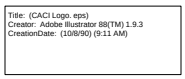
MODSIM III®

The Language for Object-Oriented Programming

Tutorial



Products Company

3333 North Torrey Pines Court, La Jolla, California 92037 • (619) 824.5200 • Fax(619) 457-1184
Watchmoor Park, Riverside Way, Camberley, Surrey GU15 3YL, UK • 1276 671 671 • Fax 1276 670 677
1600 Wilson Blvd., 13th Floor, Arlington, Virginia 22209 • (703) 875-2900 • Fax (703) 875-2904

December 1996

For product information or technical support contact:

In the US and Pacific Rim:

CACI Products Company
3333 North Torrey Pines Court
La Jolla, California 92037
Phone: (619) 824.5200
Fax: (619) 457-1184

In Europe:

CACI Products Division
Watchmoor Park
Riverside Way
Camberley, Surrey
GU15 3YL, UK
Phone: 1276 671 671
Fax: 1276 670677

Contents

Preface

This Document

This manual is intended to help teach the MODSIM III language to any simulation analyst with prior programming experience. The manual gives an overview of the language syntax, its object-oriented, modular, simulation and graphical features.

MODSIM III Documentation

There are four documents pertaining to MODSIM III:

- ***MODSIM III Reference Manual*** - The language reference. Contains information about the syntax and structure of MODSIM III as a programming language. Also covers object-oriented programming, simulation, graphics and I/O
- ***MODSIM III Tutorial*** - This document.
- ***MODSIM III User's Manual*** - It contains information about: **mscomp**, the compilation manager; MODBENCH, the development environment under Windows; MODSIM III compiler options; and debugging MODSIM.
- ***SIMGRAPHICS II User's Manual for MODSIM III*** - This manual contains information about SIMGRAPHICS II, the integrated graphics development and animation environment for MODSIM III.

Free Trial & Training

MODSIM III is available exclusively from CACI Products Company. MODSIM III can be sent to your organization for a free trial. We provide everything needed for a complete evaluation on your computer: software, documentation, sample models, and immediate support when you need it.

Training courses in MODSIM III are scheduled on a recurring basis in the following locations:

La Jolla, California
Washington, D.C.
London, United Kingdom

For information on free trials or training, please contact the following:

In the U.S. and Pacific Rim:

CACI Products Company
3333 N. Torrey Pines Ct.
La Jolla, CA 92037
(619) 824.5200
Fax (619) 457-1184

In Europe:

CACI Products Division
Watchmoor Park, Riverside Way
Camberley, Surrey
United Kingdom
1276 671 671
Fax 0276 670 677

Section I. Introduction

1. Overview of MODSIM III

The Modular Simulation language, MODSIM III, is a general-purpose, modular, block structured language which provides support for object-oriented programming, discrete event simulation and animated graphics. It is intended to be used for building large process-based discrete event simulation models through modular and object-oriented development techniques.

1.1 MODSIM III as a Programming Language

MODSIM's syntax and control mechanisms are similar to those of Modula-2, Pascal and Ada, so a simple MODSIM program which does not use the object or simulation extensions will look very much like a Modula-2, Pascal or Ada program.

For instance, the short program below computes the average of a sequence of numbers which have been input.

```
MAIN MODULE Sample1;

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.
```

Note: MODSIM III, like Modula-2, requires reserved words such as **`BEGIN`**, and **`OUTPUT`** etc. to be capitalized. Also like Modula-2 and C, MODSIM is case sensitive. Thus, the variable name **`number`** is different from **`NUMBER`** and **`Number`**.

The two most commonly-used areas where MODSIM III differs from Modula-2 are:

- Block structure: the **`END`** is always followed by a keyword. e.g.

  `IF` … **`END IF`**,
  `FOR` … **`END FOR`**, etc.

- Parameter declarations: the syntax for declaring variable parameters is comparable to Ada, i.e. the direction in which parameters are passed is explicitly stated:
  `IN, OUT, INOUT`.

MODSIM III supports most Modula-2 features such as modules, strong typing, data hiding, records, enumerated types, symbolic constants and Modula-2 style control structures.

MODSIM III supports the standard simple and structured data types but it also includes dynamic data types and monitored types which are used to collect statistics

- **Scalar types**: **`INTEGER`**, **`REAL`**, **`CHAR`**, **`BOOLEAN`**
- **String type:** A full implementation of dynamic strings which manage their own memory
- **Dynamic Structured types**: **`ARRAY`**, **`RECORD`**.
- **Fixed Structured types**: **`FIXED ARRAY`**, **`FIXED RECORD`**.
- **Monitored Type:** A variable to which a statistical monitoring probe is attached. Whenever an assignment is made to the variable or the variable is referenced, statistics can be gathered or user-specified procedures can be automatically invoked.
- **Enumerated types**: Types in which the allowable values are explicitly enumerated in a **`TYPE`** declaration.
- **Subrange types**: Subranges of the existing scalar types and of enumerated types.

The MODSIM **`INTEGER`** value is 32 bits long. The MODSIM **`REAL`** value is 64 bits long. These are equivalent, respectively, to the C long int and double on a 32-bit architecture. No "smaller" numeric types like the C short or float are supported. MODSIM supports IEEE floating-point arithmetic on machines which provide this capability.

MODSIM is a modular language. This means that it provides formal support for importing definitions and declarations from other modules. These are checked for consistency when individual modules are compiled. There will be no surprises at link time. Non-modular languages such as C and C++ do not provide this service.

MODSIM itself makes extensive use of this facility. The core of the language is quite small. Most support facilities for I/O, simulation and animated graphics are made available from standard modules.

The code example above is a simple case in which the entire MODSIM program is contained in one main program module.

A major innovation in MODSIM III is the provision of automated program building. This means that the user does not have to write "make" scripts to tell the compiler how to compile and link a MODSIM III program. Given the name of a main module, MODSIM's compiler knows how to compile and link all relevant parts of the program. It knows which individual modules need to be compiled and which are already compiled. It also keeps track of inter-module dependencies.

Thus, there are two major types of modules in MODSIM:

- **Main Modules**
- **Library Modules**

Main modules may be compiled and executed on their own. They may **IMPORT** constants, types, variables and procedures from library modules, but nothing can be **IMPORT**ed from a main module. A simple MODSIM program can consist solely of a main module.

Library modules are compiled separately. They contain declarations of types, constants, variables, procedures and the actual implementation code for procedures and methods for objects. Each library module typically contains a set of related procedures and objects

There are two parts to a Library Module: the **DEFINITION MODULE** and the **IMPLEMENTATION MODULE**. The **DEFINITION MODULE** contains descriptions of those aspects of the library module which can be imported by other modules. The **IMPLEMENTATION MODULE** contains the code which implements the functionality of the library, but which does not have to be visible outside the library module itself.

1.2 Control Structures in MODSIM III

MODSIM's control structure syntax differs subtly from that of Modula-2. In most cases the syntax is closer to that of Ada. MODSIM also includes a unique **WAIT** statement for use in discrete simulation.

We will briefly cover the syntax and use of control statements here so that the examples of MODSIM code used throughout this tutorial are easier to read. The MODSIM III Reference manual contains a detailed description of the entire language.

In the informal illustrations below, **StatementSequence** is zero or more statements each separated by semicolons. The semicolon is a statement separator. Optional portions of the syntax are enclosed in brackets. Alternatives are separated by a vertical bar.

IF:

```
IF Boolean Expression
  StatementSequence
[ ELSE
  StatementSequence ]
END IF;

     or

IF Boolean Expression1
  StatementSequence
[ELSIF Boolean Expression2
  StatementSequence
ELSIF Boolean Expression3
  StatementSequence
    . . . ]
[ ELSE
  StatementSequence ]
END IF;
```

CASE:

```
CASE [ordinal type | string]
  WHEN  a..e, m: StatementSequence
  WHEN   p: StatementSequence
  WHEN   x..z: StatementSequence
[ OTHERWISE
  StatementSequence ]
END CASE;
```

LOOP:

```
LOOP
  StatementSequence
END LOOP;
```

```
WHILE Boolean Expression
  StatementSequence
END WHILE;
```

```
REPEAT
  StatementSequence
UNTIL Boolean Expression;
```

```
FOR ident := expression TO | DOWNTO expression
                           [ BY expression ]
  StatementSequence
END FOR;
```

```
FOREACH object IN group  [REVERSED]
 StatementSequence
END FOREACH;
```

EXIT:

The **EXIT** statement causes control to pass to the end of the enclosing loop. Unlike Modula-2, MODSIM's **EXIT** works in all loop constructs.

WAIT:

The **WAIT** statement is used to elapse simulation time. Its syntax is similar to that of the **IF** statement. It contains a statement sequence to be executed if the wait was completed successfully and an optional statement sequence to be executed if the wait was interrupted.

```
WAIT reason
  [ StatementSequence ]
END WAIT;
```

```
or
```

```
WAIT reason
  [ StatementSequence ] ⇦ executed if WAIT completed
[ON INTERRUPT
  StatementSequence   ⇦ executed if WAIT interrupted ]
END WAIT;
```

The **reason** in the **WAIT** statement can be one of two types:

DURATION: an interval of simulation time

FOR: another time-elapsing method to be invoked and completed.

1.3 Input and Output in MODSIM III

MODSIM provides a complete and coherent I/O facility. Module **IOMod** provides device-independent text stream input and output to and from any device or file.

IOMod also supports block-oriented, random access I/O. Binary files can be created and read. The **Position** method is used to seek to a particular block while the **ReadBlock** and **WriteBlock** methods are used to read and write binary, random access files.

The number of files and/or devices which can be open simultaneously is limited only by the operating system of the machine on which MODSIM is running.

Facilities are also provided for defining I/O streams, opening and closing files, checking for the existence of files before attempting to open them, determining access status (Read, Write, Read/Write, etc.) and determining file size and last modified or access time.

The I/O capability is implemented in a style familiar to Modula-2 and Ada users. Each variable type has its own input and output procedure. Each input or output statement operates on one variable at a time.

Each I/O stream in MODSIM is an object. Stream I/O objects provide a facility which is consistent with the object oriented architecture of MODSIM.

IOMod defines the following type:

```
FileUseType = ( Input, Output, InOut, Append, Update, CreateBinary );
```

FileUseType is used when a file is opened to indicate how it should be handled. **Input**, **Output**, **InOut** and **Append** are used to open text files. **CreateBinary** is used to create a binary file. **Update** is used to open a binary file for read and write.

Excerpted below is a partial list of MODSIM's stream I/O methods. The **ASK** methods will be covered under the topic of object oriented programming. For now it would be appropriate to understand the word **PROCEDURE** wherever **ASK METHOD** appears below.

```
ASK METHOD Open(IN FileName:    STRING;
                IN IOdirection: FileUseType);

ASK METHOD Close;

ASK METHOD Delete;

ASK METHOD ReadChar(OUT ch: CHAR);

ASK METHOD ReadInt(OUT n: INTEGER);

ASK METHOD ReadReal(OUT x: REAL);

ASK METHOD ReadString(OUT s: STRING);

ASK METHOD ReadLine(OUT str: STRING);

ASK METHOD WriteChar(IN ch: CHAR);
```

```
ASK METHOD WriteInt(IN num, fieldwidth: INTEGER);

ASK METHOD WriteHex(IN num, fieldwidth: INTEGER);

ASK METHOD WriteReal(IN num: REAL;
                     IN fieldwidth,
                        precision: INTEGER);

ASK METHOD WriteExp(IN num: REAL;
                    IN fieldwidth,
                       precision: INTEGER);

ASK METHOD WriteString(IN str: STRING);

ASK METHOD WriteLn;

ASK METHOD Position(IN moveTo : INTEGER);
                       { random access file seek }

PROCEDURE ExistsFile(IN fname: STRING) : BOOLEAN;

PROCEDURE DeleteFile(IN fname: STRING);

PROCEDURE FileSize(IN fname: STRING) : INTEGER;
```

Each Stream I/O object also has two fields which can be used to control I/O:

```
eof:       BOOLEAN;

ioResult:  INTEGER;
```

MODSIM also provides several standard procedures, **INPUT, OUTPUT** and **PRINT ... WITH** for doing non-object oriented, free formatted I/O

The **INPUT** procedure takes one or more arguments. The **OUTPUT** procedure takes zero or more arguments. The arguments may be any of the following types :

```
INTEGER, REAL, CHAR, STRING, ENUMERATION
```

For example:

```
OUTPUT("Input height & weight for item number", n);
INPUT(height, weight);  OUTPUT;
```

The **INPUT** procedure reads values for each argument from standard input. The **OUTPUT** procedure writes the value of each argument to standard output, followed by a newline character. When it is used without arguments, it writes a newline character alone.

The **PRINT ... WITH** procedure supports formatted output to standard I/O:

```
VAR
  formatStr : STRING;
BEGIN
...
  formatStr := "At time ****.** the number waiting is **";
  PRINT (SimTime(), numWait) WITH formatStr;
```

1.4 Object-oriented Programming

Programs written in MODSIM are organized around object types. Each object type has two interrelated sets of properties. These respective properties are **fields** and **methods**. The state of an object instance at any instant is described by the values in a series of fields, similar to those of a record. Its behavior, or the actions it is capable of performing, are described in its methods which are executable routines with special characteristics associated with the object type.

This formal association of data and code provides an inherent encapsulation of the data, because the information in an object's fields can be changed only by a method which belongs to the object or by a method which it has inherited.

In other words, the fields of an object may not be directly modified except by the object itself. Other parts of a MODSIM program external to a particular object can change the value of the object's fields only indirectly. For instance a **`controller`** object might want a vehicle object's speed to be zero. The **`controller`** object could **`ASK Vehicle TO Stop`**. The vehicle object's **`Stop`** method would then set the vehicle's **`speed`** field to zero in response to this message.

Other parts of a MODSIM program may request the value of an object's fields by sending it a message. e.g. **`CarSpeed := ASK Vehicle speed`**, where **`Vehicle`** is an object and **`speed`** is one of its fields.

Essentially, an object's fields are “read only” from the perspective of code not included in the object, but are “read and write” from within the object itself.

Experience with the object-oriented approach shows that it is even more effective at modularizing the interactions of a program than the structured programming techniques which spawned Algol, Pascal, Modula-2 and Ada. The part of the program which invokes some behavior or action can always invoke an action in the same way over a wide range of object types; e.g. **`ASK Car TO Stop`**, **`ASK Aircraft TO Stop`**. Each object, however, may have a different behavior in response to this same message. In this case the aircraft will want to land before stopping!

Note that, in any program, there will likely be a number of different methods which have the same name. Since each is encapsulated within an object, this does not lead to any ambiguity. Each invocation of a method is accomplished by “sending a message” to the object requesting it to execute one of its methods.

Thus, the calling entity need not have detailed knowledge of how an object will accomplish the action it is being told to do. It simply asks for a generic action to be performed and the object which receives the request message has its own, tailored routine to perform that action.

New object types can be defined in terms of existing object types. When this is done, all of the fields and methods of the existing object are incorporated as a proper subset of the new object type. This capability is known as **inheritance**. Where an inherited method is inappropriate, it can be **overridden** and a more appropriate method substituted.

