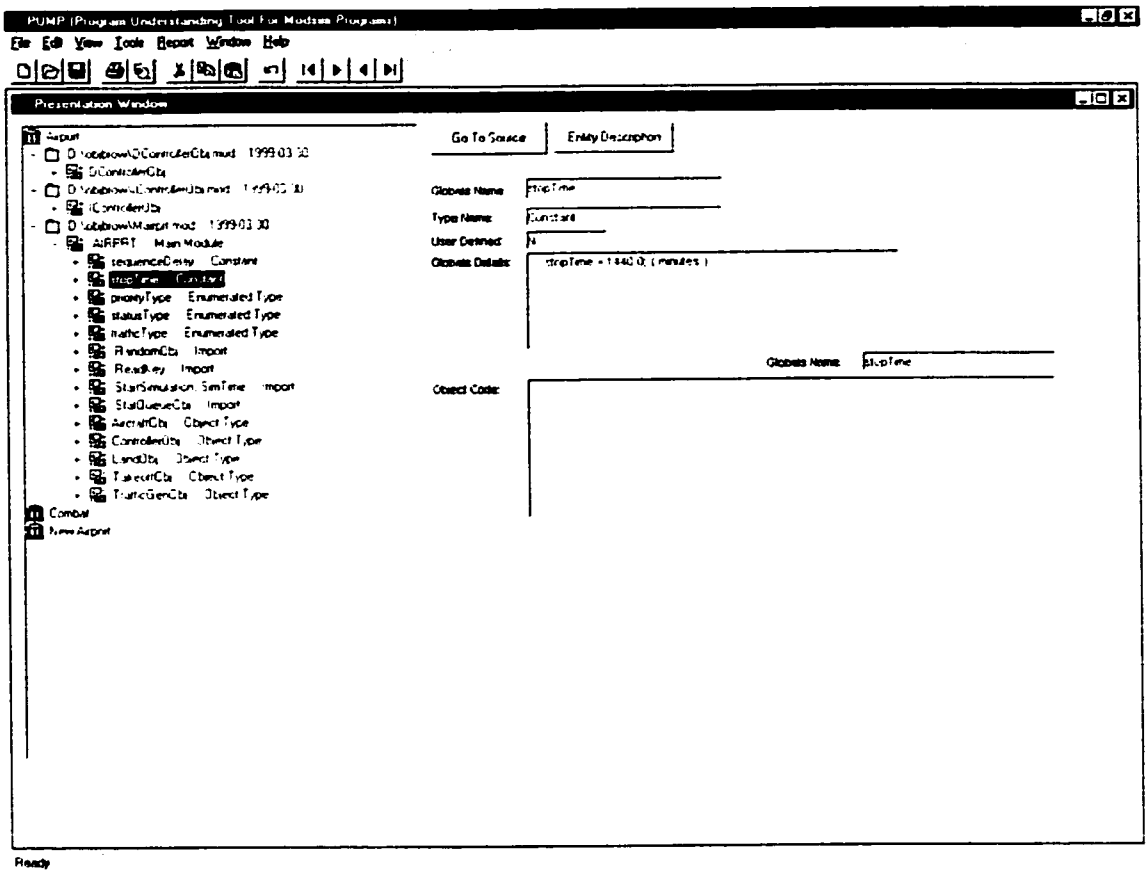


Figure 5.22 The value of constant *stopTime* before the change.

2.) Using the 'Go To Source' command button, the user locates the string '*stopTime*'. The user then changes the value of *stopTime* from 1440.0 to 2440.0 and saves the change to the source code file.

3.) The user can then re-analyze the project Airport. The updated value of constant *stopTime* is now stored in the database as shown in Figure 5.23. PUMP is thus able to assist a novice programmer to perform this task both quickly and correctly.

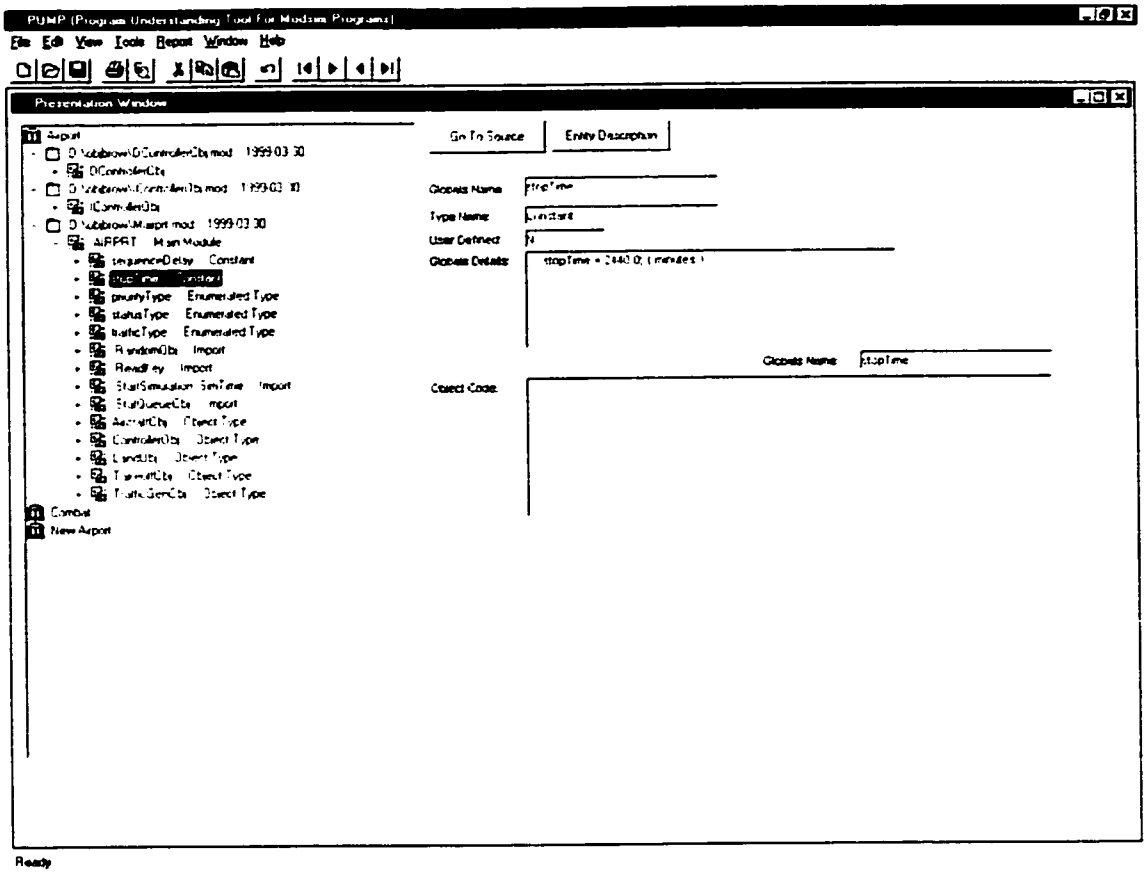


Figure 5.23 The value of constant *stopTime* after the change.

b.) The task to be carried out here is a change to the Airport model. The change is the addition of a second runway. The following steps are performed to carry out this change:

1.) Search for the variable *runway* in the Tree View datawindow using the Find feature provided by PUMP. The variable runway is selected in the Tree View datawindow and the runway details are presented in Detailed datawindow. The tree node for the variable runway is expanded show all the child nodes as show in Figure 5.24. It is noted that the variable *runway* is an instance of the enumerated variable of type *statusType*. It is found in three methods:

   a.) ASK Method TakeoffClerance and TELL Method ClearOfRunway in ControllerObj object.

   b.) TELL Method Land in LandObj object.