The appropriate use of inheritance encourages software reusability more effectively than any library of procedures. Unlike procedural libraries, a properly-structured object library allows selective redefinition of some of the code while incorporating other code u n-changed. In contrast, replacing fundamental algorithms or assumptions in a procedural library normally requires a wholesale abandonment or re-implementation of the library and the code which depends on it.

The design of a MODSIM program, then, encourages a careful separation of the data re p-resentation and behaviors for each object type, as well as the declaration of related types using a common inheritance. This technique is referred to as *object-oriented program-ming*, and will be examined in greater detail in Chapter 3.

2. Background

This chapter reviews a number of programming concepts which have been used in the design of MODSIM III and may prove helpful to users of the language. Those familiar with process oriented simulation and contemporary programming languages may wish to skip this chapter.

2.1 Data Structures in Contemporary Languages

Among the many features which distinguish contemporary computer languages from their predecessors is the ability to organize data into formal structures. These languages are able to organize disparate types of data into records which can be handled as a unit of information. Some languages, typically used for simulation, also have the ability to organize, manage and manipulate groups and ordered sets of data.

All of these capabilities are implemented in MODSIM. The ability to handle sets or groups will be covered in some detail in Chapter 10; however, since an understanding of the concept of **record** data structures is basic to this tutorial, we will briefly review the subject here.

To illustrate the concept of records, we can declare a simple personnel record in MODSIM. The record contains fields for the person's last name, first name, middle initial, age and department. We allow ages in the range from 17 to 65 years old. The allowed departments are specified in an enumeration.

In MODSIM III the structure of a record is declared in a **TYPE** statement. The user then typically declares a variable of that type. The MODSIM III **RECORD** type is dynamically allocated. MODSIM also supports the **FIXED RECORD** which is statically allocated. The **TYPE** declaration for our personnel record would look like this:

```
PersRecType = RECORD
  lName, fName : STRING;
  middleInit   : CHAR;
  age          : [ 17 .. 65 ];
  department   : ( Ops, Research, Finance, Sales )
END RECORD;
```

Note that we have used the subrange type to describe the field for **age.** MODSIM will provide run-time checking to ensure that the user does not assign a value outside of that range to the field **age**.

The field for **department** has been described using an enumerated type. In other words, we have enumerated each of the legal values of the type.

To use the record type we have just declared, we might declare an array of type **PersRecType**:

```
VAR
    staff:  ARRAY INTEGER OF PersRecType;
```

This uses the MODSIM III **ARRAY** type which is dynamically allocated and sized at program run time. The statement declares an array which will hold records of type **PersRecType** and will be indexed using integers. We could also have indexed the array by characters or by a type such as **DayOfWeek**. In the program code we would allocate the array with a statement like this:

```
NEW(staff, 1..95);
```

This would allocate a copy of the array called **staff** and size it to fit 95 elements which would be numbered in the range 1..95.

We could also declare a single variable of that **RECORD** type which we would use to shuttle information back and forth from a file:

```
VAR
      employee: PersRecType;
```

We could then access the information in the various fields in the following way:

```
employee.lNname := "Smith";
staff[9].department:= Finance;
employee.age := 27;
```

The *MODSIM III Reference Manual* provides a detailed discussion of the differences between the dynamic **RECORD** and the **FIXED RECORD**.

2.2 MODSIM III's Modular Structure

The sample program presented at the beginning of this tutorial showed how a MODSIM III program can exist in just one MAIN module. For larger programs it is more typical to see MODSIM programs which consist of a main module and any number of supporting modules. These modules can be separately compiled to ease the task of development and maintenance.

Each module typically contains declarations for a set of related procedures and objects and the executable code which constitutes the procedures and methods.

MODSIM modules can be:

- Used to support a single program.
- Compiled into support libraries to be shared by many programs

MODSIM itself provides much of its capability through support modules such as **IOMod**.

A library module actually consists of two modules, each of which is stored in its own file and can be compiled separately. One is the definition module and the other is the implementation module.

The **DEFINITION MODULE** contains declarations for all of the constants, types, variables, and procedures which will be available for import by other modules. Only the headings of procedures and methods are declared. There is no executable code in a definition module. Items declared in a definition module will be accessible or visible to any other module

which imports them. For instance, another module may import a type definition from a definition module and then declare variables of that type.

The **IMPLEMENTATION MODULE** contains the actual code which implements all procedures and methods. It may include **CONST**, **TYPE** and **VAR** declarations which are needed solely within that library module. A variable or data structure which is declared globally within an implementation module is considered global to all procedures and objects in that particular module but is not visible outside of that module.

Anything declared in a definition module is implicitly known in the companion implementation module. The implementation module, of course, may also explicitly import definitions from other definition modules as well.

To illustrate the organization of the three types of modules we have split a small sample program into modules. The library module contains one simple procedure which reverses a string passed to it.

```
MAIN MODULE Sample2;
FROM TextLib IMPORT Reverse;
VAR
  someText : STRING;
BEGIN
  OUTPUT("Enter string:  ");
  INPUT(someText);
  Reverse(someText);
  OUTPUT("Reversed string: ");
  OUTPUT(text);
END MODULE.
```

The following two modules constitute the library module called **TextLib**:

```
DEFINITION MODULE TextLib;
  PROCEDURE Reverse(INOUT str: STRING);
END MODULE.
```

```
IMPLEMENTATION MODULE TextLib;
  PROCEDURE Reverse(INOUT str: STRING);
  VAR           { reverses the input string }
    k       : INTEGER;
    tempStr : STRING;
  BEGIN
    FOR k := STRLEN(str) DOWNTO 1
      tempStr := tempStr + SUBSTR( k, k, str );
    END FOR;
    str := tempStr;
  END PROCEDURE;
END MODULE.
```

We would run this sample program by:

1. Compiling the **TextLib** definition module
2. Compiling the **TextLib** implementation module

3. Compiling the **`Sample2`** main program module
4. Linking **`Sample2.`**
5. Running **`Sample2.`**

Since this is a complicated series of steps which have to be accomplished in a certain order, MODSIM provides a compilation manager to simplify this process. The compilation manager, which is called **mscomp**, offers a variety of services to help in the management of both large and small projects. Among its options it offers the following compilation choices to the user:

- Compile and link everything
- Compile & link only those modules which have been changed since the last compilation or are affected by changes in modules they import from
- Compile a single module
- Passively determine which modules need to be compiled and linked

Once this process has been done once, the user may make changes to the implementation modules or the main module and later will be able to recompile only those modules which have been changed without having to recompile the entire program.

This **mscomp** facility is integrated into the MODSIM III Windows user interface called MODBENCH.

It is important to note that MODSIM supports **separate compilation** as opposed to **independent compilation**. This means that dependencies are checked. If a change is made which will affect other modules, these are scheduled for recompilation as well. The real benefit of modularity and separate compilation is seen in large programs with multiple library modules where the effect of changes can be localized, yet effects of changes which affect other modules are correctly handled.

The *MODSIM III User's Manual* covers **mscomp**, the compilation manager, in greater detail, but one feature which deserves a mention is that **mscomp** does not require a script, project or "make" file to operate. It operates using the syntax of the language and information from the computer's file system.

2.3 What is Computer Simulation?

There are two general categories of computer simulation: *continuous simulation* and *discrete-event simulation*

Continuous simulation describes events using sets of equations which are solved numerically with respect to time. Examples of problems in this area are fluid-flow or hydraulics problems and financial modeling. Typically a time step is chosen. The continuous simulation program then steps forward by the increment of time chosen for the time step and recalculates all equations which describe the model.

Discrete-event simulation describes a system in terms of logical relationships which cause changes of state at discrete points in time rather than continuously over time. Examples of

problems in this area are most queuing situations: Objects (customers in a gas station, aircraft on a runway, jobs in a computer) arrive and change the state of the system instantaneously. Varying amounts of time elapse between events.

In discrete-event simulation, large or small amounts of simulation time can pass between events, but the state of the system is only of interest when one of its component parts changes state. MODSIM takes the capabilities of discrete systems modeling languages like Simula and SIMSCRIPT II.5 and adds object-oriented programming capability and the modular constructs of Modula-2.

2.4 Process-Oriented vs Event-Oriented Simulation

The classical approach to discrete-event simulation is event-oriented. In this approach, routines are written to describe discrete events in the operation of a system. For instance, in a simple bank model the event routines might be:

- Customer arrives
- Customer enters queue
- Customer engages services of teller
- Customer leaves

No time passes during any event routine. Instead, passage of time is handled by scheduling the next event for the object currently being manipulated. In the simple bank model, the event "Customer engages services of teller" would schedule the next event, "Customer leaves", at some future time.

This event-oriented approach is adequate for smaller models, but in larger models it is often difficult to follow or modify the flow of logic which describes the behavior of an object, such as a customer. Consider the simple bank model if we added a janitor, a security guard and some management functions. There would be many unrelated event routines. Following the logic flow which describes the behavior of a customer would be like tracing through a sequence of GOTO statements in a large BASIC program.

The process approach simplifies larger models by allowing many aspects of an object's behavior in a model (e.g. bank customers) to be described in one method which allows for the passage of time at one or more points in its code.

There is a further advantage to the process technique. Once the actions of a class of objects (such as customers in a bank) have been gathered together in an object, the simulation program can create multiple, concurrent instances of the object. In our bank, for example, the simulation program would generate a new instance of the customer object each time a customer arrived. It could also pass information about the customer in the parameter list of the object's initialization method. Perhaps it would pass in information about the sort of customer (*young* or *elderly*) and the expected service time for the customer. While there would be multiple, distinct copies of the customer object operating simultaneously, each could have different values of their fields to describe the particular customer's properties.

Finally, objects can interact. In our example, an instance of the customer object with the *young* attribute might yield its place in the queue to a customer object with the *elderly* attribute.

This process approach is the one supported in MODSIM. It exploits object-oriented programming features to simplify both the original development and the subsequent maintenance of large models.

A simulation model written in MODSIM defines a system in terms of processes because the process technique provides a powerful structure for expressing most categories of simulation problems, and provides significant advantages over the direct use of discrete events.

The advantages of processes are both conceptual and labor-saving. The process statements are expressed sequentially, in a manner which is analogous to the system being described. This practice is recommended by standard design methodologies.

Section II. Object-Oriented Language Features

3. Object Oriented Programming

Objects in MODSIM are dynamically allocated data structures coupled with routines, called *methods*. The fields in the object's data structure define its state at any instant in time while its methods describe the actions which the object can perform. The values of the fields of an object can be modified only by its own methods. Since no other part of the program can modify these values, program maintenance and debugging is greatly simplified.

Other entities can query the value of an object's fields or ask it to perform its methods by sending messages to the object. This is an important feature of objects. Instead of invoking an object's methods by a call, the user invokes the method by sending a message to the object requesting it to perform the method. e.g. **`ASK Car54 TO ReportPosition`**. This small refinement in the way code is invoked is responsible for many of the advantages in object-oriented programming.

It allows the code to be executed against a specific set of data, the fields of a particular object. It also allows several different object types to have a method of the same name. Each one responds to a method call by executing its own code.

Below is a list of facts about objects in MODSIM. The remainder of this chapter and the next chapter will expand on these facts.

- A MODSIM object consists of fields (variables) which describe its state at any instant and methods (procedures) which describe the actions it is capable of performing. The fields and methods are sometimes described as attributes and behaviors of an object.
- Object types are declared in the **`TYPE`** section of a program in a manner similar to the declaration of record types. Variables of the object type are then declared in the **`VAR`** section of a program.
- The user creates new instances of any object type dynamically, as needed, and disposes of the object instances when they are no longer needed.
- There can be (and usually are) multiple instances of each type of object existing concurrently in a MODSIM program.
- Code in any part of a MODSIM program can ask an object instance the value of any of its fields.
- Only an object instance itself can change the value of any of its fields.
- One invokes an object instance's methods by sending a message to the object asking it to perform a particular method. e.g. **`ASK SomeObj TO DoSomething`**
- New object types can be defined in terms of existing object types. This capability is know as inheritance

3.1 What is Object-oriented Programming?

The significance of object oriented programming is that disparate types of objects which share the same ancestry can each have their own distinct methods which have the same name as methods in other objects. This means that generic operations can be invoked with one method name which will cause appropriate (and distinctly different) behavior in each different object type.

An example of a generic operation might be a command to refuel. For instance, when the MODSIM statement **`TELL ... TO Refuel`** is received by objects such as trucks, cars, helicopters, etc., each would react differently. The helicopter might take on 1,200 Lbs of jet fuel, the car would take on 11 gallons of unleaded regular and the truck would take on 380 liters of diesel fuel. Although each vehicle has inherited all of the attributes of a generic **`VehicleObj`** type, each has chosen to provide its own, specific **`Refuel`** behavior.

The logic which describes how each vehicle type which underlies **`VehicleObj`** performs refueling is associated with each vehicle object type. So if new vehicle types are added, the only change required is to add code for each new object type which describes how that particular vehicle type performs the act of refueling. The procedure associated with an object type is known as a method for that object.

One of the most important features of object-oriented programming is illustrated in the above vehicle example. If we add a new vehicle type called mule, its refueling method might be to eat a bale of hay. But it would still be appropriate to use the original calling code without changing it. It would still be appropriate to **`TELL ... TO Refuel`** even though a new vehicle type has been added.

In traditional strongly-typed languages, the type of each operand is fixed at compile time, thus statically determining the operation that will be performed when the program is run. However, an object-oriented language requires the determination of an action to be deferred until the program is run. This allows dynamic selection at run-time of the code appropriate for the given object type. This process is known as *dynamic binding*.

In many object-oriented languages, new object types can be defined in terms of an existing object type. This allows structuring of related object types. By default, the new object retains all properties of the existing object. It can selectively replace or extend any of those properties, and add new properties. The definition of a new object type in terms of an existing type is referred to as *inheritance*.

It is the combination of both dynamic binding and inheritance that gives the object-oriented approach its power and flexibility.

3.2 Messages and Behaviors

One common way to describe interactions in object-oriented programs is the object-message analogy, in which an operation in a program is described in terms of a *message* sent to an object. The object, in turn, "decides" what to do in response to the message

Although this anthropomorphic description makes a compiled program seem more human than it is, it does help to illustrate the object concept, and thus is frequently found in the literature. The object-message metaphor emphasizes the apparent active selection made

"by" each object, even though the actual implementation is through more prosaic procedural calls.

Within this framework, objects are active data structures that have associated *behaviors* in response to each message. As implemented by most languages, an object is a data structure (usually dynamically allocated) that has one or more associated procedures. These procedures are called **methods** to differentiate them from standard procedures. They describe the method used by an object to perform the action requested by a message.

An object-oriented program can issue a message requesting a specific action to a group of disparate objects. Each reacts according to its particular method for that request. All objects of the same type will have the same behavior which is implemented in the objects' methods. Methods differ from ordinary procedures in two ways:

1. First, there can be more than one method with the same name. In the vehicle example above we could have as many methods named **`Refuel`** as there are different object types derived from **`VehicleObj`**. **`Refuel`** would be the message name, while the behavior for all objects of a given type would be defined by the corresponding method for that object type. Thus, we would have a **`Refuel`** method for a helicopter, a different **`Refuel`** method for a car and a very different **`Refuel`** method for a mule.
2. Second, each method includes an implied parameter referencing the associated object data structure. A **`Refuel`** method for a truck would, for example, always reference a particular truck-type object. Within an object's methods, this object is referred to by the system defined variable **`SELF`** in MODSIM and in most object-oriented languages. For example, if the truck object were running low on fuel, one of its own methods might **`TELL SELF TO Refuel`**.