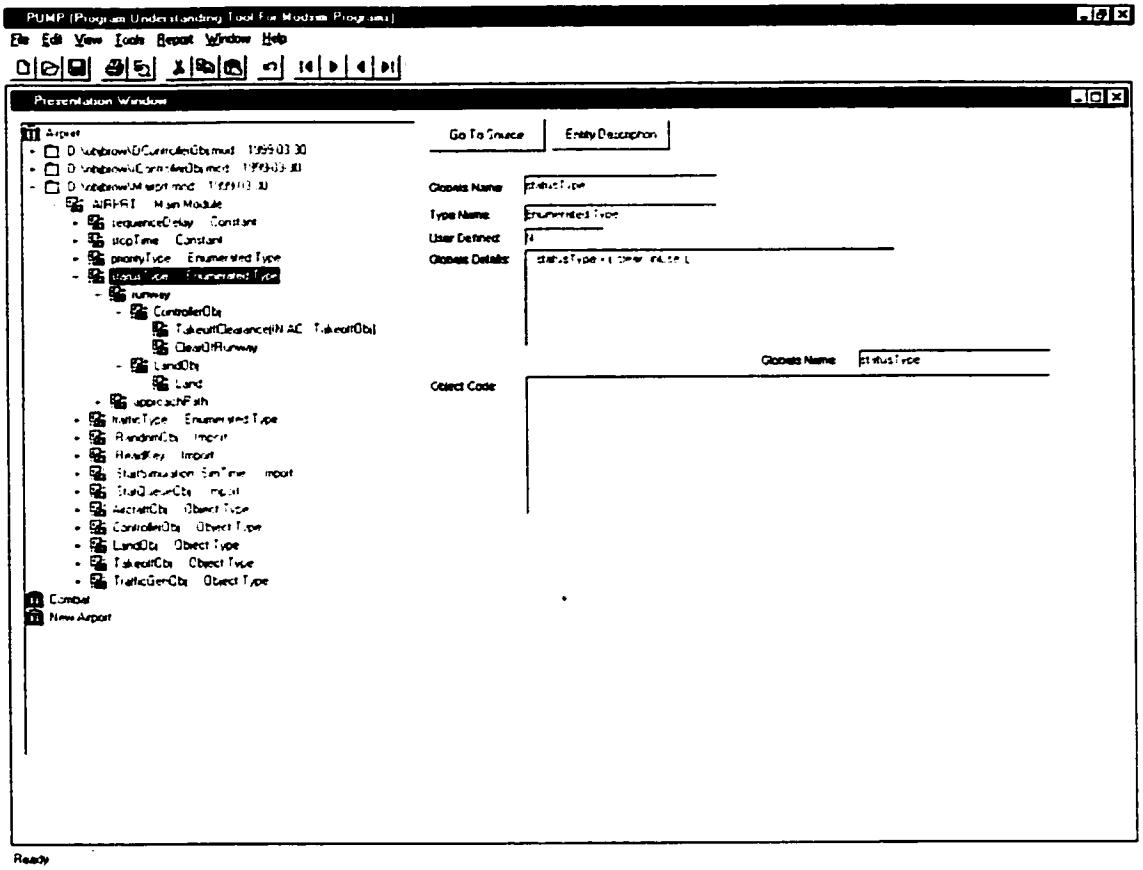


Figure 5.24 All the child nodes for variable *runway*.

2.) Using the 'Go To Source' command button, the user can locate the string 'runway' and add a new variable runway2 of type statusType on the next line. Also using the 'Go To Source' command button, the user can locate the strings 'ASK Method TakeoffClerance', 'TELL Method ClearOfRunway' and 'TELL Method Land' and insert into each the appropriate code for *runway2*. All such changes would be saved.

3.) The user can then re-analyze the project Airport. The changes are reflected in the presentations provided by PUMP as shown in Figure 5.25.

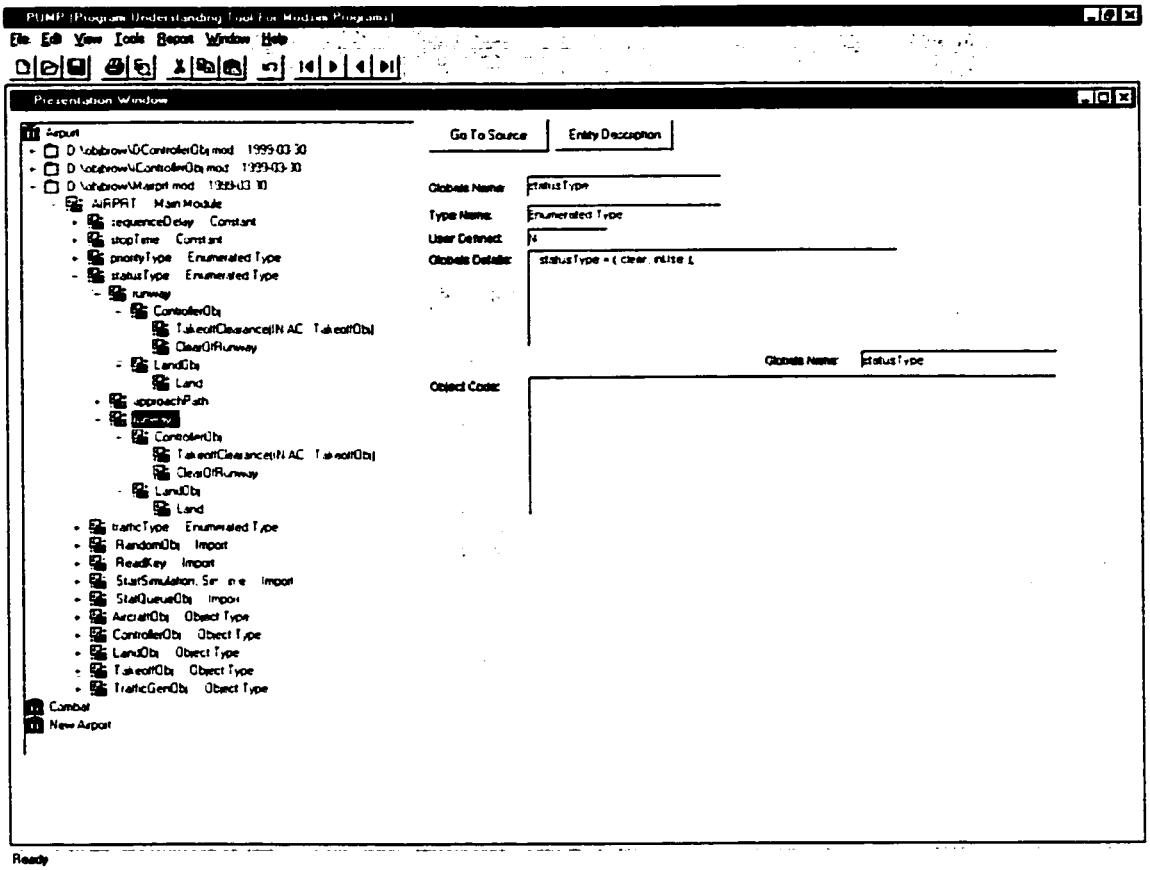


Figure 5.25 Show all the child node for *runway2*.

This illustrated how PUMP can assist a novice programmer in carry out MODSIM program enhancements.

MODSIM supports multiple inheritance (i.e., an object can be inherited from more that one object). PUMP will indicate to the user if an object has been inherited from more than one object. This inheritance is shown in the Detailed datawindow Figure 5.26.

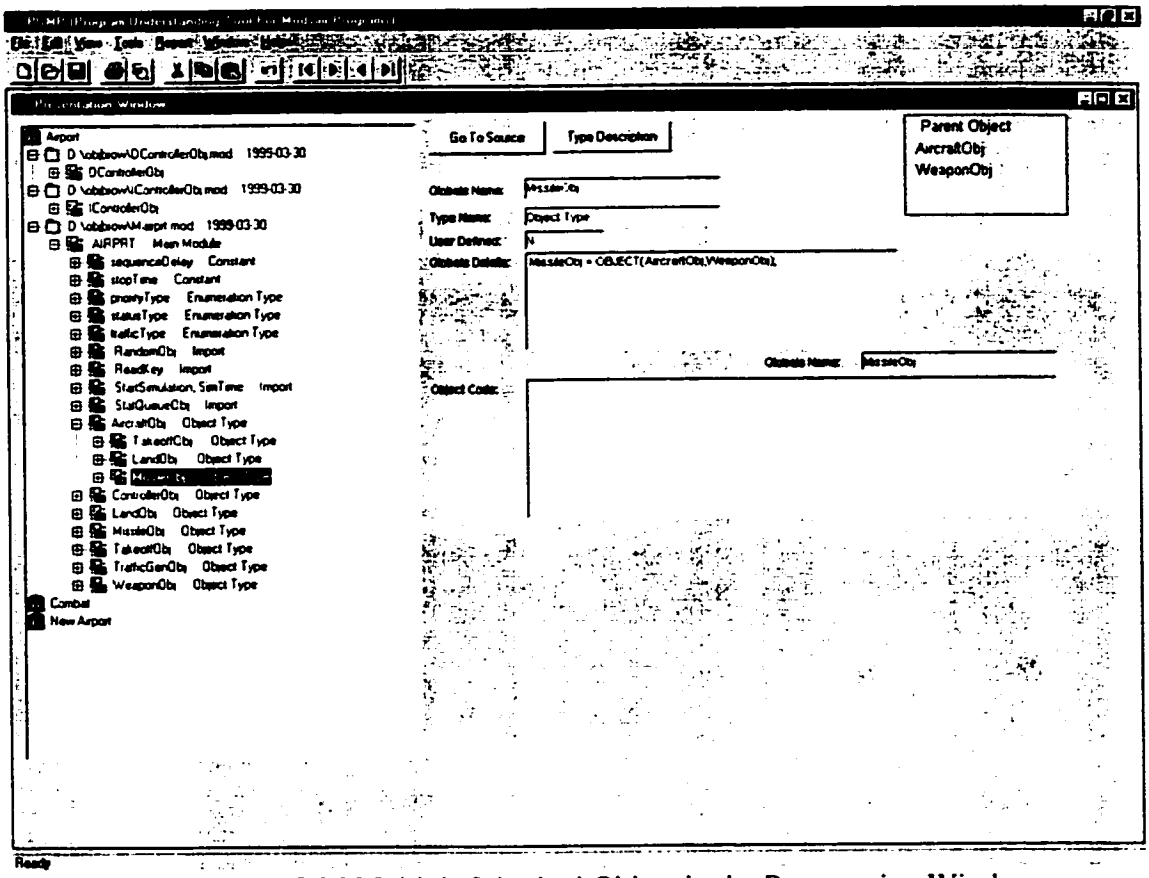


Figure 5.26 Multiple Inherited Object in the Presentation Window

The datawindow shows that MissileObj is inherited from AircraftObj and WeaponObj. If a change has to made in a parent object, PUMP will help the novice programmer by indicating which child objects are going affected by the chance.

5.11 Limitation of PUMP

Several limitations of PUMP are outlined below:

a.) PUMP cannot detect buggy programs. In other words, it is not able to detect syntax or programming errors. All the source code files that are being analyzed by PUMP are assumed to syntactically correct and executable.

b.) PUMP is language dependent. It is not able to analyze and provide program understanding support for programs written in programming languages other than MODSIM.

c.) Development of MODSIM programs can be done in two environments, UNIX and Microsoft Windows. PUMP however was developed for the Microsoft Windows environment. It does not run in a UNIX environment. In other words, PUMP is not able to analyze MODSIM programs developed under the UNIX environment.

d.) There have been only limited numbers of MODSIM programs during the completion of this thesis. In strict form it could be viewed as a prototype version and might recovers areas that require enhancements.

e.) PUMP lacks any quality assurance component that focuses on either maintaining or enhancing the quality of a MODSIM program.

6 Conclusions
6.1 Concluding Remarks

Automated program understanding systems have the potential to make important contributions to reducing the very substantial costs of program maintenance. The continually increasing complexity of new software products suggests the need for immediate progress in this area.

In this thesis we have formulated a particular approach to program understanding within the context of the object-oriented simulation language, MODSIM. The software tool that has been developed is called PUMP (Program Understanding of MODSIM Programs). The input to this tool is the syntactically correct MODSIM program code for a simulation project which typically is distributed over several files. PUMP carries out an analysis of the code and organizes data about relevant features within a collection of seventeen tables of a relational database. Information about the program (i.e. the simulation project) is formulated by accessing the database.

The main presentation is via a user interface that is organized along hierarchical lines that correspond to the organizational structure of MODSIM programs. The hierarchical approach permits examination of program features in increasing levels of detail.

The main thrust of the analysis is to identify entity types (e.g., objects, methods, variables) used in the program together with their interrelationships. This includes both those entity types taken from libraries (either the MODSIM standard Library or user-created libraries) and those created solely for the project being analyzed.

The work has demonstrated that the notion of program understanding can be viewed

from two broad perspectives, one, which is primarily syntax oriented, and the other which is primarily semantics oriented. The first focuses on identifying the various language tokens and constructs that appear in the program and interrelationships among them. The second focuses on identifying behavioral properties of the program that arise because of the interrelationships among the language tokens and constructs within the program.

The program insights provided by PUMP fall principally within the first category. The approach taken in this regard is likely easily transferable to other object-oriented environments. Work in the second category is considerably more complex and is likely to be restricted to the specific object-oriented environment that is being considered.

Some particular noteworthy features of PUMP are summarized below:

(a) Documentation

It can be plausibly argued that the cost of maintenance of an application increases with time. Several reasons can be identified. Often, for example, the maintenance team for a particular application includes few, if any, members of the development team. The turn-over of team members is typically high and the team often includes a high proportion of novice programmers. These factors all contribute to a continuous loss of information about the application. Furthermore, documentation quality often deteriorates because changes are poorly recorded. Consequently maintenance tasks become increasingly harder (and costlier) to carry out. In this respect, PUMP makes significant contribution by allowing the user to execute several pre-defined reports for a selected project. Various predefined reports relating the

structure of the MODSIM program can be produced to aid in program documentation The reports are listed below:

1) Project Stats Report – This report lists basic statistics relating to entities used in a project. This report provides important information for determining the cost of maintaining an application.

2) Project Source Files Report – This report lists the entire source files associated with a project.

3) Project Modules Report – This report lists all the modules in a source file associated with a project.

4) Project Globals Report - This report lists all the Globals in a module that is a particular source file associated with a project.

5) Project Component Names Report – This report lists all the components of objects that are included within a project.

6) Project Component Details Report – This report provides details of components that are included within a project.

(b) Knowledge Base and User Interface

PUMP creates a transformation of the MODSIM program being analyzed in the form of a knowledge base, which captures the key aspects of the program. The knowledge base is intended to support all projects in the enterprise and this considerably facilitates the re-use of program modules. The PUMP user interface provides user-friendly access to this knowledge base. It can be used both for system exploration and for browsing; e.g., it conveniently displays

complex inheritance trees. A tree-view approach allows the level of detail to be easily adjusted to accommodate specific needs of the user. This user interaction is via a convenient graphical user interface (GUI). PUMP also incorporates a powerful editing environment that can be used to examine and edit specified portions of the source code.

(c) Predefined Queries

A variety of the features of the MODSIM program being analyzed can be presented to the PUMP user via a set of predefined queries. Each of these provides program understanding information.

6.2 Evaluation Process

The software evaluation process is an essential activity for ensuring software quality. It establishes the current status of the software being considered (in this case, PUMP) and identifies the areas or points that needs to be improved. In order to evaluate the target software, an essential prerequisite is a thorough understanding of it.

Metrics are often used to evaluate the software. These are simply quantitative measures of the extent or degree to which the software possesses or exhibits a certain characteristic, quantity, property, or attribute. Metrics provides us with measurements; i.e., numbers with an associated unit of measure, which provide insight into some aspect of the software.

Metrics can be viewed in many ways [Basili81] and furthermore, they can be objective and subjective. Objective metrics are absolute measures; e.g. the time taken in the development process, the number of lines of code or the number of errors or required changes. Examples of subjective metrics are an estimate of the extent or degree to which some technique has been applied, or the classification or qualification of a problem or experience, usually expressed in terms of a relative scale.

The evaluation process requires a mechanism for determining what data is to be collected, why it is to be collected, and how the collected data is to be interpreted [Welss84]. The data collection process itself requires a basic paradigm that traces the goals of the collection process; (i.e., the reason(s) the data are being collected) to the actual data.

A determination of the extent to which the underlying goals of PUMP have been met would require extensive evaluation, which is beyond the scope of this project. Nevertheless several avenues of investigation that could be of particular value in this regard can be identified; e.g.,

a) Determination of the extent to which PUMP's effectiveness varies when applied in small, medium and large MODSIM programs.

b) Usefulness of PUMP for users with a range of expertise with MODSIM; e.g., novice through expert programmers

c) PUMP's usefulness may vary with respect to various features of MODSIM programs; e.g., inheritance, reuse of library modules and explorations of such feature dependence would be worthwhile.

The evaluation process requires evaluation planning. The evaluation planning involves a number of steps. These are summarized below within the context of an evaluation of PUMP:

1. Prepare a PUMP evaluation plan.
2. Install the PUMP software.
3. Train two test groups on the features of PUMP; the first group consists of senior programmers and the second group consists of junior programmers.
4. Select a group of MODSIM programs (perhaps 7 to 10), which vary in length (short: < 200 lines of codes, to long: > 2000 lines of code). All programs must be syntactically correct and error free.
5. For each of the programs, identify a set of changes or maintenance tasks that need to carried out.
6. Separate the group of senior programmers into two subgroups. The first carries out the prescribed tasks with the aid of PUMP and the second without PUMP. Organize the group of junior programmers in a similar manner.
7. Prepare a PUMP evaluation report which, in particular, applies appropriate statistical assessment tools to determine whether perceived performance differences are statistically relevant. The report would also include a summary of the views of the PUMP users in respect to various attributes of PUMP; e.g., user-friendliness, completeness (applicability to the prescribed tasks), documentation.

With respect to step 5, care needs to be taken in identifying performance criteria that are meaningful and useful measures of the defined tasks. Two that are likely candidates are

(a) The time required to complete the specified task and

(b) The time required to test the correctness of the changes made.

6.3 Future Work

The main area of possible future work with PUMP would be its extension into the "deeper", semantically oriented, aspect of programming understanding, as outlined earlier in section 6.1. Topics of particular interest here would be:

a) Analysis of the inherent implications of WAIT statements within objects and procedures appearing in the program code for a project; e.g., identification of potential deadlock situations.

b) Analysis of the various ASK and TELL methods within the program with a meaningful presentation of their interactions and possibly identification of potentially anomalous situations.

Although PUMP provides some pre-defined reports, an area of possible extension would be the incorporation of document templates that would permit flexibility in report presentation.

Substantial information about the object classes encountered by PUMP in the various projects analyzed to date is maintained in the PUMP database. A means of consolidating this information and presenting it from the perspective of possible re-use in new simulation projects, would be of considerable value.

Appendix A

In this Appendix, we provide, for each table in the PUMP database, the attributes, their type characterization and primary key.