3.3 Inheritance

Often, when defining a new type of object, we want a new one just like some other existing one but with a few changes and a few new features. In these cases we can simply define the new object type in terms of the existing object type, and then add new fields and methods or modify existing ones. When the new object type is defined in terms of an old object type, we say the new object type *inherits* properties from the older object type. The term property refers to the fields and methods of an object type.

Consider a generic vehicle which has fields to define its position, direction of movement and speed. It has methods to **`ProceedTo`** a new location and to **`Stop`**. A helicopter is a vehicle which has these same properties, both in terms of the data fields which describe its status and the methods it executes in response to messages. A helicopter also has at least one new property, altitude. Thus, you could define a helicopter object type in terms of a generic vehicle object, and then add new methods and data fields to handle the helicopter behaviors and its new properties.

Another case where inheritance can be useful is when two object types have several things in common, and thus it may be useful to define an ancestor object type to describe those properties shared by both types.

The object types that are inherited are called *underlying types*. The most immediate underlying object type is also called by the more specific term *base type*. The newly defined object type is called a *derived type* of its base or underlying types.

It does not matter how remote the derivation is. A new object type inherits the properties of its base object type and obviously all of the underlying types. The new object type can then supplement these with its own new properties.

A new object type can also define particular methods differently than those existing for its base types. We say that the derived type *overrides* the methods of its base type.

A derived type only overrides and redefines those methods which must be different from those of its base type. The new object can start from scratch and provide a totally new method. It also has the option of using the inherited method, but specifying additional behavior.

A library of object types can be even more powerful than a subroutine library, in that small differences from library routines which are required for user-defined operations can be specified without replacing the library methods. We merely supplement existing methods. This is one of the principle benefits an object-oriented library has over its procedural counterpart.

3.4 Declaring an Object Type

An object type declaration is similar to a record type declaration in that each includes a list of fields:

```
TYPE
  VehicleObj = OBJECT
    course : [ 0 .. 359 ]; { direction of movement }
    speed  : INTEGER;
    posX,
    posY   : REAL;
  END OBJECT;
```

Object types are declared in the **TYPE** section of a module.

3.5 Methods of an Object

An object differs from a record in that it also has methods associated with it to describe actions it can perform. The name and parameter list of each method forms a method heading which is specified as part of the **object type declaration**. The actual code which implements each method is specified in the corresponding **object implementation block**.

There are two kinds of methods: **ASK** methods and **TELL** methods. **ASK** methods are similar to those found in other object-oriented languages. The **TELL** method is used to model the passage of time in simulations.

The distinction between them will be covered in Chapters 8 and 9.

Elaborating on the preceding example, here is a complete declaration for the **VehicleObj**:

```
TYPE
  VehicleObj = OBJECT
    course : [ 0 .. 359 ];
    speed  : INTEGER;
    posX,
    posY   : REAL;
    ASK METHOD ProceedTo(IN x, y: REAL);
    ASK METHOD Stop;
  END OBJECT;
```

In this declaration, both **ProceedTo** and **Stop** are methods for type **VehicleObj**. The actual implementation code for the two methods would be contained in a corresponding object implementation block

```
OBJECT VehicleObj;     the object implementation block
  ASK METHOD ProceedTo(IN x, y: REAL);
  BEGIN
   implementation code would go here
  END METHOD

  ASK METHOD Stop;
  BEGIN
   implementation code would go here
  END METHOD;
END OBJECT;
```

There is a reason why the object declaration is split into two sections. In large MODSIM programs, the object type declaration would likely be placed in a definition module and the object implementation block would be placed in an implementation module. In a small program which consists of only one main module, the object type declaration would come first, followed later by the object implementation block.

3.6 Reference Variables

The declaration of an object type implicitly defines a new data type of the same name, known as the *reference type*. This type is similar to a pointer type. Variables declared as reference types are known as *reference variables*. When a variable of that type is declared, it initially assumes a value of **NILOBJ**, which is analogous to **NILREC** for records and **NILARRAY** for arrays.

This is an important point. Declaring a variable of type **VehicleObj**, for instance, does not actually allocate space for the object and create it. This is done dynamically, at run-time, with a call to **NEW**:

```
VAR
  car54: VehicleObj;
BEGIN
  .  ⇦ object instance “car54” doesn't exist yet
  .
  NEW(car54);
  .  ⇦ object instance “car54” now exists
  .
```

```
    DISPOSE(car54);
    .  ⇦ object instance "car54" no longer exists
    .
    NEW(car54);
    .  ⇦ a fresh object instance "car54" now exists
  END.
```

A reference variable contains a reference value which identifies a particular instance of an object type. Programs will often have many instances of a given object type at any given time. All of these instances share an identical structure, but have separate and distinct states, represented by different values in their fields. Each new instance of an object has its own place in the computer system's memory where the value of its fields are stored.

Building on the earlier definition of a Vehicle object, we can expand by defining the object type **AircraftObj** which inherits from the **VehicleObj** type. Doing so implicitly defines a corresponding reference type **AircraftObj**. MODSIM handles this job automatically. In other languages, such as Modula-2 or Pascal, the user would have to explicitly declare a pointer variable of type **AircraftObj**. Here is how it would have to be done in Modula-2:

```
TYPE
  AircraftObjPointerType = POINTER TO AircraftObj;
VAR
  plane : AircraftObjPointerType;
```

MODSIM eliminates this extra step by automatically making every object type, such as **AircraftObj,** into a reference type from which reference variables can be declared. In the example below, the global variable **Airline**, and the local variables **American75** and **United15** are all reference variables for objects of type **AircraftObj**.

In this example, we also improve the way position information is expressed by defining and using a **FIXED RECORD** type which wraps up the x and y position information. This makes the position information easier to refer to and handle.

```
TYPE
 locationType = FIXED RECORD
    x, y : REAL;
  END RECORD;

  VehicleObj = OBJECT
    course   : [ 0 .. 359 ];
    speed    : INTEGER;
    position : locationType;
    ASK METHOD ProceedTo(IN Dest: LocationType);
    ASK METHOD Stop;
    ASK METHOD ReportStatus(): INTEGER;
  END OBJECT;

  AircraftObj = OBJECT(VehicleObj)          inherits from VehicleObj
    altitude : INTEGER;       adds two new fields
    backupAC : AircraftObj;
```

```
    ASK METHOD Land;            adds a new method
  END OBJECT;

VAR
  Airline:  ARRAY INTEGER OF AircraftObj;
.
ASK METHOD . . .;
VAR
  American75, United15, plane:  AircraftObj;
BEGIN
  ...
END METHOD;
```

Reference variables can be used in a manner similar to any other type of variable. The declaration:

```
Airline:  ARRAY INTEGER OF AircraftObj;
```

is an example.

Fields of objects containing reference variables can indicate relationships between objects. In the example above, the field **backupAC** is a reference variable of type **AircraftObj** which is used to access the plane's backup aircraft, which is another object instance of type **AircraftObj**.

We have shown that a reference variable is used in a manner which is analogous to a pointer variable, however the similarity is superficial. A pointer variable simply points to a type of storage such as a record and carries information needed to de-reference the fields of a record. A reference variable for an object "points" to an object in the same way, but it also has hidden behind it the sophisticated mechanism by which the object's methods are dynamically invoked.

3.7 Allocating, Deallocating and Initializing Objects

An object instance is allocated by calling the standard procedure **NEW**, which takes as its argument a reference variable of the desired type. The reference value for the object i nstance is returned in the argument, and each field of the instance is automatically initialized to zero, **NILREC**, **NILOBJ**, or **NILARRAY,** as appropriate.

For example:

```
VAR
  United15:  AircraftObj;
    .
    .
NEW(United15);
```

The above call to **NEW** allocates an instance of type **AircraftObj** and returns its reference value in**United15**.

Note that it is not sufficient to simply declare the reference variable to obtain access to a new object instance. The reference variable contains **NILOBJ** until it is explicitly assigned a reference value to a new object instance by a call of **NEW**.

An object instance is deallocated by calling the standard procedure **DISPOSE**, which takes as its argument a reference value.

For example:

```
DISPOSE(United15);
```

de-allocates the object instance which was allocated in the previous example.

Some objects may require initialization before they are used. An example is the **RandomObj** from MODSIM's **RandMod**. Each instance of an object derived from **RandomObj** must be initialized so that the random object's seed can be set. Such initialization can be accomplished automatically. The built-in procedure **NEW** checks to see if a method named **ObjInit** has been defined for the object. If such a method exists, it is automatically invoked by **NEW**. Note that **ObjInit** takes no parameters. The user can override **ObjInit** and add more code to the behavior, but the user's replacement for the overridden **ObjInit** should always invoke the original **ObjInit** with an **INHERITED** statement. The **INHERITED** statement, which we will cover later in more detail, simply executes the original **ObjInit** method which was overridden.

Note that the method name **ObjInit** is reserved for this use. If the user builds an object "from scratch" and includes a method with the name **ObjInit**, it will be automatically invoked by **NEW**.

If the user requires a more elaborate initialization method or one which takes a parameter list, this can be accomplished with an explicit call to the user's own special initialization method after the call to **NEW**. Obviously, the more elaborate init method would need a name other than **ObjInit**!

A complementary procedure is used to perform "cleanup" before deallocating objects. If a method with the name **ObjTerminate** has been defined for an object, the method will automatically be invoked by **DISPOSE** before it deallocates an object instance. The **DISPOSE** procedure will correctly execute the **ObjTerminate** method for an object, even when the object is passed in a reference variable of type **ANYOBJ**.

3.8 Copying Objects with the CLONE Procedure

MODSIM supports a **CLONE** function procedure which is used to make a copy of an object. The **CLONE** function takes a reference variable as an argument. It then accomplishes four steps:

1. Allocate space for a new object instance of the same type passed in.
2. Copy the values in the fields of object instance passed in to the new copy.
3. Invoke the new object instance's **ObjInit** method, if one exists.
4. Invoke the object type's **ObjClone** method, if one exists.

Finally, it returns a reference to the new copy. The **ObjClone** method is analogous to the **ObjInit** and **ObjTerminate** methods. The **ObjClone** method can be used to perform any more complex behaviors which the user wants to associate with the copy.

If the programmer overrides an existing **ObjClone** method, the overridden method should be invoked with the **INHERITED** statement to ensure that all behaviors associated with copying defined by ancestors are carried forward.

4. Methods

The declaration of an object type is accompanied by the code for its associated methods . The implementation code for the methods is supplied in the object implementation block. Methods are similar to procedures. Each method may have zero or more parameters, and **ASK** methods may also return a single function value.

Methods come in two forms: **ASK** methods and **TELL** methods. There are important distinctions between **ASK** and **TELL** as pertains to simulation, but for now we will simplify the distinction somewhat.

An **ASK** call works the same way as a procedure call. When the **ASK** statement is executed, a message is sent to the object requesting it to invoke the method. The calling code then waits for the invoked method to finish before proceeding past the **ASK** statement. **ASK** methods are not allowed to pass any simulation time, so, in a simulation, the action just described takes place at one instant of simulation time. Another way to describe an **ASK** call is as a synchronous call.

The **TELL** call, which is known as a *delayed method call*, is essentially an asynchronous call which is used to build simulation models. The calling code executes the **TELL** statement which sends a message requesting the object to invoke the method. The calling code then proceeds past the **TELL** statement without waiting for the invoked method to complete execution or, for that matter, even to start. **TELL** methods are allowed to pass simulation time.

Typically, the invoked **TELL** method will start execution, under the control of MODSIM's simulation engine, as soon as the currently executing code has finished.

With **ASK** and **TELL** methods, the programmer has complete control over the way methods are invoked in a simulation, and, therefore, whether time is allowed to elapse. Using **ASK** or **TELL** to invoke a method of an object is often referred to as "sending a message" to that object.

It is important to note that the distinction between **ASK** and **TELL** methods starts when they are declared. The user can not turn an **ASK** method into a simulation method by using the **TELL** syntax to call it.

Methods of an object are invoked from outside of the object using the **ASK** or **TELL** keyword, the object's reference variable, the method name, and any arguments to the method. For example:

```
TELL United20 TO ProceedTo(OHare);
```

would request the object instance known by the reference variable **United20** to perform the method **ProceedTo** with parameter **OHare**.

Within any method, the system-defined, or built-in reference variable **SELF** contains the reference value of the object instance for which the method was invoked. **SELF** is implied throughout the method and need not be specified to reference the fields or methods of the

object instance. It may also be used when an object wants to identify itself to another o b-ject, as in

```
ASK United20 TO ReportDistance(SELF);
```

This would request **United20** to report its distance from the object making the request.

The fields of an object can only be modified within its own methods. Within a method for **AircraftObj,** the fields of the object may be accessed as if they were ordinary local variables. For example:

```
altitude := 1000;    or    backupAC := United25;
```

Also, the object's methods can be invoked as if they were ordinary procedures. For exa m-ple:

```
ProceedTo(HomeBase);
```

is equivalent to:

```
TELL SELF TO ProceedTo(HomeBase);
```

4.1 Referencing Fields

The value of an object's fields may be interrogated using a syntax similar to that for a function method. The form is:

```
ASK object field
```

For example:

```
aircraftAltitude := ASK United20 altitude;
```

would request the object **United20** to report the value of its field called **altitude** and then assign that value to the variable **aircraftAltitude.**

Code which lies outside of an object cannot directly change the value of an object's fields. The values of an object's fields can only be changed by the object's own methods. Ther e-fore, in order to change the value of an object's fields, code which lies outside an object must invoke a method of the object to do so.

The following example shows how fields of an object are referenced from outside the o b-ject:

```
IF ASK United20 position <> HomeBase
  TELL United20 TO ProceedTo(HomeBase);
  OUTPUT("Not at home base, but returning");
ELSE
  OUTPUT("Already at home base");
END IF;
```

If the same piece of code were in one of the object's own methods, it would look like this:

```
IF position <> HomeBase
  ProceedTo(HomeBase);
  OUTPUT("Not at home base, but returning");
ELSE
```

```
    OUTPUT("Already at home base");
  END IF;
```

In the second line above we could also have used:

```
  TELL SELF TO ProceedTo(HomeBase);
```

4.2 Defining Methods

Other than the use of the keywords **ASK METHOD** or **TELL METHOD** instead of **PROCEDURE**, methods are defined using a syntax similar to procedure declarations.

For example, the definition below found in an object type declaration:

```
VehicleObj = OBJECT
  ...
  TELL METHOD ProceedTo(IN Dest: locationType);
  ...
END OBJECT;
```

defines a method **ProceedTo** that operates on objects of type **VehicleObj**. The body of the method which contains the executable code is found in the corresponding object implementation block. Typically, the object type declaration is contained in the **DEFINITION** module and the object implementation block is found in the **IMPLEMENTATION** module.

A method which returns a function result is referred to as a **function method**. A method that does not return a function result is known as a **proper method**. In the example above, **ProceedTo** is a proper method.