1. **Project Table** : Stores project information
   Primary Key : project_id

Attributes	Attribute Type	Attribute Description
project_id	Integer NOT NULL	Project identification number
project_name	Varchar(80) NOT NULL	Project Name
analysis_date	Date NULL	Analysis date for the project
analyzed_by	Varchar(10) NULL	User identification of person carrying out the analysis
project_description	Long varchar NULL	Description of the project

2. **Source_file Table** : Stores the source code file names associated with each project
   Primary Key : file_id

Attributes	Attribute Type	Attribute Description
file_id	Integer NOT NULL	File identification number
filename	Varchar(80) NOT NULL	Name of the file containing the source code
date_created	Date NOT NULL	Creation date of the source code file

3. **Module_information Table :** Stores information about modules that appear in user prepared files
   Primary Key : module_id

Attributes	Attribute Type	Attribute Description
module_id	Integer NOT NULL	Module identification number
file_id	Integer NOT NULL	File identification number for the file in which the module resides
module_name	Varchar(80) NOT NULL	Name of the module
module_type	Char(1) NOT NULL	Identifier of the type of module: M-Main, I-Implementation and D-Definition

4. **Component_detail Table :** Stores details of component objects
Primary Key : component_detail_id

Attributes	Attribute Type	Attribute Description
component_detail_id	Integer NOT NULL	Component Detail identification number
globals_id	Integer NOT NULL	Identifier for the global in which this component resides
entity_id	Integer NOT NULL	Identifier for the component entity
component_name	Varchar(80) NOT NULL,	Name of the component
code_fragment	Varchar(150) NOT NULL	The code fragment associated with the component

5. **Object_inheritance Table** : Stores inheritance information about non-MODSIM library objects
Primary Key : (object_id, object_parent)

Attributes	Attribute Type	Attribute Description
object_id	Integer NOT NULL	Corresponds to the globals_id in Globals_detail Table
object_parent	Integer NOT NULL	Corresponds to the globals_id which is a parent of object_id in Globals_detail Table

6. **Entity_type Table** : Stores information about all entities inherent in MODSIM together with those created within projects
Primary Key entity_id

Attributes	Attribute Type	Attribute Description
entity_id	Integer NOT NULL	Entity identification number
entity_name	Varchar(40) NULL	Entity name
entity_description	Varchar(40) NOT NULL	Brief description of the entity
user_defined	Char(1) NULL	Yes/No Flag – identifies user-defined entity

7. **Project_stats Table** : Stores basic statistics relating to entities used in a project
Primary Key : (project_id, entity_id)

Attributes	Attribute Type	Attribute Description
project_id	Integer NOT NULL	Same as project_id in Project Table
entity_id	Integer NOT NULL	Same as entity_id of Declarations Table
count	Integer NOT NULL	Count of the entity referenced to by entity_id within the project referenced by project_id

8. **Library_information Table** : Stores information about all libraries thus far encountered by PUMP (initialized with the MODSIM standard library)
Primary Key : library_id

Attributes	Attribute Type	Attribute Description
library_id	Integer NOT NULL	Library Identification number
library_name	Varchar(40) NOT NULL	Library Name
modsim_library	Char(1) NULL	Yes if library is the MODSIM library, No otherwise
library_description	long varchar NULL	Description of the library

9. **Project_libraries Table** : Stores information about libraries that are referenced a particular project
Primary Key : (project_id, library_id)

Attributes	Attribute Type	Attribute Description
project_id	Integer NOT NULL	Same as project_id in Project Table
library_id	Integer NOT NULL	Same as library_id in Library_information Table

10. **Library_modules Table** : Stores information about every Library Modules thus far encountered by PUMP (initialized with the modules within the MODSIM standard library)
Primary Key : module_id

Attributes	Attribute Type	Attribute Description
module_id	Integer NOT NULL	Module Identification number
library_id	Integer NOT NULL	Same as library_id in Library_information Table
module_name	Varchar(50) NULL	Name of the Module

11. **Project_library_modules Table** : Stores information about modules from Libraries that are referenced in a particular project
Primary Key : module_project_id

Attributes	Attribute Type	Attribute Description
module_project_id	Integer NOT NULL	Unique identification number
project_id	Integer NOT NULL	Same as project_id in Project Table
module_id	Integer NOT NULL	Same as module_id in Library_modules Table

12. **Library_objects Table :** Stores information about every library object thus far encountered by PUMP (initialized with the objects in the MODSIM standard library)
Primary Key : object_id

Attributes	Attribute Type	Attribute Description
object_id	Integer NOT NULL	Object identification number
module_id	Integer NOT NULL	Same as module_id in Library_modules Table
entity_id	Integer NOT NULL	Same as entity_id in Declaration Table
object_name	Varchar(50) NOT NULL	Name of object
object_details	Long varchar NOT NULL	Description of object

13. **Library_object_inheritance Table** : Stores inheritance information about library objects
Primary Key : (object_id, object_parent)

Attributes	Attribute Type	Attribute Description
object_id	Integer NOT NULL	Same as object_id in the Library_object Table
object_parent	Integer NOT NULL	Relates to the object_id in Library_object table

14. **Project_library_objects Table**– Stores information about Library Objects that are used in a particular project
Primary Key - object_project_id

Attributes	Attribute Type	Attribute Description
object_project_id	Integer NOT NULL	Unique Identification Number
project_id	Integer NOT NULL	Same as project_id in Project Table
object_id	Integer NOT NULL	Same as object_id in Library_object Table

15. **Project_source_files Table**: Stores information about source code files used in the various projects
Primary Key : (project_id, file_id)

Attributes	Attribute Type	Attribute Description
project_id	Integer NOT NULL	Same as project_id in Project Table
file_id	Integer NOT NULL	Same as file_id in Source_file Table

16. **Object_code Table** : Stores the entire code for each non-library object
Primary Key : (object_id, code_type)

Attributes	Attribute Type	Attribute Description
object_id	Integer NOT NULL	globals_id of the Globals_detail Table
code_type	Char(1) NOT NULL	The code type – I = Implementation D = Definition
object_code	Long varchar NOT NULL	The entire code of object_id

17. **Globals_detail Table**: Stores information about all user-defined objects together with all entities that are global to the program
Primary Key : globals_id

Attributes	Attribute Type	Attribute Description
globals_id	Integer NOT NULL	Globals identification number
module_id	Integer NOT NULL	Corresponds to module_id in Modsim_information Table and represents the location of globals_id
entity_id	Integer NOT NULL	Correspond to entity_id in Declarations Table and represents the identifier for the type of entity
globals_name	Varchar(40) NOT NULL	The name of the instance of entity type referred by entity_id
globals_details	Varchar(80) NOT NULL	The code fragment defining the entity instance

Appendix B

Shown below is the SQL Script to create an instance of the PUMP database.

```
% Usage:    isql read D:\SQLANY50\WIN32\PUMP.SQL
%
% This command file reloads a database that was unloaded using
"dbunload".
%
%

SET OPTION Statistics = 3;
SET OPTION Date_order = 'YMD';


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%    Create userids and grant user permissions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

GRANT CONNECT TO "DBA" IDENTIFIED BY ENCRYPTED
'\xF0\x63\x0D\x6E\xDA\x08\xC0\x66\x43\x2F\x42\x18\x92\xF1\xBA\xB7\x8D\xE
0\x12\xA9\xED\x6F\xF7\x7E\xC2\x02\x68\xE9\xD9\x27\x49\x72\x25\xEF\x10\x0
0';
GRANT RESOURCE, DBA, SCHEDULE TO "DBA";
GRANT CONNECT TO "DBO";
GRANT GROUP TO "DBO";
GRANT RESOURCE, DBA TO "DBO";
commit work;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%    Create user-defined types
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

commit work;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%    Create tables
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
CREATE TABLE "DBA"."Project"
(
      "project_id"            integer NOT NULL,
      "project_name"          varchar(80) NOT NULL,
      "analysis_date"         date NULL,
      "analyzed_by"           varchar(10) NULL,
      "project_description"         long varchar NULL,
      PRIMARY KEY ("project_id")
);
CREATE TABLE "DBA"."source_file"
(
      "file_id"               integer NOT NULL,
      "filename"              varchar(80) NOT NULL,
      "date_created"          timestamp NOT NULL,
      PRIMARY KEY ("file_id")
);
```

```
CREATE TABLE "DBA"."module_information"
(
      "module_id"              integer NOT NULL,
      "file_id"                integer NOT NULL,
      "module_name"            varchar(80) NOT NULL,
      "module_type"            char(1) NOT NULL,
      PRIMARY KEY ("module_id")
);
CREATE TABLE "DBA"."component_detail"
(
      "component_detail_id"          integer NOT NULL,
      "globals_id"             integer NOT NULL,
      "entity_id"              integer NOT NULL,
      "component_name"               varchar(80) NOT NULL,
      "code_fragment"          varchar(150) NOT NULL,
      PRIMARY KEY ("component_detail_id")
);
CREATE TABLE "DBA"."object_inheritance"
(
      "object_id"              integer NOT NULL,
      "object_parent"          integer NOT NULL,
      PRIMARY KEY ("object_id", "object_parent")
);
CREATE TABLE "DBA"."project_stats"
(
      "project_id"             integer NOT NULL,
      "entity_id"              integer NOT NULL,
      "count"                  integer NOT NULL,
      PRIMARY KEY ("project_id", "entity_id")
);
CREATE TABLE "DBA"."library_information"
(
      "library_id"             integer NOT NULL,
      "library_name"           varchar(40) NOT NULL,
      "modsim_library"               char(1) NULL,
      "library_description"          long varchar NULL,
      PRIMARY KEY ("library_id")
);
CREATE TABLE "DBA"."project_libraries"
(
      "project_id"             integer NOT NULL,
      "library_id"             integer NOT NULL,
      PRIMARY KEY ("project_id", "library_id")
);
CREATE TABLE "DBA"."library_modules"
(
      "module_id"              integer NOT NULL,
      "library_id"             integer NOT NULL,
      "module_name"            varchar(50) NULL,
      PRIMARY KEY ("module_id")
);
CREATE TABLE "DBA"."project_library_modules"
(
      "module_project_id"            integer NOT NULL,
      "project_id"             integer NOT NULL,
      "module_id"              integer NOT NULL,
      PRIMARY KEY ("module_project_id")
```

```
);
CREATE TABLE "DBA"."library_objects"
(
      "object_id"             integer NOT NULL,
      "module_id"             integer NOT NULL,
      "entity_id"             integer NOT NULL,
      "object_name"           varchar(50) NOT NULL,
      "object_details"              long varchar NOT NULL,
      PRIMARY KEY ("object_id")
);
CREATE TABLE "DBA"."library_object_inheritance"
(
      "object_id"             integer NOT NULL,
      "object_parent"         integer NOT NULL,
      PRIMARY KEY ("object_id", "object_parent")
);
CREATE TABLE "DBA"."project_library_objects"
(
      "object_project_id"           integer NOT NULL,
      "project_id"            integer NOT NULL,
      "object_id"             integer NOT NULL,
      PRIMARY KEY ("object_project_id")
);
CREATE TABLE "DBA"."project_source_files"
(
      "project_id"            integer NOT NULL,
      "file_id"               integer NOT NULL,
      PRIMARY KEY ("project_id", "file_id")
);
CREATE TABLE "DBA"."object_code"
(
      "object_id"             integer NOT NULL,
      "code_type"             char(1) NOT NULL,
      "object_code"           long varchar NOT NULL,
      PRIMARY KEY ("object_id", "code_type")
);
CREATE TABLE "DBA"."globals_detail"
(
      "globals_id"            integer NOT NULL,
      "module_id"             integer NOT NULL,
      "entity_id"             integer NOT NULL,
      "globals_name"          varchar(40) NOT NULL,
      "globals_details"             varchar(80) NOT NULL,
      PRIMARY KEY ("globals_id")
);
CREATE TABLE "DBA"."entity_type"
(
      "entity_id"             integer NOT NULL,
      "entity_description"          varchar(40) NOT NULL,
      "entity_name"           varchar(40) NULL,
      "user_defined"          char(1) NULL,
      PRIMARY KEY ("entity_id")
);
commit work;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%    Reload data
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

commit work;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%    Add foreign key definitions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

ALTER TABLE "DBA"."module_information"
      ADD FOREIGN KEY "mod_info_source_fk" ("file_id")
      REFERENCES "DBA"."source_file" ("file_id");

ALTER TABLE "DBA"."component_detail"
      ADD FOREIGN KEY "fk_entity_type" ("entity_id")
      REFERENCES "DBA"."entity_type" ("entity_id");

ALTER TABLE "DBA"."component_detail"
      ADD FOREIGN KEY "fk_globals_object" ("globals_id")
      REFERENCES "DBA"."globals_detail" ("globals_id");

ALTER TABLE "DBA"."object_inheritance"
      ADD FOREIGN KEY "fk_obj_inheritance" ("object_id")
      REFERENCES "DBA"."globals_detail" ("globals_id");

ALTER TABLE "DBA"."object_inheritance"
      ADD FOREIGN KEY "fk_obj_parent" ("object_parent")
      REFERENCES "DBA"."globals_detail" ("globals_id");

ALTER TABLE "DBA"."project_stats"
      ADD FOREIGN KEY "project_stats_fk" ("project_id")
      REFERENCES "DBA"."Project" ("project_id");

ALTER TABLE "DBA"."project_stats"
      ADD FOREIGN KEY "project_types_fk" ("entity_id")
      REFERENCES "DBA"."entity_type" ("entity_id");

ALTER TABLE "DBA"."project_libraries"
      ADD FOREIGN KEY "library_information_idx" ("library_id")
      REFERENCES "DBA"."library_information" ("library_id");

ALTER TABLE "DBA"."project_libraries"
      ADD FOREIGN KEY "project_libraries_fk" ("project_id")
      REFERENCES "DBA"."Project" ("project_id");

ALTER TABLE "DBA"."library_modules"
      ADD FOREIGN KEY "modsim_module_fk" ("library_id")
      REFERENCES "DBA"."library_information" ("library_id");

ALTER TABLE "DBA"."project_library_modules"
      ADD FOREIGN KEY "project_modsim_module_fk" ("project_id")
      REFERENCES "DBA"."Project" ("project_id");

ALTER TABLE "DBA"."project_library_modules"
      ADD FOREIGN KEY "module_modsim_module_fk" ("module_id")
      REFERENCES "DBA"."library_modules" ("module_id");
```

```
ALTER TABLE "DBA"."library_objects"
      ADD FOREIGN KEY "object_modsim_module_fk" ("module_id")
      REFERENCES "DBA"."library_modules" ("module_id");

ALTER TABLE "DBA"."library_objects"
      ADD FOREIGN KEY "declaration_modsim_module_fk" ("entity_id")
      REFERENCES "DBA"."entity_type" ("entity_id");

ALTER TABLE "DBA"."library_object_inheritance"
      ADD FOREIGN KEY "object_modsim_inheritance_fk" ("object_id")
      REFERENCES "DBA"."library_objects" ("object_id");

ALTER TABLE "DBA"."library_object_inheritance"
      ADD FOREIGN KEY "object_modsim_parent_fk" ("object_parent")
      REFERENCES "DBA"."library_objects" ("object_id");

ALTER TABLE "DBA"."project_library_objects"
      ADD FOREIGN KEY "project_object_fk" ("project_id")
      REFERENCES "DBA'."Project" ("project_id");

ALTER TABLE "DBA"."project_library_objects"
      ADD FOREIGN KEY "modsim_object_fk" ("object_id")
      REFERENCES "DBA"."library_objects" ("object_id");

ALTER TABLE "DBA"."project_source_files"
      ADD FOREIGN KEY "fk_source_file_file" ("file_id")
      REFERENCES "DBA"."source_file" ("file_id");

ALTER TABLE "DBA"."project_source_files"
      ADD FOREIGN KEY "fk_project_file" ("project_id")
      REFERENCES "DBA"."Project" ("project_id");

ALTER TABLE "DBA"."object_code"
      ADD FOREIGN KEY "fk_object_code" ("object_id")
      REFERENCES "DBA"."globals_detail" ("globals_id");

ALTER TABLE "DBA"."globals_detail"
      ADD FOREIGN KEY "fk_types_id" ("entity_id")
      REFERENCES "DBA"."entity_type" ("entity_id");

ALTER TABLE "DBA"."globals_detail"
      ADD FOREIGN KEY "fk_module_info" ("module_id")
      REFERENCES "DBA"."module_information" ("module_id");
commit work;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%    Create functions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

commit work;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%    Create views
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
commit work;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%     Set option values
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


SET OPTION Statistics =;
SET OPTION Date_order =;


%
%SQL Option Statements for user
%

SET OPTION "PUBLIC"."Blocking" = 'On';
SET OPTION "PUBLIC"."Checkpoint_time" = '60';
SET OPTION "PUBLIC"."Conversion_error" = 'On';
SET OPTION "PUBLIC"."Timestamp_format" = 'YYYY-MM-DD HH:NN:SS.SSS';
SET OPTION "PUBLIC"."Time_format" = 'HH:NN:SS.SSS';
SET OPTION "PUBLIC"."Date_format" = 'YYYY-MM-DD';
SET OPTION "PUBLIC"."Date_order" = 'YMD';
SET OPTION "PUBLIC"."Isolation_level" = '0';
SET OPTION "PUBLIC"."Precision" = '30';
SET OPTION "PUBLIC"."Recovery_time" = '2';
SET OPTION "PUBLIC"."Replicate_all" = 'Off';
SET OPTION "PUBLIC"."Row_counts" = 'Off';
SET OPTION "PUBLIC"."Scale" = '6';
SET OPTION "PUBLIC"."Thread_count" = '0';
SET OPTION "PUBLIC"."Wait_for_commit" = 'Off';
SET OPTION "PUBLIC"."Quoted_identifier" = 'On';
SET OPTION "PUBLIC"."Allow_nulls_by_default" = 'On';
SET OPTION "PUBLIC"."Automatic_timestamp" = 'Off';
SET OPTION "PUBLIC"."Query_plan_on_open" = 'Off';
SET OPTION "PUBLIC"."Cooperative_commits" = 'On';
SET OPTION "PUBLIC"."Cooperative_commit_timeout" = '250';
SET OPTION "PUBLIC"."Delayed_commits" = 'Off';
SET OPTION "PUBLIC"."Delayed_commit_timeout" = '500';
SET OPTION "PUBLIC"."Non_keywords" = '';
SET OPTION "PUBLIC"."ANSI_blanks" = 'Off';
SET OPTION "PUBLIC"."Auto_commit" = 'Off';
SET OPTION "PUBLIC"."Auto_refetch" = 'On';
SET OPTION "PUBLIC"."Bell" = 'On';
SET OPTION "PUBLIC"."Commit_on_exit" = 'On';
SET OPTION "PUBLIC"."Echo" = 'On';
SET OPTION "PUBLIC"."Headings" = 'On';
SET OPTION "PUBLIC"."Input_format" = 'ASCII';
SET OPTION "PUBLIC"."ISQL_log" = '';
SET OPTION "PUBLIC"."NULLS" = '(NULL)';
SET OPTION "PUBLIC"."On_error" = 'Prompt';
SET OPTION "PUBLIC"."Output_format" = 'ASCII';
SET OPTION "PUBLIC"."Output_length" = '0';
SET OPTION "PUBLIC"."Screen_format" = 'Text';
SET OPTION "PUBLIC"."Statistics" = '3';
SET OPTION "PUBLIC"."Truncation_length" = '30';
```

```
SET OPTION "PUBLIC"."Command_delimiter" = ';';
SET OPTION "PUBLIC"."Verify_all_columns" = 'Off';
SET OPTION "PUBLIC"."Delete_old_logs" = 'Off';
commit work;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%    Create user messages
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

commit work;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%    Create procedures
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

commit work;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%    Create triggers
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

commit work;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%    Create SQL remote definitions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

CREATE REMOTE TYPE "FILE" ADDRESS '';
CREATE REMOTE TYPE "MAPI" ADDRESS '';
CREATE REMOTE TYPE "VIM" ADDRESS '';
CREATE REMOTE TYPE "SMTP" ADDRESS '';
commit work;
```