The method may optionally include a list of parameters. The type of the parameters, as well as the return type of a function method, may be any valid MODSIM type or a type defined by the user. The parameter list does not include the object itself, since that is supplied as an implied parameter, **SELF**, to all methods.

TELL methods are not allowed to have **OUT** or **INOUT** parameters and may not be function methods. This is because they are invoked asynchronously by code which does not wait for the results. Because of this there is no place to which information can be returned!

4.3 Implementing Methods

After an object type has been declared, its methods must be coded in a corresponding object implementation block. Typically the object type declaration is placed in a **DEFINITION MODULE** so it is available for import to other modules, and the corresponding object implementation block is placed in an **IMPLEMENTATION MODULE**.

The implementation code for the methods of an object type is placed within an **OBJECT ... END OBJECT** block labeled with the name of the object type. This placement of the implementation code within a named block allows different objects to have methods with the same name.

For example, if the previous declaration of **VehicleObj** were part of **DEFINITION MODULE TrafficModule**, the corresponding implementation module might include statements such as:

```
IMPLEMENTATION MODULE TrafficModule;

  VAR
    { Variables common to all Object types in this module }
    kindOfVehicle: ARRAY INTEGER OF STRING;

  OBJECT VehicleObj;
    TELL METHOD ProceedTo(IN Dest: locationType);
    BEGIN
      implementation code ...
    END METHOD;
    TELL METHOD Stop;
    BEGIN
      implementation code ...
    END METHOD;
    ASK METHOD ReportStatus(): INTEGER;
    BEGIN
      implementation code ...
      RETURN Status;
    END METHOD;
  END OBJECT;

  OBJECT AircraftObj;
    ...
    TELL METHOD Land;
    BEGIN
      implementation code ...
    END METHOD;
  END OBJECT;
END MODULE.
```

This module illustrates several important points about the nature and scope of variables in a MODSIM program:

- Any variable declared within a module (prior to any object implementation blocks) is global to the entire module. There will be only one copy of the variable. The variable is visible to every object instance's methods as well as to every procedure in the module. If the variable is declared in the definition module instead of the implementation module, it will also be visible in any other module of the program which chooses to import it.
- Any field declared within the definition of an object will be visible in the usual sense within that object's methods. From outside the object, we can **ASK** an object instance for the value of any of its fields, but we cannot directly change their value with an assignment. Each instance of an object type has its own separate copies of all of the fields.

- Any variable declared within the body of a method will be visible only within that method. There will be a unique copy of that local variable for each invocation of that method. Note that it is possible to have more than one invocation of a particular **TELL** method for a particular object instance running concurrently with respect to simulation time. Chapter 9 will explain how this can happen and why it is useful.

4.4 Using Methods

Methods differ in one crucial way from procedures; they are always invoked with reference to a specific object:

```
ASK | TELL object [ TO ] method[ (parameter list) ]
```

TO is a "noise word" which can be optionally specified in method calls to make the code more readable.

Function methods are invoked using a syntax similar to that used for accessing fields of an object:

```
value := ASK object method()
```

This syntax is the same as that used to access the values of a field. However, a reference to a function method may also include arguments, which follow the method name, as in

```
value := ASK object method(arguments);
```

Within a method for a particular object, any of the fields of the object can be referenced directly since the object's fields are global to its methods. The **ASK** syntax is not required. For example, the method **Land** might be implemented as

```
TELL METHOD Land;
BEGIN { implied argument SELF: AircraftObj }
  ...
  speed := 0;
  ...
END METHOD;
```

A method invoking another method of the current object can omit the

```
ASK | TELL object.
```

For example, a function method **ReportStatus** can be referenced from within an **AircraftObj** method with either:

```
OUTPUT( ReportStatus() );
```

or

```
OUTPUT( ASK SELF TO ReportStatus() );
```

Similarly:

```
ProceedTo( HomeBase );
```

and

```
TELL SELF TO ProceedTo( HomeBase );
```

have the same significance.

However, if a method is being referenced from outside of the object, the method must be qualified by an appropriate reference variable.

```
TELL plane TO ProceedTo(HomeBase);
```

4.5 Formal Parameter Qualifiers: IN, OUT, INOUT

MODSIM, like Ada, makes a distinction between input and output parameters . Each parameter is declared as one of three possible variants:

IN: value may only be passed in to procedure from caller (pass by value)

INOUT: value may be passed in either direction (pass by reference)

OUT: value may only be passed out from procedure to caller (pass by reference)

Note: Constants and literals cannot be used as **OUT** or **INOUT** parameters for the same reason that they cannot be used on the left side of an assignment statement.

As an example of formal parameter qualifiers, a procedure which updates a counter might be declared in MODSIM as:

```
PROCEDURE Update(INOUT counter: INTEGER);
```

The direction specifier is used only in declarations, not in calls. Thus the call to procedure **Update** would be: **Update(TheValue)**.

These rules apply to both **METHOD** and **PROCEDURE** declarations. For example, a method of **AircraftObj** might be declared as:

```
ASK METHOD reportPosit(OUT posit : locationType;
                       OUT alt   : INTEGER);
```

Only **ASK** methods can include **INOUT** or **OUT** parameters. **TELL** methods cannot use **INOUT** or **OUT** parameters because the invoking code would proceed past the **TELL** statement before the invoked method was able to supply the return values.

5. Inheritance

A new object type can be defined in terms of an existing object type. The newly derived object type is then termed a derived type of the base type.

The derived type will normally include new fields and/or methods not present in the base type. It may also redefine (override) the implementation of a method defined in an underlying object type, after explicitly declaring the override.

A overridden method can invoke the method of the same name in an underlying or base type, by use of the **INHERITED** keyword.

Any method not overridden by the derived type is automatically inherited from the nearest underlying object type. Similarly, the derived type also inherits all fields of its underlying object types.

While a derived type can redefine inherited methods, it cannot redefine inherited fields. It can, however, add new fields of its own.

A reference value for an object can safely be stored in a reference variable of one of its underlying object types. This corresponds to redefining the reference variable from the specific to the more general object type. The converse assignment is not always safe, and thus requires an explicit action by the programmer to circumvent the strong type checking rules.

5.1 Declaring an Inheritance

A new type of object can be declared in terms of a previously-declared object type. This provides the new object type with the fields and methods of the earlier type, or, to use the idiom of object-oriented programming, the new type inherits the properties of the earlier type.

In fact, the simple object type declarations shown in earlier examples will only be used for the least interesting types of object. Most MODSIM object types are built upon the definition of other objects, either those from the standard MODSIM library, or user-defined object types.

For example, objects in a simulation often need to have the capability to contain a queue of other objects. We can define object type **VehicleObj** as inheriting the properties of **QueueObj** by the syntax:

```
VehicleObj = OBJECT(QueueObj)
  . . .
END OBJECT;
```

We say that **QueueObj** is a *base type* of **VehicleObj**. **VehicleObj** is termed a derived type of **QueueObj**. Any number of new object types can be derived from a single base type. A derived type can have more than one base type by means of multiple inheritance which is explained in Chapter 6.

New object types can also be defined in terms of other derived types. A hierarchy of o bject types occupying multiple layers is shown here.

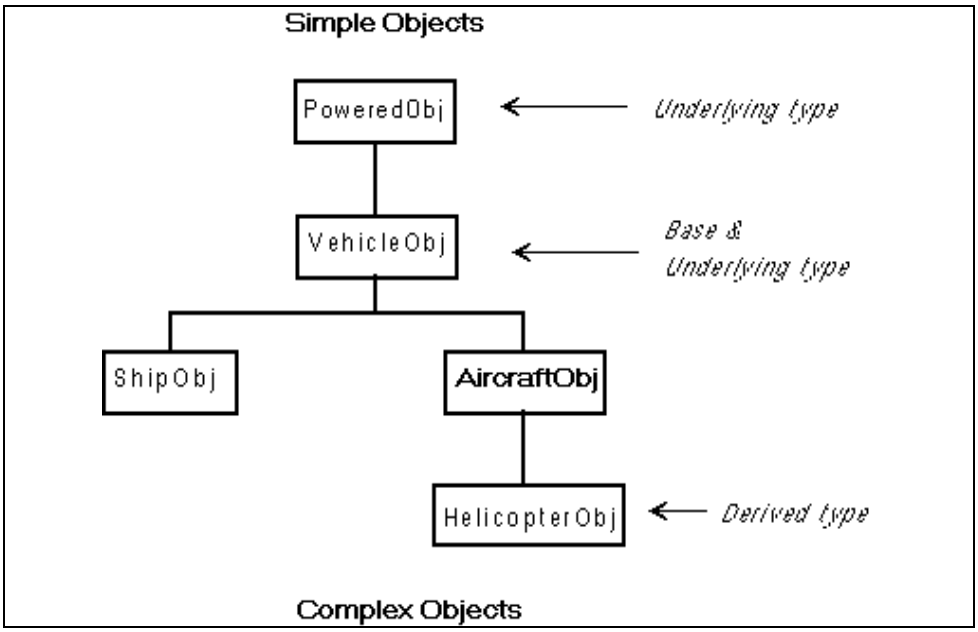


Figure 5-1. Sample Inheritance Tree

Any object type that is either a base type of **SomeObject**, or a type n levels below **SomeObject**, is referred to as an underlying type of **SomeObject**; and **SomeObject** includes all the properties of the underlying types.

In the illustration, **VehicleObj** is a base type of **AircraftObj,** as well as underlying type of **HelicopterObj** and **ShipObj**. **HelicopterObj** is a derived type of **AircraftObj**.

5.2 Extending Object Properties

A derived object type can, in addition to any properties inherited from the base type, d efine its own additional properties, as in the declaration:

```
HelicopterObj = OBJECT(AircraftObj)
  InHover : BOOLEAN;
  TELL METHOD Hover(IN Posit : locationType;
                    IN Alt   : INTEGER);
END OBJECT;
```

The derived object has access to all of the properties of its base type, in addition to its own unique properties, so we could implement **Hover** using properties of both **AircraftObj** and **HelicopterObj.**

```
OBJECT HelicopterObj;
  ...
  TELL METHOD Hover(IN Posit : locationType;
```

```
                    IN Alt   : INTEGER);
  BEGIN
    ProceedTo(Posit);
    Stop;
    altitude := Alt;
    InHover  := TRUE;
    ...
  END METHOD;
END OBJECT;
```

5.3 Overriding Methods

It often occurs in inheritance that a derived object type may wish to modify a method which it has inherited from an underlying object type.

For example, the **VehicleObj** type defines a rudimentary mechanism for moving an object between two locations in two dimensions. However, a more elaborate method would be needed for **AircraftObj**, which would have specific constraints for climbing, descending and turning in three dimensions.

Since **AircraftObj** already has inherited a standard **ProceedTo** method, it must explicitly *override* this method and define a new method. This is accomplished by noting the override in the object type declaration of the new object and supplying the replacement method in the object implementation block.

The declaration of **AircraftObj** would look like

```
AircraftObj = OBJECT(VehicleObj)
  OVERRIDE
    TELL METHOD ProceedTo(IN dest : locationType);
END OBJECT;
```

The object implementation blocks for **VehicleObj** and **AircraftObj** would respectively contain definitions for the original method and the method which **AircraftObj** is substituting for the overridden one

```
OBJECT VehicleObj;
  TELL METHOD ProceedTo(IN dest: locationType);
    ..original vehicle code here.

OBJECT AircraftObj;
  TELL METHOD ProceedTo(IN dest: locationType);
  ..replacement code for aircraft here.
```

The original, inherited, behavior was coded in the **ProceedTo** method for **VehicleObj,** while the **ProceedTo** method which describes the new behavior for **AircraftObj** is described in the object implementation block for **AircraftObj**.

5.4 Extending Inherited Behaviors

In some cases, the overriding method completely replaces the method from the underlying type. However, it is more typical that the original code will be invoked and new code added. We extend the underlying method. In these cases the new method can invoke the

overridden method, as appropriate, and then provide additional code which describes the modified behavior.

Invoking the inherited method which has been overridden is accomplished by preceding the method call with the **INHERITED** keyword.

For example, to implement the proper method **ProceedTo** for an **AircraftObj**, it may be easier to build upon existing **ProceedTo** code defined for its base type, **VehicleObj**. This would be done in its implementation module using statements such as:

```
OBJECT AircraftObj;
  TELL METHOD ProceedTo(IN dest: locationType);
  VAR
    deltaltitude: REAL;
  BEGIN
    deltaltitude := altitude - dest.z;
         { more flying-specific code here }
    INHERITED ProceedTo(dest);
  END METHOD;
END OBJECT;
```

Thus the **ProceedTo** method for an AircraftObj would perform some unique calculations, and then invoke the **ProceedTo** method from the underlying object type, in this case **VehicleObj**.

An inherited call can be performed for a function method as well. Operations to be performed "before" and "after" a particular method can be handled in MODSIM by the ordering of code before and after the inherited call. In general, each method that uses inherited code will take the form:

```
ASK METHOD thisMethod(args);
BEGIN
  { code prior to invoking original method }
  INHERITED thisMethod(args);
  { code after invoking original method }
END METHOD;
```

This mechanism is both simple and versatile, and is appropriate for all single inheritance combinations of methods. When an object inherits methods from more than one object type, a slightly different approach is used. This approach is described in Chapter 6.

5.5 Assignment Compatibility

A reference value for an object type can always be assigned to a reference variable of a base or underlying type. This is because all fields and methods of the base or underlying type are, by definition, known to the derived object. For example, using the previous relationships, we might have

```
VAR
  flyer : AircraftObj;
  helo  : HelicopterObj;     ⇦ derived from AircraftObj
  ...
```

```
BEGIN
  flyer := helo;
```

This relaxed assignment compatibility also applies to parameters of procedures and methods.

The relaxed compatibility works only in one direction since it is not always safe to assign a variable to a reference variable of a more specific object type. The compiler will not allow:

```
helo := flyer;
```

because the **HelicopterObj** may have defined new fields or methods which are unknown in the realm of **AircraftObj**s. An attempt to reference the fields or invoke the methods would be undefined. For instance, if we allowed the above assignment and then tried to **ASK helo TO Hover**, we would have a problem, since the object stored in the reference variable called **helo** is really an **AircraftObj** and it does not have a method called **Hover**.

6. Multiple Inheritance

MODSIM III allows an object type to be defined in terms of more than one base object type. This capability is called multiple inheritance.

When a new object type is defined in this way, it has a copy of each field and each method of its base types. Like many powerful features in any system, this can be a two-edged sword. If the base types from which the new object type has been derived have used the same names for any of their fields or methods, we are left with an ambiguous situation. MODSIM provides facilities to resolve some of these conflicts.

6.1 Declaring Multiple Base Types

A derived object type may be defined in terms of multiple base types. They are listed in the declaration, as in:

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

This declaration corresponds to the illustration in figure 6-1.