Appendix C

This Appendix provides the complete program listing for the MODSIM program used for illustrative purpose throughout the main body of the thesis. This program has been taken from [MODTUT98]. This MODSIM program has been organized into three source code files; namely,

a.) mairport.mod
b.) dcontrollerobj.mod
c.) icontrollerobj.mod

a.) The contents of c:\modsim\mairport.mod are given below:

```
MAIN MODULE airport;
{ Simple non-graphic airport model -- Rules for the airport:

1. Takeoff: The controller may clear an aircraft for takeoff if no arriving aircraft is in the 6 mile approach
path and the runway is clear. Arriving
aircraft have priority over those waiting to take off. Departing aircraft are placed in a FIFO queue if they
cannot be cleared immediately upon requesting takeoff clearance.
2. Landing : The approach corridor is 6 miles long. No other aircraft may be cleared to commence an
approach if the approach path is occupied. If the runway is not clear when an arriving aircraft reaches its
threshold, it must go around for another approach. It then has priority for landing ahead of other arriving
aircraft. Go-arounds always take 5 minutes to complete. At the end of 5 minutes, the aircraft commences
another approach or is placed in the arrival queue ahead of arriving traffic.
3. Arriving aircraft which cannot be immediately cleared for landing are place in a FIFO queue for landing.
The controller clears each aircraft to commence landing
approach if no aircraft is using the approach path.

Goals for the model:
1. Run the model with various traffic rates
2. Measure the following parameters which will be important to users of the airport:
a. Arriving and departing queues:
- max size
- average size
- average delay time (time in a queue)
b. Number of aircraft which arrived & departed
c. Number of arriving aircraft which executed a go-around.

Objects involved in the simulation:
Controller - Modeled behaviors:
a. Clear aircraft to land
b. Clear aircraft to takeoff
c. Receive notification of arriving and departing aircraft.
d. Receive notification when arriving and departing aircraft have cleared the runway.
e. Receive progress reports from aircraft making approaches.

Aircraft - Modeled behaviors:
a. Perform takeoff when controller gives takeoff clearance
b. Perform landing when controller gives landing clearance
c. Perform go-around if runway is occupied. Traffic Generator - Modeled behavior:
a. Generate arriving & departing aircraft and request landing or takeoff clearance.

Time base for the model is minutes.
```

```
Distance is measured in Nautical Miles - 1 NM = 1.15
Miles = 1.85 Km
Speed is measured in Knots ( Nautical Miles per Hour )
AC = aircraft }

1       FROM GrpMod IMPORT StatQueueObj;
2       FROM SimMod IMPORT StartSimulation, SimTime;
3       FROM RandMod IMPORT RandomObj;
4       FROM IOMod IMPORT ReadKey;
5       FROM ControllerMod IMPORT ControllerObj;
6
7       TYPE
8       trafficType = ( arrive, depart );
9       statusType = ( clear, inUse );
10      priorityType = ( normal, goAround );
11
12      AircraftObj = OBJECT; { virtual aircraft type with common attributes }
13              ovhdTime : REAL; { overhead time: taxi onto runway for TO or roll out after landing
14              taskTime : REAL; { time required to takeoff or fly approach }
15              startTime : REAL; { sim time at which AC starts Land or TO }
16      END OBJECT;
17
18      TakeoffObj = OBJECT(AircraftObj);
19              TELL METHOD Takeoff;
20              ASK METHOD ObjInit; { set takeoff performance attributes }
21              ASK METHOD ObjTerminate; { report statistics before DISPOSing }
22      END OBJECT;
23
24      LandObj = OBJECT(AircraftObj);
25              landPriority : priorityType; { normal or goAround which is higher }
26              TELL METHOD Land;
27              TELL METHOD GoAround;
28              ASK METHOD ObjInit; { set landing performance attributes }
29              ASK METHOD ObjTerminate; { report statistics before DISPOSing }
30      END OBJECT;
31
32      TrafficGenObj = OBJECT
33              numACgen : INTEGER; { number of AC generated }
34              numACcomp : INTEGER; { number of AC completed landing/takeoff }
35              totTimeSpent : REAL; { total time spent by AC completing task }
37              ranGen : RandomObj; { random number gen. used by this obj }
38              TELL METHOD GenTraffic(IN interarrivalRate : REAL; IN kindOfAC : traffic-Type);
39              ASK METHOD LogCompletion(IN whenStarted: REAL); { when done }
40              ASK METHOD ObjInit;
41      END OBJECT;
42
43      VAR
44              runway : statusType;
45              approachPath : statusType; { approach corridor }
46              ArriveGen : TrafficGenObj;
47              DepartGen : TrafficGenObj;
48              Controller : ControllerObj;
49              randSeed : INTEGER; { each new generator uses a new seed }
50              trafficRanGen : RandomObj; { used by aircraft to set their fields }
51              goAroundCount : INTEGER;
52              interRate : REAL; { interarrival rate }
```

```
53          ch : CHAR;
54
55      CONST
56          stopTime = 1440.0; { minutes }
57          sequenceDelay = 1.0; { interval between departing AC }
58
59      OBJECT TakeoffObj;
60          TELL METHOD Takeoff;
61        BEGIN
62          WAIT DURATION ovhdTime + taskTime { taxi into position & takeoff }
63          END WAIT;
64          TELL Controller ClearOfRunway;
65              DISPOSE(SELF);
66          END METHOD;
67
68          ASK METHOD ObjInit; { takeoff }
69            BEGIN
70              ovhdTime := trafficRanGen.Exponential(0.9);
71              taskTime := trafficRanGen.UniformReal(0.5, 0.9):
72              startTime := SimTime();
73          END METHOD;
74
75          ASK METHOD ObjTerminate;
76            BEGIN
77              ASK DepartGen LogCompletion(startTime);
79          END METHOD;
80      END OBJECT;
81
82      OBJECT LandObj;
83          TELL METHOD Land;
84            BEGIN
85              WAIT DURATION taskTime
86              END WAIT;
87              TELL Controller ClearOfApproach;
88              IF (runway <> clear) { is runway clear? }
89                  GoAround;
90                  RETURN; landing has been aborted, so exit this method
91              END IF;
92              runway := inUse;
93              WAIT DURATION ovhdTime { roll out time }
94              END WAIT;
95              TELL Controller ClearOfRunway;
96              DISPOSE(SELF);
97          END METHOD;
98
99          TELL METHOD GoAround;
100           BEGIN
101             INC(goAroundCount);
102             WAIT DURATION 5.0
103             END WAIT;
104             landPriority := goAround;
105             ASK Controller LandingClearance(SELF);
106         END METHOD;
107
108         ASK METHOD ObjInit; { land }
109           BEGIN
```

```
110                 taskTime := trafficRanGen.UniformReal(2.8, 3.0);
111                 ovhdTime := trafficRanGen.UniformReal(0.8, 1.2);
112                 landPriority := normal;
113                 startTime := SimTime();
114           END METHOD;
115
116           ASK METHOD ObjTerminate;
117             BEGIN
118                 ASK ArriveGen LogCompletion(startTime);
119           END METHOD;
120     END OBJECT { AircraftObj };
121
122     OBJECT TrafficGenObj;
123           TELL METHOD GenTraffic(IN interarrivalRate : REAL; IN kindOfAC : trafficType);
124             VAR
125                 planeTO : TakeoffObj;
126                 planeLand : LandObj;
127             BEGIN
128                 WHILE (SimTime <= stopTime)
129                     WAIT DURATION ranGen.Exponential(interarrivalRate);
130                     END WAIT;
131                     INC(numACgen);
132                     CASE (kindOfAC)
133                         WHEN arrive:
134                             NEW(planeLand);
135                             ASK Controller LandingClearance(planeLand);
136                         WHEN depart:
137                             NEW(planeTO);
138                             ASK Controller TakeoffClearance(planeTO);
139                     END CASE;
140                 END WHILE;
141           END METHOD;
142
143           ASK METHOD LogCompletion(IN whenStarted: REAL);
144             BEGIN
145                 totTimeSpent := totTimeSpent + (SimTime() - whenStarted);
146                 INC(numACcomp);
147           END METHOD;
148
149           ASK METHOD ObjInit;
150             BEGIN
151                 NEW(ranGen);
152                 INC(randSeed); { each generator gets a unique seed }
153                 ASK ranGen TO SetSeed(randSeed);
154           END METHOD;
155
156     END OBJECT { TrafficGenObj };
157
158     PROCEDURE ShowResults;
159       BEGIN
160           OUTPUT;
161           OUTPUT("Simulation Run is Finished at SimTime ", SimTime());
162           OUTPUT("Mean interarrival rate used for arrivals & depar-tures: ", interRate);
163           OUTPUT("# of arriving AC: ", ASK ArriveGen numACgen, " # of departing AC: ", ASK
              DepartGen numACgen);
164           OUTPUT("# of go arounds = ", goAroundCount);
```

```
165             OUTPUT("Max # AC waiting to takeoff & Mean # waiting to take off: ",
                Controller.departQ.Maximum, " - ", Controller.departQ.Mean);
166             OUTPUT("Mean time spent departing: ", ASK DepartGen totTimeSpent / FLOAT(ASK
                DepartGen numACcomp));
167             OUTPUT("Max # AC waiting to land & Mean # waiting to land:",

                Controller.arriveQ.Maximum, " - ", Controller. arriveQ.Mean);
168             OUTPUT("Mean time spent landing: ",
169             ASK ArriveGen totTimeSpent / FLOAT(ASK ArriveGen numAC-comp));
170     END PROCEDURE;
171
172      BEGIN
173             OUTPUT("Mean time between arrivals / departures?");
174             OUTPUT(" (optimum value is around 6 minutes)");
175             INPUT(interRate);
176             NEW(Controller);
177             NEW(trafficRanGen);
178             INC(randSeed);
179             ASK trafficRanGen TO SetSeed(randSeed);
180             NEW(ArriveGen);
181             TELL ArriveGen TO GenTraffic(interRate, arrive);
182             NEW(DepartGen);
183             TELL DepartGen TO GenTraffic(interRate, depart);
184             StartSimulation;
185             ShowResults;
186             OUTPUT;
187             OUTPUT(".... Hit any key to terminate");
188             ch := ReadKey; { on PC Windows holds window open till key is hit }
189     END MODULE.
```