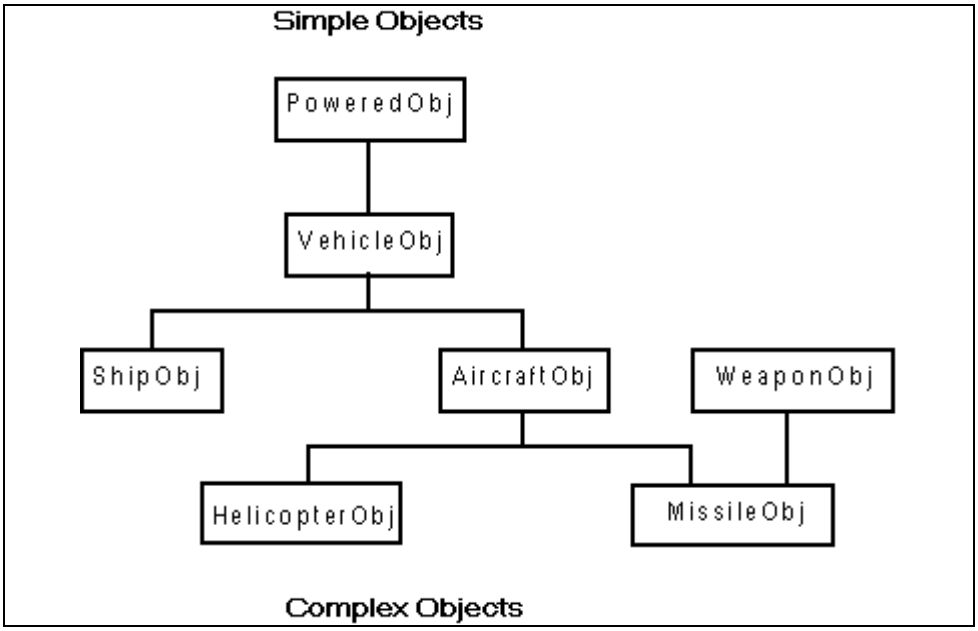


Figure 6-1. Multiple-path Inheritance

6.2 Conflicting Fields

If field identifiers of the same name exist in two or more of the base types, the derived object type will contain a field for each one. Obviously, any attempt to reference those fields in the derived object type would be ambiguous, particularly if some of the fields with

matching names were of differing types. Because of this, MODSIM does not allow refe r-ences of this sort and will flag them as a compile time error.

If a field from a base type must be accessed and some other base type has a field of the same name, extra code must be provided to disambiguate the field. This code can assign the reference value of the object to an object of the desired base type, and then unambi-guously access the desired field.

Consider the situation which would occur if the **AircraftObj** and the **WeaponObj** from which **MissileObj** was derived each had a weight field. And just to make things more difficult, the **WeaponObj**'s weight field is of type **REAL** and expressed in kilo-grams. The **AircraftObj**'s weight field is of type **INTEGER** and is expressed in pounds.

Assume we have three reference variables called **Aircraft**, **Weapon** and **Missile** to match their respective types. If we assign an instance of **MissileObj** to all three ref-erence variables, we have the following situations:

```
Missile  := ASK armory TO Issue(CruiseMissile);
Aircraft := Missile;
Weapon   := Missile;

n := ASK Missile weight   ⇐  illegal reference
n := ASK Aircraft weight  ⇐  n gets Aircraft's weight (an INTEGER)
x := ASK Weapon weight    ⇐  x gets Weapon's weight (a REAL)
```

6.3 Resolving Conflicting Methods

Cases where two or more of the base types have methods of the same name are permitted only when the method is derived from a common ancestor. If there is not a common an-cestor, the MODSIM compiler produces an error message.

The definition of the object that joins the ancestors must override the common method if any of the intervening ancestors overrides it. Otherwise, polymorphism will not be able to work for this method and the MODSIM compiler will produce an error message.

You can supply a completely new method implementation or, as with normal inheritance, the inherited method can be invoked as part of the implementation. When the method is inherited from multiple ancestors, a qualified form of the inherited method invocation can be used to specify the desired version of the method.

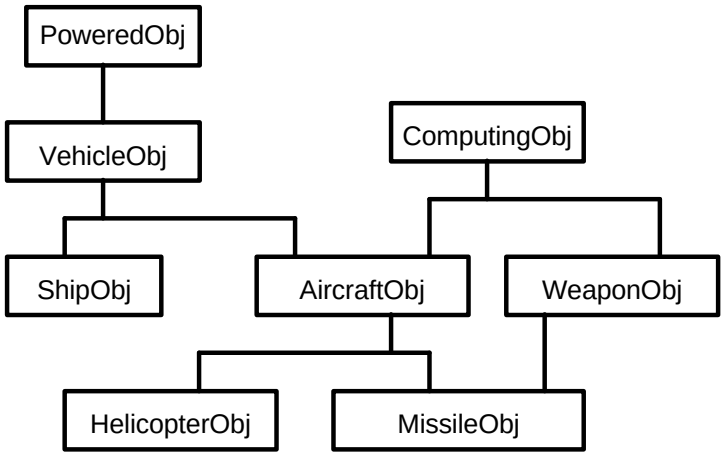


Figure 6-2. Common Ancestor

As an example, we can consider the **MissileObj** which was derived from an **AircraftObj** and a **WeaponObj**. Assume that each of the base types has a method called **FindTarget**. Observe that the **FindTarget** method is itself a method of some **ComputingObj** from which both **AircraftObj** and **WeaponObj** inherit:

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

The inheriting object must override the common method and provide its own:

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

If the common method is to be invoked in the implementation of the inheriting object, a qualified inherited call must be used. The qualified inherited call explicitly specifies the desired version of the method.

Continuing with the previous **MissileObj** as an example, the implementation might provide the following method:

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

A qualified inherited call requires the qualifier to be a base type of the object that is being defined. The **INHERITED FROM** syntax cannot be used to access methods of unrelated objects. In the example above this means that the inherited call for the **FindTarget** method can only be qualified by one of the two base types of **MissileObj**; either **WeaponObj** or **AircraftObj**. We could not inherit the **Dive** method from **SubmarineObj**, since we are not descended from it.

6.4 Combining Multiple Methods

In many cases, it may be necessary to combine the inherited methods from multiple paths in the derived type's method. This can be done as long as the **INHERITED** statement qualifies the inheritance to avoid ambiguity.

Elaborating on the previous example, we could do the following:

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

6.5 Conflicting Field and Method Names

If a method name from one base type is the same as a field name from another base type, MODSIM flags this as a compile-time error. There is no way to resolve this conflict except by renaming one of the fields. This is intentional.

No conflict resolution mechanism has been provided in this case since it would lead to code which, although it could be understood by the compiler, would be confusing or misleading to those responsible for code maintenance.

7. Data Hiding

MODSIM III separates the definition of methods from the details of their implementation. It also allows the separation of the definition of procedures from the details of their implementation. This provides for separate compilation and data hiding in large projects.

The basic compilation unit is the module. Objects, etc. are typically defined in a **DEFINITION MODULE**, and then implemented in an **IMPLEMENTATION MODULE**.

Some fields or methods will not be appropriate for use outside the object. Fields or methods declared as **PRIVATE** can be used only within the object itself, or within derived object types.

7.1 Definition Modules

Objects types are declared in a MODSIM definition module. More than one object type can be declared in the same module.

If an object type is imported from the definition module, all of its field and method identifiers are also imported.

The methods for an object type are defined in the corresponding implementation module. They are contained in their own object implementation block which is named after the object type.

For example, a subset of the object types from the previous section might be defined using the structure:

```
DEFINITION MODULE FlightModule;
FROM MovingModule IMPORT VehicleObj;

TYPE
  AircraftObj = OBJECT(VehicleObj)
    altitude : REAL;
    TELL METHOD ClimbTo(IN height: REAL);
    TELL METHOD Circle;
  END OBJECT;

  HelicopterObj = OBJECT(VehicleObj)
    ...
  END OBJECT;
END MODULE.

IMPLEMENTATION MODULE FlightModule;

  OBJECT AircraftObj;

    TELL METHOD ClimbTo(IN height : REAL);
    BEGIN
      ...
    END METHOD;
```

```
      TELL METHOD Circle;
      BEGIN
        ...
      END METHOD;

    END OBJECT;

    OBJECT HelicopterObj;
      ...
    END OBJECT;

  END MODULE.
```

7.2 Private Fields and Methods

In defining an object type, it is often necessary to include properties which are internal to the object's implementation. These component fields and/or methods should be unavai l-able to external users of the object type.

This can be accomplished using a **PRIVATE** section within the object type declaration, much as **VAR**, **CONST** and **TYPE** delimit sections of a module. In the declaration

```
TYPE
  AircraftObj = OBJECT(VehicleObj)
    altitude : REAL;
    TELL METHOD ClimbTo(IN height: REAL);
    TELL METHOD Circle;
    PRIVATE
      liftCoeff : REAL;
      ASK METHOD CalcLiftCoeff;
  END OBJECT;
```

we would refer to **liftCoeff** and **CalcLiftCoeff** as *private properties*. The private properties of an object are visible only within that object type or a derived object type.

Up to the **PRIVATE** declaration, all properties are assumed public. The **PRIVATE** section must precede the **OVERRIDE** section. Any method in the **OVERRIDE** section has the same scope as the original definition.

Section III. Simulating with MODSIM III

8. Object-Oriented Simulation

MODSIM III contains powerful and flexible tools to build discrete-event simulation models. Each MODSIM object is capable of carrying on multiple, concurrent activities each of which elapses simulation time. An **activity** is scheduled by an object instance using a **WAIT** statement in a **TELL** method. An **activity** is what occurs in the model as time elapses. An **event** is a point in time at which the state of the model changes in some way. Any or all activities of an object can be interrupted, if necessary.

Another way to view the relationship between events and activities is to say that MODSIM III is in a **WAIT** statement, not executing code, while an activity transpires. But, during an event, MODSIM is executing code...; typically the code which is changing the model's state in some way.

During an activity, time is elapsing and no MODSIM code is executing to cause this to happen. During an event, MODSIM code is executing to make the model's state change, but no simulation time is elapsing.

The **TELL** method is MODSIM's tool for writing models. It has several unique and powerful capabilities for modeling and simulation which standard object-oriented methods, such as the **ASK** method do not have:

- It can elapse simulation time using a **WAIT** statement
- It can execute synchronously or asynchronously with respect to simulation time. This means that it can operate concurrently, in a separate thread of execution.
- It can be scheduled to execute at some time in the future.

Not only can one object instance have multiple **TELL** methods carrying on activities simultaneously with respect to simulation time, but any one method of the object instance can be invoked multiple times. Each of these method invocations can be carrying on an activity at the same point in simulation time.

A method can perform a sequence of related actions. In a time-elapsing **TELL** method, these actions may be punctuated by intervals during which simulation time elapses, i.e. they perform a **WAIT**. When a **WAIT** statement is encountered, MODSIM saves the state of the time-elapsing method and then suspends its execution until the **WAIT** is completed or is interrupted. MODSIM then resumes execution of the time-elapsing method at the appropriate simulation time. During the **WAIT,** other activities may be taking place.

8.1 Simulation Time

MODSIM III performs discrete event simulation using several unique capabilities. The **WAIT** construct and the time-elapsing **TELL** method are built into the language itself. Other constructs used to implement simulation models are contained in standard modules such as **SimMod** (general simulation tools), **ResMod** (constrained resource modeling), **RandMod** (random sampling from statistical distributions) and **StatMod** (automated gathering and reporting of simulation statistics).

The units of time used during simulation are dimensionless. They can represent whatever time granularity is appropriate for the simulation - years, months, days, hours, minutes, seconds, milliseconds, nanoseconds. Time is expressed as a standard REAL number, e.g. a 64-bit floating point number.

Simulation time is maintained by MODSIM. It is available to the user through the **REAL**-valued function **SimTime()** which can be imported from **SimMod**.

8.2 Elapsing Simulation Time in TELL Methods

The term **activity** describes what happens in a time-elapsing **TELL** method when a **WAIT** statement is executed. In other words, an activity which elapses simulation time is occurring. A **TELL** method with three **WAIT** statements in its body can perform three activities. One object can have a number of methods, each of which can have zero or more activities / **WAIT** statements. No simulation time passes while the code in a **TELL** method is being executed. Simulation time only elapses during **WAIT**.

A **WAIT** statement is thus used to specify that simulation time should elapse at some point in a method while the model is engaged in an activity. It optionally specifies a sequence of statements to be executed after the **WAIT** is successfully completed, and an optional sequence of statements to be executed if the **WAIT** is interrupted before completing normally.

The structure of a **WAIT** statement is similar to that of an **IF** statement. It has the general form:

```
WAIT reason
  [ StatementSequence ]
[ ON INTERRUPT
  StatementSequence ]
END WAIT;
```

where **reason** is a keyword **DURATION**, which specifies how long to wait or the keyword **FOR** which indicates that the method should wait until the method mentioned in the **FOR** statement completes execution. The **WAIT..FOR** blocks until another activity is completed / an event happens.

The **ON INTERRUPT** clause is optional. If the **WAIT** is “successful”, the first statement sequence is executed. If the **WAIT** is “not successful”, the statement sequence after the **ON INTERRUPT** is executed instead. In either case, execution continues after the **END WAIT** unless one of the statement sequences executes a transfer of control, such as a **TERMINATE**.

Although the **ON INTERRUPT** clause is optional, if it is omitted and a **WAIT** is interrupted, a run-time error occurs.

A **WAIT** statement can only appear in a **TELL** method. If it is placed elsewhere, the compiler will flag the error at compile time.

The most basic **WAIT** is one for a specific period of time. A wait for a specified period of simulation time is achieved by the **WAIT DURATION** statement. The syntax of the state-

ment is:

```
WAIT DURATION timevalue
  [ Statement Sequence ]
[ ON INTERRUPT
  Statement Sequence ]
END WAIT;
```

where **timevalue** is an expression of type **REAL**.

8.3 Delayed Method Calls

MODSIM includes a straightforward extension to its basic framework to support object-oriented interactions within a simulation. This allows for simple event-oriented simulations to be constructed with ordinary objects, and, more significantly, provides a general mechanism which allows objects to interact while simulation time elapses.

An earlier chapter introduced the **ASK** and **TELL** methods. Although both "send a message" to the receiving object, the two statements differ in how they interact with simulation time.

In many cases, when an object is sent a message to invoke one of its methods, we want to know that the invoked method has completed before we perform the next step. For example, for an **AircraftObj** to land on a runway, it first must have one properly assigned to it, as in:

```
ASK controller TO AssignRunway(myrunway, assignOK);
IF assignOK
  destination := myrunway;
ELSE
  destination := alternateAirport;
END IF;
...
```

In this case, the simulation logic requires that the **AssignRunway** method for object **controller** be complete before the following **IF** statement is executed. The invocation of an **ASK METHOD** is comparable to an ordinary procedure call, i.e., the **AssignRunway** method is required to complete before the next statement is executed.

We say that the program blocks while waiting for the **AssignRunway** method to complete execution. Another way to describe this on a traditional sequential or single-processor architecture is to say that the invoking code relinquishes the processor to the routine being called, **AssignRunway** in this case, and then regains control and continues execution after the invoked method has completed execution. A single thread of execution results. Another way to describe this standard way of doing business is to say that the activities are synchronous.

While an activity simulated by a method elapses simulation time, it may not be necessary or appropriate for the invoker to pause while that method completes. The invoking code may wish to do other things while the invoked routine is running. It may want to send a message to the object, invoking one of its time-elapsing methods, and then continue, without waiting for the activity to complete.

This capability is provided by the **TELL** method. The invoking process executes the **TELL** statement and then continues on without waiting for the invoked time-elapsing method to complete (or even to start) execution. When a **TELL** method is invoked, the code which invokes it does not block. We then have two processes executing simultaneously with r e-spect to simulation time. The activities are asynchronous and there are multiple threads of execution with respect to simulation time.

The complete syntax of the **TELL** statement is:

```
TELL object [ TO ] method[(arguments)] [ IN delay ]
```

The **TELL** statement can appear anywhere in a program. It is used to invoke **TELL** methods, and may not be used to invoke **ASK** methods. **TELL** methods are proper methods with **IN** parameters only.

A **TELL** method cannot be a function method and cannot have **OUT** or **INOUT** parameters since there is no place to which this returned information can be passed. The invoking code has proceeded past the **TELL** statement without waiting for any return information.

To take an example, a dispatcher might want to start a truck enroute to a particular loc a-tion, using code such as:

```
TELL METHOD DispatchTruck(IN dest: Point);
VAR
  truck : VehicleObject;
BEGIN
  ...
  TELL truck TO ProceedTo(dest);
  ...
END METHOD;
```

In this case, the **DispatchTruck** method would complete execution at the same simul a-tion time at which it began, no matter how long it eventually took the truck to **Pro-ceedTo** the destination. Another way to describe the behavior is to say that the **DispatchTruck** method schedules the **ProceedTo** method but does not relinquish control of the processor so that the **ProceedTo** method can execute. The **ProceedTo** method in this example will actually execute at some point later in real time, but at the same point in simulation time, when the **DispatchTruck** method eventually completes or performs a **WAIT**. In either of those two conditions, the **DispatchTruck** method would relinquish the processor and the **ProceedTo** method would have a chance to start executing.

Also note that, even though this is a **TELL METHOD**, it performs no **WAIT**s. TELL meth-ods need not perform a **WAIT**. Since it is a **TELL** method, however, this means that it can be scheduled to execute at some time in the future as in:

```
TELL truck TO ProceedTo(dest) IN 20.0;
```

9. Object Interaction

Operations involving objects may be combined into more complex arrangements than described in the previous chapter.

Each object may perform several concurrent activities with respect to simulation time. Time-elapsing methods may synchronize by awaiting the completion of another method. Or a method may suspend execution and await a continuation signal from a trigger object.

An object instance may have several methods which are concurrently carrying out activities with respect to simulation time.

Any single time elapsing **TELL** method of an object instance may be invoked multiple times such that there are multiple instances of one method carrying out activities concurrently with respect to simulation time. Each method instance has its own distinct, unshared set of local variables.

9.1 Concurrency in MODSIM III

At this point it is worth reviewing how MODSIM supports concurrency. Note that when we speak of concurrency we mean that a number of activities can be happening simultaneously with respect to simulation time. On traditional sequential computer architectures, each method which must execute at the same instant in simulation time actually takes turns (in real time) using the one processor. On parallel computer architectures however, activities could actually take place concurrently with respect to real time.

For the purposes of this discussion we will consider sequential architectures where concurrency is defined to be with respect to simulation time only.

- Two distinct object instances can each be carrying out activities at the same point of simulation time.
- One object instance can have two different TELL methods carrying out activities at the same point of simulation time.
- One particular TELL method of one object instance can be invoked multiple times so that distinct instances of that one method are each carrying out activities at the same point of simulation time.

9.2 Synchronized Activities

In some scenarios, two methods must operate synchronously. One method starts a second method and then waits over a period of simulation time for the second method to complete before the first one resumes execution.

To accomplish this, MODSIM provides the **WAIT FOR** statement:

```
WAIT FOR object [ TO ] Time-elapsing method[ (arg) ]
  [ Statement Sequence ]
[ ON INTERRUPT
  Statement Sequence ]
END WAIT;
```

The effect of this statement is to:

```
TELL object TO Time-elapsing method();
```

and then wait for the method to complete. Once the invoked method completes, the statement sequence after the **WAIT FOR** is executed.

If the invoking method is interrupted while still waiting for the invoked method to complete, the statement sequence after the **ON INTERRUPT** is executed.

If the invoked method **TERMINATE**s, i.e. prematurely finishes execution, then the invoking method which was waiting for execution of the invoked method to complete also **TERMINATE**s.

In the other forms of the **WAIT** statement two conditions can occur:

- The **WAIT** completes normally
- The **WAIT** is interrupted before it is finished

In the **WAIT FOR** statement a third condition is possible:

- The routine invoked by the WAIT FOR terminates, and the method which contains the WAIT FOR also terminates.

The effect of **TERMINATE** is recursive. If method 1 does a **WAIT FOR** method 2 which does a **WAIT FOR** method 3, and method 3 then **TERMINATE**s, then method 2 **TERMINATE**s and this causes method 1 to **TERMINATE**.

As with other forms of the **WAIT**, the **WAIT FOR** can only be used in **TELL** methods of an object.

The time-elapsing **TELL** method being waited for can belong to any object; this includes another method of **SELF**.

To illustrate use of the **WAIT FOR**, suppose a simulation includes a transportation capability. The shipping process for some freight might include a method which waits while an **AircraftObj** flies the freight to its desired destination:

```
TELL METHOD Ship(IN dest : Point);
VAR
  ourtransport : TransportObject;
BEGIN
  ourtransport := TransportManager.nextTransport;
  WAIT FOR ourtransport TO FlyTo(dest)
    TELL Operations MyStatusIs(Arrived);
  ON INTERRUPT
    TELL Operations MyStatusIs(Delayed);
  END WAIT;
END METHOD;
```

When the **WAIT FOR** statement is encountered, **ourtransport** is asked to execute its **FlyTo** method. The **Ship** method waits for the **FlyTo** method to complete before it proceeds to its next statement.

9.3 Arbitrary Synchronization with Trigger Objects

Some processes will need to wait until a specified condition occurs. For these situations, MODSIM provides a special object type, **`TriggerObj`**, which, along with the **`WAIT FOR`** statement, allows a method to pause and wait until some condition occurs.

The syntax of the statement is:

```
WAIT FOR trigger object [ TO ] Fire
  [ Statement Sequence ]
[ ON INTERRUPT
  Statement Sequence ]
END WAIT;
```

When the **`WAIT FOR ... Fire`** statement is encountered, the method suspends and waits until the trigger object's **`Trigger`** method is invoked by some other method. At that time, the statement sequence after the **`WAIT FOR ... Fire`** is executed. If the trigger object's **`InterruptTrigger`** method is invoked, the suspended method's statement sequence after the **`ON INTERRUPT`** is executed instead. A trigger object can have any number of methods waiting for it to **`Trigger`** or **`InterruptTrigger`**.

Taking the example of an **`AircraftObj`**, a refueling method might prudently wait until the plane is on the ground before requesting that the tanks be “topped off”, as in:

```
TYPE
  landedSignal: TriggerObj;
...
  IF flying
    WAIT FOR landedSignal TO Fire      ⇦ i.e. wait until some other
    END WAIT;                            method releases
  END IF;                                trigger landingSignal

  ASK airport TO assignRefueler(tankTruck);

  WAIT FOR tankTruck TO refuel(SELF, fuelCapacity);
  END WAIT;
```

9.4 Multiple Process Activities

To construct realistic simulation models, it is often necessary to model a physical object which can perform several operations simultaneously. An aircraft in an airport model, for instance, may be required to perform movement, communications and collision avoidance activities simultaneously. Although this is a fairly common situation, it has traditionally been difficult to model, particularly when the activities may interact.

To support such models, MODSIM allows an object to do more than one thing at once. For instance, an object can receive multiple messages and handle those messages simultaneously, even when some actions will require time-elapsing sequences of activities.

MODSIM code can be written so that the messages are handled in a way that resolves potentially contradictory states. For example, an object may be in the middle of one op-

eration when it receives a message to perform a different, conflicting operation. In response, the object can:

- Interrupt the conflicting time-elapsing method which is waiting
- Ignore the new request
- Defer the new request.

An operation which an object is to perform over a span of simulation time is termed an **activity** and is handled by a **WAIT** statement in one of the methods of the object.

Several simple activities could be coded in one time-elapsing method which has several **WAIT**s or a complicated behavior could be composed of several different methods.

9.5 Interrupting Activities

MODSIM has provisions for interrupting and stopping any or all activities prematurely. Any time-elapsing method which is **WAIT**ing can be interrupted. This is done by invoking the **Interrupt** procedure which can be imported from **SimMod**. For example:

```
Interrupt(AircraftObj, "ProceedTo");
```

The procedure takes two arguments; the object whose activity is to be interrupted and the particular method of that object instance which is to be interrupted.

Interrupting the currently executing activity has no effect. Interrupting an activity that is waiting will cause it to execute the **ON INTERRUPT** clause of the **WAIT** statement. If there is no **ON INTERRUPT** clause, a run time error will occur.

If it is necessary to interrupt all **WAIT**ing activities of an object instance, the **InterruptAll** procedure of **SimMod** can be used:

```
InterruptAll(AircraftObj);
```

9.6 How Objects and Their Activities Interact

In MODSIM, every object maintains an **ActivityList** which is a sorted group of activities. The activities are ranked by **timeNext**, the time each activity is scheduled to finish its **WAIT**.

An activity record is placed on an object instance's activity list each time one of the object instance's time-elapsing methods executes a **WAIT**. The activity record contains all of the information needed to resume execution of a time-elapsing method after its **WAIT** is complete or has been interrupted.

When the **Interrupt** procedure is invoked, it scans the particular object's activity list and interrupts the most imminent activity which matches the given name. If there are no matches, nothing happens. We could do the following:

```
Interrupt(AircraftObj, "flyTo");
```

and the **flyTo** method's **WAIT** would be interrupted. If a method contains multiple **WAIT** statements, then whichever one is currently waiting is interrupted. If it is important to the user to conditionally control which **WAIT**s are interrupted, then the method can be broken into separate methods for each activity, or a status can be set before each wait, and then checked by the interrupting code. For more precision control, MODSIM allows the pr o-grammer to get a "handle" to the particular wait of a particular method invocation and specify the interrupt by using this handle.

The **TERMINATE** statement is used by any time-elapsing method which wants to finish execution prematurely. It not only stops execution of the current method, but also **TER-MINATE**s the method which invoked it using a **WAIT FOR**. The effect of the **TERMINATE** is recursive. In other words, the invoking routine becomes **TERMINATE**ed and, therefore, **TERMINATE**s the method which invoked it. Like the **WAIT** statement, the **TERMINATE** statement may only appear within a **TELL** method.

To summarize:

- The **Interrupt** procedure is used from outside an object's time-elapsing method to "wake up" the method before it completes the **WAIT**. The interrupted method resumes execution by performing the statement sequence after **ON INTERRUPT**.
- The **TERMINATE** statement is used from inside an object's **TELL METHOD** to pre-maturely stop execution of the method and the method which called this method if it used the **WAIT FOR** construct.

10. Grouping Objects

Multiple objects in MODSIM are associated through groups. Objects may be selectively added or removed from a group, and a MODSIM program can iterate through the members of a group. A group can hold any type of object.

Each MODSIM III group has a variant which is capable of automatically gathering detailed statistics on the operation of the group.

10.1 Associating Objects

When making extensive use of dynamic data structures, such as objects, a language needs a way to associate multiple objects for common manipulation.

This is especially true for simulations, which typically group objects queuing for a resource (the proverbial bank teller or barber) or a series of events scheduled to happen at a specific time. Such associations are referred to as *groups* in MODSIM.

10.2 Groups

A group may contain zero or more of any type of object. An object can belong to any number of groups. All of MODSIM's groups support prototyping. This means that the user can derive a new group object from one of MODSIM III's built-in groups and specify the type of object that the group is meant to hold.

Probably the most commonly used group is **`QueueObj`**, which is a First-In-First-Out (FIFO) group. Objects are added to the back and removed from the front.

The **`StackObj`** is a Last-In-First-Out (LIFO) group. Objects are added and removed from the front.

The **`RankedObj`** is ranked according to the value of an object's field or fields. The user specifies the ranking by overriding the RankedObj's **`Rank`** method and providing code to compare two objects for sorting purposes. The object uses this Rank method to sort each added object into its correct position in the ranked group.

For more efficient handling of large ordered groups which will be added and removed randomly, MODSIM III supports the **`BTreeObj`** which orders objects according to the value of a string which is used as a key.

10.3 The Queue Group

For the **`QueueObj`** type, the following methods are defined:

```
ASK METHOD Includes(IN candidate: ANYOBJ) : BOOLEAN;
ASK METHOD Add(IN NewMember: ANYOBJ); { behind Last }
ASK METHOD Remove()   {  removes First  }   : ANYOBJ;
ASK METHOD First()                          : ANYOBJ;
ASK METHOD Last()                           : ANYOBJ;
        {  First ... candidate ...  Last
                <- Prev | Next ->           }
ASK METHOD Next(IN candidate: ANYOBJ)    : ANYOBJ;
```

```
ASK METHOD Prev(IN candidate: ANYOBJ)    : ANYOBJ;
ASK METHOD RemoveThis(IN member: ANYOBJ);
ASK METHOD AddBefore(IN ExistingMember,
                        NewMember: ANYOBJ);
ASK METHOD AddAfter(IN ExistingMember,
                       NewMember: ANYOBJ);
```

QueueObj also has defined the field **numberIn** which can be queried to determine the number of objects in a group.

The **Add** method places an object at the back end of the group while **Remove** takes it from the front of the group. **First**, **Last**, **Next** and **Prev** return reference values for those objects without changing the composition of the group. **RemoveThis** removes the specified object from a group. **AddBefore** and **AddAfter** add an object next to the specified object in the group.

The **Includes** method determines whether a specific object is part of a particular group without traversing the group. This is an important efficiency consideration. Each object in MODSIM III keeps an internal list of groups to which it belongs. The **Includes** method interrogates this list, which is likely to be shorter than most groups, to determine its answer.

10.4 The Stack Group

The **StackObj** type inherits all of the fields and methods of the **QueueObj**. It overrides the **QueueObj**'s **Add** method and substitutes an **Add** method which places objects at the front of the group instead of the back.

10.5 The Ranked Group

The **RankedObj** type inherits all of the fields and methods of the **QueueObj**. It overrides the **QueueObj**'s **Add** method and substitutes an **Add** method which inserts new objects into the group using a**Rank** method to determine their proper position.

```
ASK METHOD Rank(IN a, b: ANYOBJ) : INTEGER;
```

The user overrides the default **Rank** method and substitutes one which returns the following values:

```
-1  if  a < b
 0  if  a = b
 1  if  a > b
```

The user specifies how the comparisons, e.g. **a > b**, are made.

Since the **IN** parameters to method **Rank** are of type **ANYOBJ**, the user will need to assign them to variables of the appropriate type before attempting comparison of any fields. As an example, the following implementation for method **Rank** could be used to rank a group of cargo objects according to their weight field:

```
ASK METHOD Rank(IN a, b: ANYOBJ) : INTEGER;
VAR
  BoxA, BoxB:  CargoObj;
```

```
BEGIN
  BoxA := a;  BoxB := b;
  IF ASK BoxA weight < ASK BoxB weight
    RETURN -1;
  END IF;
  IF ASK BoxA weight > ASK BoxB weight
    RETURN 1;
  END IF;
  RETURN 0;
END METHOD;
```

10.6 The BTree Group

The **BTreeObj** is an implementation of the group objects which uses the more efficient Btree (balanced tree) data structure and algorithms to maintain a group which is ordered according to the value of a string supplied by the user.