b.) The contents of c:\modsim\dcontrollerobj.mod are given below

```
1       DEFINITION MODULE ControllerObj;
2               ControllerObj = OBJECT;
3                       arriveQ : StatQueueObj;
4                       departQ : StatQueueObj;
5                       ASK METHOD LandingClearance(IN plane : LandObj);
6                       ASK METHOD TakeoffClearance(IN plane : TakeoffObj);
7                       TELL METHOD ClearOfRunway;
8                       TELL METHOD ClearOfApproach;
9                       ASK METHOD ObjInit;
10              END OBJECT{ ControllerObj };
11      END MODULE. {ControllerMod }
```

c.) The contents of c:\modsim\icontrollerobj.mod source code file.

```
1       IMPLEMENTATION MODULE ControllerObj;
2        OBJECT ControllerObj;
3               ASK METHOD LandingClearance(IN AC : LandObj);
4                 BEGIN
5                       IF ((arriveQ.numberIn = 0) AND (approachPath = clear))
6                               approachPath := inUse;
7                               TELL AC TO Land;
8                       ELSE { AC on go around are put first in arriveQ }
9                               IF ((AC.landPriority = goAround) AND (arriveQ.numberIn > 0))
```

```
10                              ASK arriveQ TO AddBefore(arriveQ.First, AC);
11                      ELSE
12                              ASK arriveQ TO Add(AC);
13                      END IF;
14              END IF;
15      END METHOD;
16
17      ASK METHOD TakeoffClearance(IN AC : TakeoffObj);
18        BEGIN
19              IF ((departQ.numberIn = 0) AND (runway = clear) AND (approachPath =
                clear))
20                      runway := inUse;
21                      TELL AC TO Takeoff;
22              ELSE
23                      ASK departQ TO Add(AC);
24              END IF;
25      END METHOD;
26
27      TELL METHOD ClearOfRunway;
28      { AC which have completed landing rollout or takeoff use this method to report that they
        have cleared the runway. Controller then checks to see if an AC is waiting for takeoff }
29        VAR
30              AC : TakeoffObj;
31        BEGIN
32              runway := clear;
33              WAIT DURATION trafficRanGen.Exponential(sequenceDelay)
34              END WAIT;
35              IF ((departQ.numberIn > 0) AND (approachPath = clear) AND (runway =
                clear))
36                      AC := departQ.Remove;
37                      runway := inUse;
38                      TELL AC TO Takeoff;
39              END IF;
40      END METHOD;
41
42      TELL METHOD ClearOfApproach;
43      { AC which have cleared the approach corridor use this method to inform the controller.
        The controller then clears the next arriving aircraft to land. }
44        VAR
45              AC : LandObj;
46        BEGIN
47              IF (arriveQ.numberIn > 0)
48                      AC := arriveQ.Remove;
49                      approachPath := inUse;
50                      TELL AC TO Land;
51              ELSE
52                      approachPath := clear;
53              END IF;
54      END METHOD;
55
56      ASK METHOD ObjInit;
57        BEGIN
58              NEW(arriveQ);
59              ASK arriveQ TO SetDelayStats(TRUE); { turn ON stats collecting }
60              NEW(departQ);
61              ASK departQ TO SetDelayStats(TRUE);
```

```
62              END METHOD;
63
64          END OBJECT { ControllerObj };
65      END MODULE. {ControllerMod }
```

Appendix D

This Appendix provides an overview of the PowerBuilder Development tool used in the development of PUMP.

PowerBuilder

PowerBuilder is a powerful object-oriented client/server front-end development system. [HB97] PowerBuilder is best known for its intuitive user interface, ease-of-use, object-oriented features, and powerful high-level contructs. These provide the most-important reasons for using PowerBuilder as the development environment for PUMP.

A PowerBuilder application is defined by creating PowerBuilder objects (such as windows and controls) with various PowerBuilder Painter utilities. PowerBuilder's DataWindow object is a high-level construct that encapsulates data access within an intelligent database-aware object. DataWindows are in many ways the essence of PowerBuilder's client/server power. It basically stands between the application and the database. All data access in the application is done using the DataWindow.

PowerBuilder also creates and maintains its own database in the target database system. This database stores information about the tables, columns and indexes that are used by an application.

PowerBuilder takes full advantage of the event-driven Windows environment and provides database independent development. Microsoft Window is an event-driven environment and an understanding of events and messaging is essential for the PowerBuilder developer. Events are the actions or state changes that apply to an object. Each object type generally has a distinctive set of events associated with it. Developers can also define their own events for a PowerBuilder application. Events provides the

means for code execution. The developer writes PowerScript code (scripts) to respond to specific events. An event occurrence will trigger the execution of that code. Essentially, all code execution is triggered as the result of an event.

The PowerBuilder objects that are created using various PowerBuilder painters can contain program scripts (written in PowerScript) that execute in response to different events.

PowerBuilder Painters

A PowerBuilder application consists of many components including windows, controls, functions and programming code. Each of these components can be developed using various PowerBuilder painters. Each painter provides a development environment for a specific area of development. For example, the window painter provides the functions that are required to develop the graphical user interface. It allows the creation of a new window and add controls and other graphical objects to this window. A brief overview of each painter is given below:

1) Application painter – This is where the most general application attributes (such as name of the application and libraries that will be used for the application) are defined and where the application object is defined. There can be only one application object for each application. The application object is the starting point; i.e, it is the entry point for the entire application.
2) Window painter – This is where all the windows (together with their controls) for an application are developed.

3) Menu painter – This is where all menus that will be attached to the windows of an application, are developed.

4) DataWindow painter – This is where all the datawindows that are used in the application are developed. DataWindow is the object that encapsulates data access for an application.

5) Structure painter – This is where all the composite variables (i.e. structures) are created. A structure is a set of data elements grouped together as a unit. The set of variables is then referenced by a single name. A paticularly useful application of a structure is as function argument. Here the entire structure is passed to the function as a unit, thereby reducing the number of arguments necessary in a function call.

6) Database painter –The database painter provides a convenient interface to the database. Tables, indexes, views, extended attributes, validation criteria, and display formats can be created using this painter. It allows for the creating, editing, and activating of database connection profiles, and also allows for the export and import of database definitions.

7) Data Manipulation painter – This is in fact, part of the Database painter. Several operations on the database can be performed here such as, select, update, insert and delete Also allowed is the application of filters and the sorting of the result set. The export and import of data in a wide range of formats is possible using the Data Manipulation painter

8) Database Administrator-This is where SQL statements are entered and execution is initiated. The result set can be manipulated as described above in the Data Manipulation painter.

9) Table painter- Here a subset of the operations provided by the Database painter can be executed; e.g., the creation and modification of database tables and the creation, modification and deletion of primary keys.

10) Pipeline painter- The principal function of this painter is the copying of data from one database to another database.

11) Query painter-This painter enables the graphical creation of SQL SELECT statements.