This is the group object which should be used in situations where large ordered groups of objects need to be maintained. The Btree will be more efficient for larger groups and the Ranked group will be more efficient for smaller groups.

10.7 Iterating Through a Group

It is often necessary to iterate through a group to perform some action on selected members of the group. The standard methods provided for group objects allow the user to easily accomplish this by writing code, but MODSIM has a built-in construct to make the job simpler. The **FOREACH** construct iterates through each member of a group and makes each member available for inspection or removal.

```
FOREACH object IN group
   do something
END FOREACH
```

This is a robust construct which, aside from the convenience it offers, has a special capability. It is impervious to the manipulations of the current object which take place within the loop. If the logic inside the loop deletes the current object, this construct will still find the next object. If a new object is added after the current object, the **FOREACH** will still go to the "original" next object on the next iteration instead of the one which was just added.

To illustrate the **FOREACH**, we can go through a group of vehicles in a **VehicleGroup** and schedule any vehicle which is loaded with less than a 60% fuel load to be refueled. The group has truck objects, car objects, aircraft objects and boat objects. Each is derived from **VehicleObj** which defines the field **fuelState**.

```
VAR
  vehicle:   VehicleObj;
    .
  FOREACH vehicle IN VehicleGroup
    IF vehicle.fuelState <  0.6
      TELL vehicle TO Refuel;
    END IF;
```

```
    END FOREACH;
       .
```

11. A Simple Airport Model

This chapter examines a simple model of an airport. The model illustrates a number of the basic simulation constructs built into MODSIM III, but does not use any of the more advanced constructs which have not been covered in this text. These include items such as the **`Resource Object`**, **`Statistics Object`** and monitored variables.

This version of the model is meant to show a model in its most elemental form. A more elaborate version of this same model will be described after the chapters on SIMGRAPHICS II. That version uses animated graphics and other more advanced features of MODSIM III.

The source code for both versions of the model is included in the distribution media for MODSIM III, so you can experiment with this model and try changes to it.

11.1 Why Model an Airport?

Airports and their operating rules are familiar to most people. In the case of a simple airport (such as this one!) there is one runway which is used for both arrivals and departures. Arriving aircraft have priority over departing aircraft because their fuel capacity does not permit them to wait for long periods while flying circles around the airport. Departing aircraft, which wait on the ground, are less constrained.

An airport is conceptually easy to model because it has clearly stated operating rules and a limited set of behaviors to be modeled. It can be an interesting system to model because landing and departing aircraft are both competing for one limited resource, the runway. The interaction of the landing and departing aircraft objects and the controller object provides ample opportunities to illustrate object oriented programming.

Finally, it is a classic discrete-event simulation model characterized by randomly occurring events and queues which grow as demand on the resource exceeds capacity.

11.2 The Source Code

The source code for the model is presented here in its entirety. The code and its embedded comments are shown in Courier font. The code is interspersed with explanatory remarks using the Times Roman font. The model is fairly short and uncomplicated, so it is written in one module.

Initially, the most important thing to do is to read the rules by which the airport operates and the goal of this model. These are all stated at the start of the source code as a long comment.

```
MAIN MODULE airprt;

{ 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 ap-
      proach      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 ap-
      proach or is placed in the arrival queue ahead of ar-
      riving 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.
```

Now we have a description of the airport's operating rules, but we do not know why we are writing a model of the airport. What is it that we hope to learn by running the model? The answer to this question will determine how we design the model and how we conduct experiments with it. It will also determine the level of detail we will include.

```
Goals for the model:

   1.  Run the model with various traffic rates

   2.  Measure the following parameters which will be impor-
       tant 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.
```

Now we know the goal for the model. We will use it to determine the maximum rate at which aircraft can arrive and depart without causing arrival and departure queues which are "too long".

The next task is to design the model. We start by thinking about the objects involved in the model and how they will interact. In an object-oriented model this step is, literally, the top level of the model design. All we need to do is write the object type definitions and then flesh out the objects' methods with code to describe their behaviors

```
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 air-
          craft.
      d.  Receive notification when arriving and departing
          aircraft have cleared the runway.
      e.  Receive progress reports from aircraft making ap-
          proaches.

    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.
```

What is this traffic generator object? It wasn't a literal part of the airport description, o p-erating rules or model goals, however there is an implicit requirement to generate traffic for the airport. Nearly every model needs a mechanism to generate arriving objects such as aircraft, customers, phone calls, etc.

Next we state some of the assumptions used in the model. A very important item is a statement about the units used to measure time and distance in the model.

```
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     }

FROM GrpMod  IMPORT StatQueueObj;
FROM SimMod  IMPORT StartSimulation, SimTime;
FROM RandMod IMPORT RandomObj;
FROM IOMod   IMPORT ReadKey;

TYPE
  trafficType  = ( arrive, depart );
  statusType   = ( clear, inUse );
  priorityType = ( normal, goAround );

  AircraftObj = OBJECT; { virtual aircraft type with common
                          attributes }
    ovhdTime  : REAL; { overhead time: taxi onto runway for
TO
                        or roll out after landing }
```

```
    taskTime  : REAL; { time required to takeoff or fly ap-
                        proach }
  startTime : REAL; { sim time at which AC starts Land or TO }
  END OBJECT;

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

  LandObj = OBJECT(AircraftObj);
    landPriority : priorityType; { normal or goAround which is higher }
    TELL METHOD Land;
    TELL METHOD GoAround;
    ASK  METHOD ObjInit; { set landing performance attributes }
    ASK  METHOD ObjTerminate; { report statistics before DISPOSing
}
  END OBJECT;

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

  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;

VAR
  runway         : statusType;
  approachPath   : statusType;  { approach corridor }
  ArriveGen      : TrafficGenObj;
  DepartGen      : TrafficGenObj;
  Controller     : ControllerObj;
  randSeed       : INTEGER; { each new generator uses a new seed }
  trafficRanGen  : RandomObj; { used by aircraft to set their fields }
  goAroundCount  : INTEGER;
  interRate      : REAL;        { interarrival rate }
  ch             : CHAR;

CONST
  stopTime = 1440.0;    { minutes }
```

```
    sequenceDelay = 1.0;  { interval between departing AC }

    OBJECT TakeoffObj;
      TELL METHOD Takeoff;
      BEGIN
        WAIT DURATION ovhdTime + taskTime { taxi into position & takeoff
}
        END WAIT;
        TELL Controller ClearOfRunway;
        DISPOSE(SELF);
      END METHOD;
```

Note that once the aircraft has completed its take off or landing, it is no longer needed in the model, so it is discarded by having it DISPOSE of itself

```
      ASK  METHOD ObjInit; { takeoff }
      BEGIN
        ovhdTime  := trafficRanGen.Exponential(0.9);
        taskTime  := trafficRanGen.UniformReal(0.5, 0.9);
        startTime := SimTime();
      END METHOD;
```

In this simple model, the operating parameters are “hard wired” into the aircraft. It would be more realistic to allow the user to change these parameters at runtime to facilitate e x-perimentation. This is done in the graphical version of the model. The only parameter in this model which can be changed at run time is the interarrival rate of the aircraft.

Note that each new aircraft samples its operating parameters from one random number generator, **`trafficRanGen`**. If each aircraft created its own random number generator it would be necessary to seed each one with a unique seed and the random number generator object would only be used a few times in the **`ObjInit`** method. It would then be discarded. It is simpler and more efficient to set up one random number generator to be used by all aircraft.

```
    ASK METHOD ObjTerminate;
    BEGIN
      ASK DepartGen LogCompletion(startTime);
    END METHOD;
  END OBJECT;

  OBJECT LandObj;
    TELL METHOD Land;
    BEGIN
      WAIT DURATION taskTime
      END WAIT;
      TELL Controller ClearOfApproach;
      IF (runway <> clear)  { is runway clear? }
        GoAround;
        RETURN;    landing has been aborted, so exit this method
      END IF;
      runway := inUse;
      WAIT DURATION ovhdTime  {  roll out time }
      END WAIT;
```

```
      TELL Controller ClearOfRunway;
      DISPOSE(SELF);
    END METHOD;

    TELL METHOD GoAround;
    BEGIN
      INC(goAroundCount);
      WAIT DURATION 5.0
      END WAIT;
      landPriority := goAround;
      ASK Controller LandingClearance(SELF);
    END METHOD;

    ASK  METHOD ObjInit;  { land }
    BEGIN
      taskTime  := trafficRanGen.UniformReal(2.8, 3.0);
      ovhdTime  := trafficRanGen.UniformReal(0.8, 1.2);
      landPriority := normal;
      startTime := SimTime();
    END METHOD;

    ASK METHOD ObjTerminate;
    BEGIN
      ASK ArriveGen LogCompletion(startTime);
    END METHOD;
  END OBJECT   { AircraftObj };
```

The traffic generator object creates either landing or departing aircraft. It's **GenTraffic** method runs continuously in a loop until simulation time exceeds the stop time. It uses a random number generator which it creates at the time it is initialized.

```
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;

    ASK METHOD LogCompletion(IN whenStarted: REAL);
```

```
    BEGIN
      totTimeSpent := totTimeSpent + (SimTime() - whenStarted);
      INC(numACcomp);
    END METHOD;
```

The **ObjInit** method creates a random number generator object which is used to provide pseudo random interarrival times. Since there could be multiple instances of this object (two are used in this program), each instance needs to use a different seed for the random number generator object. If this was not done, every instance would generate planes at the same exact times.

```
   ASK METHOD ObjInit;
    BEGIN
      NEW(ranGen);
      INC(randSeed);  { each generator gets a unique seed }
      ASK ranGen TO SetSeed(randSeed);
    END METHOD;
  END OBJECT  { TrafficGenObj };

  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 control-
         ler 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 };

  PROCEDURE ShowResults;
  BEGIN
    OUTPUT;
    OUTPUT("Simulation Run is Finished at SimTime ", SimTime());
    OUTPUT("Mean interarrival rate used for arrivals & depar-
           tures: ", interRate);
    OUTPUT("# of arriving AC: ", ASK ArriveGen numACgen,
           "    # of departing AC: ", ASK DepartGen numACgen);
    OUTPUT("# of go arounds = ", goAroundCount);
    OUTPUT("Max # AC waiting to takeoff & Mean # waiting to take
           off: ", Controller.departQ.Maximum, " - ", Control-
           ler.departQ.Mean);
    OUTPUT("Mean time spent departing: ",
              ASK DepartGen totTimeSpent / FLOAT(ASK DepartGen
              numACcomp));
    OUTPUT("Max # AC waiting to land & Mean # waiting to land:
",
```

```
            Controller.arriveQ.Maximum, " - ", Control-
            ler.arriveQ.Mean);
    OUTPUT("Mean time spent landing: ",
              ASK ArriveGen totTimeSpent / FLOAT(ASK ArriveGen numAC-
              comp));
  END PROCEDURE;

BEGIN
  OUTPUT("Mean time between arrivals / departures?");
  OUTPUT("  (optimum value is around 6 minutes)");
  INPUT(interRate);
  NEW(Controller);
  NEW(trafficRanGen);
  INC(randSeed);
  ASK trafficRanGen TO SetSeed(randSeed);
  NEW(ArriveGen);
  TELL ArriveGen TO GenTraffic(interRate, arrive);
  NEW(DepartGen);
  TELL DepartGen TO GenTraffic(interRate, depart);
  StartSimulation;
  ShowResults;
  OUTPUT;
  OUTPUT(".... Hit any key to terminate");
  ch := ReadKey;  { on PC Windows holds window open till key is
                    hit }
END MODULE.
```

11.3 Results of the Model

These are the statistics which result when the model is run with the recommended 6 mi n-ute inter-arrival time:

```
Simulation Run is Finished at SimTime 1466.567241
Mean interarrival rate used for arrivals & departures:
6.000000
# of arriving AC: 243    # of departing AC: 230
# of go arounds = 4
Max # AC waiting to takeoff & Mean # waiting to takeoff: 25 - 10.467890
Mean time spent departing: 60.910749
Max # AC waiting to land & Mean # waiting to land: 6 - 1.344961
Mean time spent landing: 6.098950
```

Note: The model did not finish exactly at the end of 1,440 minutes of simulated time. That was the time at which the model stopped generating new aircraft traffic. There were still aircraft landing or departing at that time, so the model did not actually finish until 26 minutes later.

We can audit some of the model's results by simple consistency checks. This helps to validate that the model is operating correctly and builds confidence in the results which we can not easily compute.

The model was run with the recommended inter-arrival rate of 6 minutes. This means that an average of 10 aircraft per hour would arrive and the same average number depart. Since the model generated traffic for 1,440 minutes (24 hours), we would expect 24 x 10

or 240 aircraft to arrive and depart. We had 243 arrivals and 230 departures. For a short run with a small sample size this is reasonable.

Had we run the model for a longer time or over a number of replications, the number of observed arrivals and departures would come closer to the anticipated number. In fact, most of the model's statistics would be of better quality. The one exception to this rule is the maximum queue lengths. The longer we run the model, the more likely we are to encounter some random combination of events which would lead to a long queue building up. Since this would be a rare event and not characteristic of the system's "normal" behavior it is typically not a useful measure.

The solution to this problem is to run the model a number of times and average the maximum queue lengths. We say we have run X number of replications of the model. Thus, instead of characterizing the maximum queue length as the "longest queue in a month's worth of operation" it might be more useful to offer "the average daily maximum queue length resulting from a month's worth of operations".

11.4 Dissection of the Simple Model

With the introductory comments removed, the source code above contains about 340 lines of code. It is a complete model of a plausible system which allows the user to conduct experiments and arrive at some conclusion about the potential traffic capacity of the airport.

Is it a realistic model? Well, it is a realistic model of the airport which was described. However, that description was kept deliberately simple so the model would be easy to study.

In "real life" the airport and its model would be more complex. Here are some potential shortcomings of this model:

- It does not consider the possible effects of adverse weather on traffic flow.
- Although traffic is generated randomly, it is always at the same average rate. There are no peaks and valleys as the day progresses.
- It presents its data in aggregate form as a series of averages and maximums. It would be useful to know more about the nature of operations. Animation would help, as would some information about the variance of the performance data.
- In the real world, controllers would likely "break" the rules and take shortcuts to improve efficiency, if safety could be assured. The model's operating rules and logic would have to be made more complex to reflect this.
- No allowance is made for the startup transient. Most models would be allowed to "warm up" to a steady state, before starting to collecting statistics.
- There is a bug in the model! When aircraft are forced to go around because the runway is not clear, they take 5 minutes to go around and then are placed ahead of all other traffic in the landing queue. Unfortunately, there may be other aircraft

which had to go around already in the queue. The latest aircraft to go around would be placed in the queue ahead of other aircraft which had gone around. So the logic in the Controller object's **`LandingClearance`** method needs to be changed so that aircraft on a go around are placed in the landing queue ahead of "normal" aircraft, but behind all other "go around" aircraft.