12) PowerScript painter- This is where PowerScript code are created and edited. After declaring a function, the PowerScript editor is used to enter the code.

13) Project painter-This is where project objects are created and maintained. A project object contains all the information that is necessary for building application executables.

14) Library painter-This is the utility for creating and managing PowerBuilder libraries. Libraries are the repositories where the components of the applications reside.

15) User Object painter- The developer can create special types of custom objects in this painter. After creating such a custom object, it can be used in the same way as any native PowerBuilder control (such as a command button).

16) Debugger- The PowerBuilder debugger helps to test an application and provides the features that are needed to track down and fix programming

errors. Features includes setting of breakpoints, single-steping through the code, examination and modification of variable values.

The PowerScript Language

PowerScript is PowerBuilder's programming language. It is a high-level, event-driven programming language with many object-oriented features. It is modeled after C++ and supports flow control statements (conditional branching and looping) and variables with a wide range of date types and a variety of scoping options.

PowerBuilder functions are similar to the functions that are found in developmental libraries that are available for most languages such as C, Pascal, and BASIC. These include functions for data conversion, file access, string manipulation, time, and date manipulation. PowerScript provides approximately 500 built-in functions with a wide range of functionality. It allows the direct embedding of SQL statements for database access and provides support for the event-driven Windows environment. PowerScript also provides functions that allow communications between objects through messaging (i.e., events).

Datawindow

The DataWindow is the single most powerful feature of PowerBuilder. The DataWindow object is a high-level construct that encapsulates data access within an intelligent, data-centric object. Basically, the DataWindow stands between an application and the database. Datawindows are used for nearly all of data display, data manipulation, and updating the database. There are many possible data source for a Data Window. Data can come from relational databases (Sybase, Oracle, SQLBase, Ingres, or any database

that uses the ODBC driver), text and dBase files (.TXT and . DBF), or can be input by the user directly. The Data Window performs data validation based upon what it knows about the data (the data type-string, numeral, and so on) and any rules that have been defined (for example, the age of an employee must be greater that 18, but less than 65).

The Data Window displays the data based on a user-specified format specified. Data Windows have a wide variety of presentation styles, including Freeform, Tabular, Graphs, and Crosstabs. These presentation styles provide a default format for displaying the data. However, there's still a wide latitude for enhancing the default format to meet specific needs.

When paired with a Data Window control (placed in the window by the Window painter), it provides a dual interface: an interface to the user and an interface to the database management system.

References

[Basili81] Basili, V. R. "Evaluating Software Development Characteristics: Assessment of Software Measures in the Software Engineering Laboratory." Proceedings of the Sixth Annual Software Engineering Workshop, December 1981.

[Basili82] Basili, V. R. and Mills, H. D. "Understanding and documenting programs." IEEE Trans. on Software Engineering, SE-8(3):270-283, 1982.

[Basili91] Basili, V. R., Abd-El-Hafiz, S. K., and Caldiera, G. "Towards automated support for extraction of reusable components." In Proceedings of the conference on Software Maintenance, Sorrento, Italy pages 212-219, 1991.

[Bertels93] Bertels, K., Vanneste, P., and De Backer, C. "A Cognitive Approach to Program Understanding." In Proceedings of the Working Conference on Reverse Engineering, Baltimore, Maryland, pages 1-7, 1993.

[Birta91] Birta L.G., Abou-Rabia O., Ören T.I., King D., Wendt R. "Reverse Engineering in The Simulation Lifecycle," SAMS Vol 9 pp 69-84. 1991.

[Booch86] Grady Booch. "Object-oriented Development." " IEEE Transactions on Software Engineering, pp. 211-221, February 1986.

[BPBFLS83] Bratley, P.; B. Fox; and L. Schrage. 1983. A Guide to Simulation. Springer-Verlag, New York.

[Brotsky84] Brotsky D. C. "An algorithm for parsing flow graphs." Master's thesis, MIT, 1984.

[CHBKJT93] Charles Herring, Biju Kalathil and Joseph Teo. "Research in Persistent Simulation: Development of the Persistent MODSIM Object-Oriented Programming Language." USACERL Interim Report FF-93/07, July 1993.

[Cle89] Cleveland L. "A program understanding support environment." IEEE Computer Society Press, pp. 324-344, 1989

[Corbi89] T.A.Corbi. "Program Understanding Challenge for the 1990s." IBM System Journal, pp 294-306 , 1989.

[Cox86] Brad J.Cox. 1986. Object-Oriented Programming. Addison-Wesley, Massachusetts.

[Dirk89] Dirk Ourston. "Program Recognition." IEEE Expert, pp. 36-49, 1989.

[Habermann90] Habermann, A. N. "Engineering larger knowledge-based systems." Data & knowledge Engineering, 5:105-117, 1990.

[Harandi88] Harandi, M. T. and Ning, J. Q. "PAT: a knowledge-based program analysis tool." In Proceedings of the Conference on Software Maintenance, Phoenix, Arizona, pages 312-318, 1988.

[Hartman91] Hartman, J. "Understanding natural programs using proper decomposition." In Proceedings of the 13th International Conference o Software Engineering, pages 62-73, Austin, Texas, 1991.

[Hausler90] Hausler, P. A., Pleszkoch, M. G. Linger, R. C., and Hevner, A. R. "Using function abstraction to understand program behavior." IEEE Software, 7(1):55-63,1990.

[HB97] Howard Block, Millard Brown, Boris Gasin, William Green and Andy Tauber, 1997, PowerBuilder Foundation Class Library Professional Reference, McGraw-Hill.

[Hoare69] Hoare, C. A. R. "An axiomatic basis for computer programming" Communications of the ACM, 12(10):576-580,1969.

[Johnson85] Johnson, W. L. and Soloway, E. "PROUST: knowledge-based program understanding." IEEE Trans. On Software Engineering, SE-11(3):267-275, 1985.

[JY90] Judith E. Grass and Yih-Farn Chen. ``The C++ Information Abstractor.'' 1990 USENIX Conference, 1990, pp. 265-277.

[Kemmerer85] Kemmerer, R. A. and Eckmann, S. T. "UNISEX: a unix-based symbolic executor for Pascal ." Software Practice and Experience, 15(3):439-458, 1985.

[Kozaczynski89] Kozaczynski, W. and Ning, J. Q. "SRE: a knowledge-based environment for large-scale software reengineering activities." In Proceedings of the 11th International Conference on Software Engineering, Phoenix, Arizona, pages 113-122, 1989.

[Letovsky87] Letovsky, "S. Program understanding with the lambda calculus." In Porceedings of the 10th International Joint Conference on AI, pages 512-514, 1987.

[Mills75] Mills, H. D. "The new math of computer programming." Communications of the ACM. 18(1):43-48.1975.

[MODTUT98] MODSIM III The Language for Object-Oriented Programming - Tutorial CACI publication. 1998. Pp. 63-70

[Ning89] Ning, J. Q. "A knowledge-based approach to automatic program analysis." Ph.D. thesis, University of Illinois, Urbana-Champaign, 1989.

[Prem92] Premkumar T. Devanbu. "GENOA - A Customizable, Language- and Front-End Independent Code Analyzer." Proceedings of the Fourteenth

International Conference on Software Engineering, Melbourne, Australia, May 1992, pp. 307-319.

[Quilici93] Quilici, A. "A hybrid approach to recognizing programming plans." In Proceedings of the Working Conference on Reverse Engineering, pages 126-133, Baltimore, Maryland, 1993.

[Rich81] Rich, C. "A formal representation for plans in the programmer's apprentice." In Proceedings of the 7th International Joint Conference on AI, pages 1044-1052, 1981.

[Rich90] Rich, C. and Wills, L. M. "Recognizing a program's design: a graph-parsing approach." IEEE Software, 7(1):82-89, 1990.

[Rob91] D.J. Robson, K.H. Bennett, B.J. Cornelius and M. Munro. "Approaches to Program Understanding." Journal of System and Software, pp. 79-84, Feb 1991.

[Soloway86] Soloway, E. "Learning to program = learning to construct mechanisms and explanations." Communications of the ACM, 29(9):850-858, 1986.

[Ted89] Ted J. Biggerstaff. "Design Recovery for Maintenance and Reuse." IEEE Computer, pp. 36-49, July 1989.

[Thomas89] Dave Thomas. "What's in an object." BYTE pp. 231-240 , March 1989.

[Ward89] Ward. M., Calliss, F. W., and Munro, M. "The maintainer's assistant." In proceedings of the Conference on Software Maintenance, Miami, Florida, pages 307-315,1989.

[Weiser84] Weiser, M. "Program slicing." IEEE Trans. on Software Engineering, SE-10(4):352-357, 1984.

[Welss84] David M. Welss and Victor R. Basili, "A Methodology for Collecting Valid Software Engineering Data." IEEE Trans. on Software Engineering, Vol. SE-10, No. 3, pp 728-738, November 1984.

[Wirth82] Wirth, N. 1982. Programming in Modula-2. Springer-Verlag, New York.

[YMR90] Yih-Farn Chen, Michael Y. Nishimoto, and C. V. Ramamoorthy. ``The C Information Abstraction System." IEEE Transactions on Software Engineering, pp. 325-334, March 1990.