There are obvious improvements which could be made, but this model has served its pu r-pose which is to illustrate how a collection of interacting objects can be organized into a model of a "real world" system. The program also illustrates how MODSIM III's simula-tion engine is unobtrusively embedded into the language. The WAIT DURATION stat e-ments in the TELL METHODS are the only visible evidence of the simulation capability. Note also that statistics on the models queues are automatically gathered behind the scenes so the code is not cluttered with statistics statements which might obscure the logic of the model.

Later we will expand this model to include animated graphics, more extensive statistics gathering, presentation graphics and a graphical user interface. However, the fundamental portions of that model will remain very similar to this code.

Section Ⅳ. Animated Graphics - SIMGRAPHICS Ⅱ

12. SIMGRAPHICS II

12.1 Background

SIMGRAPHICS II is a set of pre-defined objects used by MODSIM III programmers to build portable programs which use the graphical user interface, graphics and animation features of window systems such as Microsoft Windows and X Windows. It supports:

- Data visualization, charting and plotting.
- Animation and interactive graphics.
- User interface through dialog boxes, menus, palettes, etc.
- Editing of graphical and user interface objects.
- Three dimensional animation.

The two features which distinguish SIMGRAPHICS II from other graphics systems are its portability and its close integration with MODSIM III's simulation engine. This means that the implementation of animated graphics is simple and the same code can be moved from machine type to machine type without change. When MODSIM III programs which use SIMGRAPHICS II are moved from one window system to another, the appearance of the menus, dialog boxes, etc. change to conform with the "look and feel" of the particular window system, but the functionality remains the same.

12.2 What is SIMGRAPHICS II?

By default, SIMGRAPHICS II uses vector graphics to draw icons, charts and backgrounds, but it can also display and scale bit-mapped graphics. This means that photographs and other bit-mapped graphics can be displayed as part of a simulation. Conversion programs to translate between most bit-mapped and vector graphics standards are provided with the Unix versions of MODSIM III.

Vector graphics which use popular standards such as Autocad can be imported and used. Finally, SIMGRAPHICS II produces Postscript output files which can be used in documentation and reports.

This section of the tutorial:

- Introduces the user to the principal components of SIMGRAPHICS II
- Provides example code which shows how to implement each major graphics capability. The sections of code used for illustration are from sample programs. Each of these programs is included in the MODSIM III distribution.

12.3 SIMGRAPHICS II Object Types

SIMGRAPHICS II is implemented using MODSIM III object types. This gives it a number of advanced features which are usually not available in most graphics systems

- The objects are adaptable through multiple inheritance. The user can take any graphical object and add additional behavior to it through inheritance.
- Existing objects can be turned into graphical objects by simply inheriting from a graphical object.
- The graphics system is portable. SIMGRAPHICS II is an integral part of MODSIM III's library and is always the same on every system. There are no parts of the system which are machine or operating system dependent, so MODSIM code which uses SIMGRAPHICS II can be moved from one machine / operating system type to another by simply re-compiling it.
- The graphics are integrated with simulation. The animated objects in SIMGRAPHICS II have their positions updated automatically by the simulation engine. You only need to specify starting position and speed. When you tell the object to **`MoveTo`** a new position, the job of re-drawing, scaling, etc. is handled automatically.

There are approximately 40 modules in the SIMGRAPHICS II library. These contain objects used to build graphics images and user interface objects such as dialog boxes. The modules also contain a number of **virtual** object types. These are building blocks from which the other objects in the library are built. The virtual object types, whose names all end with **`VObj`**, are used only to define other object types and are never used directly.

The virtual types are made visible and accessible because they are the "foundation" of the graphics system and define many of the basic fields and methods inherited by the graphics object types which are used to build graphics and user-interface windowed programs.

Listed here are the object types which define the core of SIMGRAPHICS II:

- **`GraphicVObj`** - This virtual object is the base graphic object type. It contains fields and methods which are used by the graphics system to manipulate and manage graphical objects. Objects derived from **`GraphicVObj`** can be drawn/erased, positioned, selected, hidden, copied and made selectable or non-selectable. They are capable of loading/saving their graphical representation from/to a library object, and adding/removing/manipulating child objects.
- **`ControlVObj`** - This virtual object type is the base class for object types which support user interaction, such as dialog boxes and menus.
- **`WindowObj`** - Corresponds to a display window. This is the canvas upon which all the graphic objects appear. It can be sized and positioned programatically or by user interaction with the windowing system. Its default drawing area is the largest centered square within a display window, but it can be made to appear anywhere on the user's screen, fill the entire screen or open as a rectangular rather than square window. It defines a base coordinate system whose default is (0, 0) - (32,767, 32,767). The origin is at the lower left corner of the window.

  It contains an optional menu bar, palettes, status bar, and graphical images and has associated dialog boxes. It supports mouse tracking and button click detection.

The user can override its **MouseClick** and **MouseMove** methods and add custom functionality to these actions.

Its appearance and the way in which users re-size and move it follow the standards of the particular windowing system such as Microsoft Windows, OSF/Motif etc.

- **ImageObj** - This is the base class for all graphics other than dialog boxes, menus, etc. It defines the fields which govern the object's appearance. Objects derived from **ImageObj** can be assigned color, and can be highlighted, scaled, rotated, and translated. Hierarchies of these objects can be built, so child images can be added to an **ImageObj** object and positioned relative to its coordinate system.

- **GraphicLibObj** - This object can hold descriptions of any graphic objects, including icons, dialog boxes and menus. It is typically loaded with these images from a file produced by MODSIM's graphics editor, SIMDRAW. SIMDRAW allows the user to build all aspects of a graphical user interface from icons to dialog boxes. SIMDRAW stores the descriptions of graphic objects in files with the extension "**.SG2**." The files can be saved in text format, for portability, or in binary format for speed.

12.4 Example: Drawing an Image in a Window

This example assumes that the user has previously drawn a representation of the graphic object called 'airplane' using the SIMDRAW graphical editor and stored this image in a file called **mypics.sg2**. The graphical depiction called **airplane** is then associated with the **ImageObj** called **myIcon** in this program, and is then displayed.

```
MAIN MODULE demo1;

FROM Window  IMPORT WindowObj;
FROM Image   IMPORT ImageObj;
FROM Graphic IMPORT GraphicLibObj;

VAR
  win    : WindowObj;
  lib    : GraphicLibObj;
  myIcon : ImageObj;
  input  : ANYOBJ;

BEGIN
  NEW( win ); { Create a window }
  NEW( lib ); { Create a library and load it from a file }
  ASK lib TO ReadFromFile ("mypics.sg2");
  NEW( myIcon );  { Create an image and load its graphical
                    representation from the library }
  ASK myIcon TO LoadFromLibrary (lib, "airplane");
             { Add icon to window and ask window to Draw }
  ASK win TO AddGraphic( myIcon );
  ASK win TO Draw;
             { Wait for user to click in window }
```

```
    input := ASK win TO AcceptInput;
END MODULE.
```

This short program illustrates a powerful feature of SIMGRAPHICS II. The graphic which the program displays is first drawn by the programmer using the SIMDRAW graphics editor, and is then saved to a file. When the program runs, it loads this description of the image into a library and then associates that image with a graphical object called **myIcon** and draws it in the Window. If the program had a menu bar or used dialog boxes for user interaction, these could also be built using the **Graphics Editor**.

SIMGRAPHICS II also allows the programmer to make calls to the graphics libraries and draw images, dialog boxes, menus, etc. programatically.

Unlike many windowed systems, SIMGRAPHICS II does not bind the graphic objects to its executable. They exist in a separate file, in this case **mypics.sg2,** and are loaded by the program when the program is run. This means that changes to icons and dialog boxes can be made without recompiling and linking the program.

12.5 SIMDRAW - the Graphics Editor

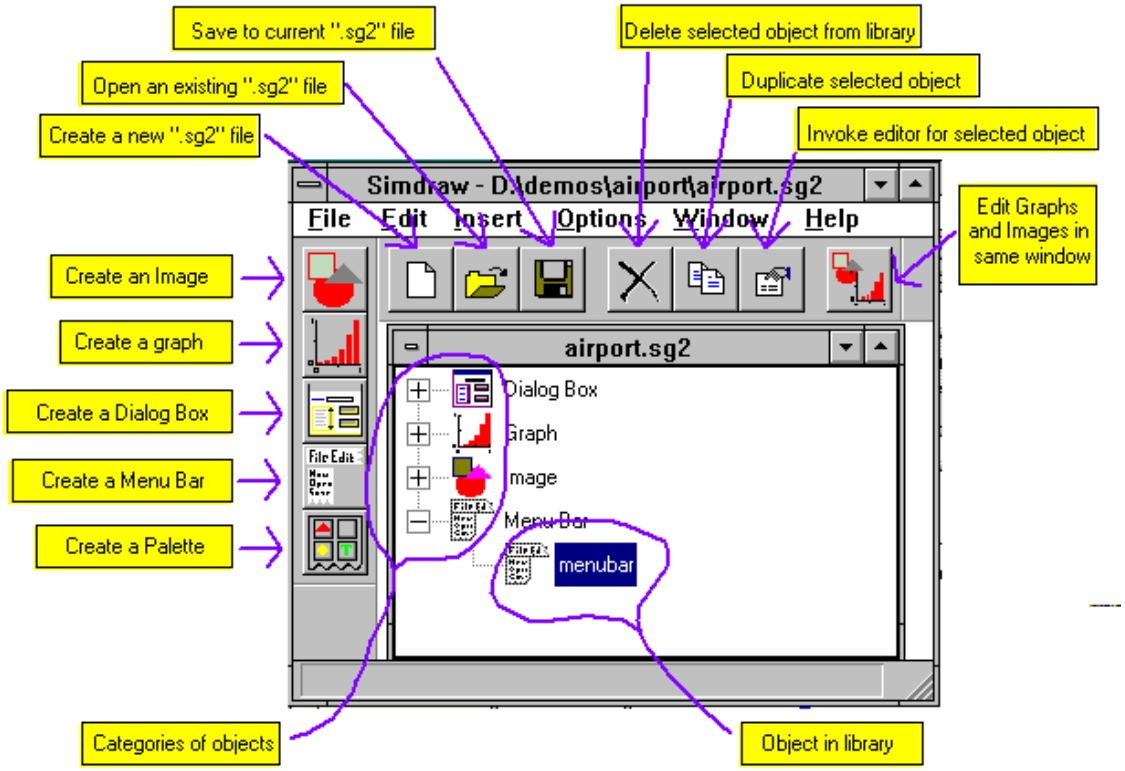
SIMDRAW is the editor provided by the SIMGRAPHICS II system to create and edit graphic images, charts and graphs and user interface items such as menu bars, dialog boxes and palettes. The editor is portable across all systems on which MODSIM III is available. It produces graphics files which are also portable.

SIMGRAPHICS II provides all of the methods and procedures needed to create and modify graphic items programatically, but, since it is much simpler to do this kind of work with SIMDRAW, the capability is usually used only when graphics or user interface items must be created or changed by a program “on the fly”.

Graphic items are built from primitives such as lines, circles, polygons, etc. Typically these parts are grouped together for convenience in handling. Each group and primitive can have a name and integer ID. These are used as “handles” to identify the graphic objects when they are in a graphics library or when they are part of another graphics object.

Any graphics itemcan be saved in a graphics library.

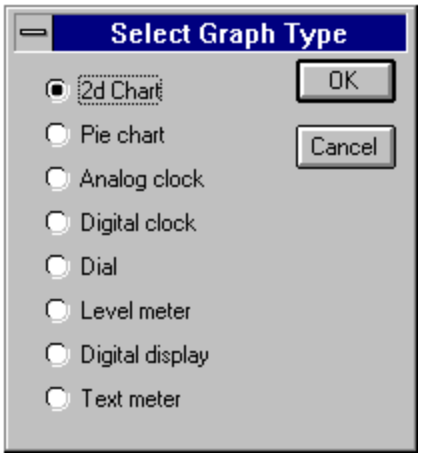
SIMDRAW's main window is shown below. This window categorically lists all objects contained in the currently loaded library file. From this window you can create and edit images, graphs (2-D charts, level meters, etc.), dialog boxes, menu bars, and palettes. A separate window is created for each object being edited. This allows you to copy parts of an object into the *clipboard* and paste them into another object of the same type. To add an object to the library, select one of the palette buttons on the left side of the window. Editing an existing object can be accomplished by double clicking on its name in the listing.



Double clicking on an image in the listing will invoke a separate window called the **Image Editor** which will contain only that image. Images are composed of circles, polygons, se c-tors, arcs, polylines, text, and bitmaps. Primitives are added to the image by selecting a primitive type from the **Mode** palette on the left. Bitmaps are added using the **File/Import** option. Exiting primitives can be repositioned, resized, flipped, and rotated. The style and color of a selected primitive can be changed using the **Color** palette on the bottom and the **Style** palette on the right side of the **Editor**. Points defining a polygon or polyline can be added, removed and repositioned.

A **Layout Editor** allows you to position and resize any number of images and graphs within the same window.

The **Graph Editor** allows you to edit variety of 2-D charts, pie charts, clocks and meters. Clicking on the **Bar Chart** palette button on the left side of the **List** window will present the followingdialog:



The **2-D Chart** is a standard x/y plot which can be either static or dynamic. It can have many data sets and be a bar graph, histogram, surface chart or line graph.

The **Level Meter** is a "thermometer" type graph used to show the value of a variable. It is updated dynamicallyas the model runs.

The **Dial** is the rotary equivalent to the **Level Meter**. It is analogous to a pressure gauge with an analog pointer.

The **Pie Chart** can be either dynamic or static.

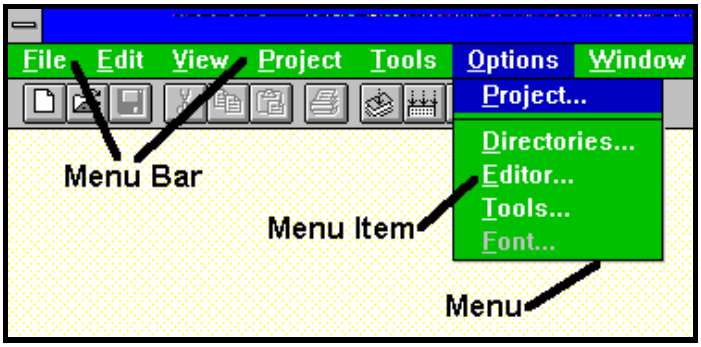
The **Digital** and **Text Display**s are simple controls which can contain numeric or text data. They can be updated dynamically while a program is running.

12.6 Constructing a User Interface

Users of windowed programs perform most interaction through controls such as menus, palettes and dialog boxes. In the below illustration a menu bar, menu and menu item are shown. Immediately below the menu bar is the pallete containing palette buttons, each with an icon. For further information on palettes, refer to paragraph 12.

In SIMGRAPHICS II menus, palettes and dialog boxes are usually created using the SIMDRAW **Menu Bar** dialog box and **Palette Editor**s. The components in the dialog boxes, menu bars and palettes are then connected to the program's code. These user interface items can also be constructed and modified programatically by making calls to the SIMGRAPHICS II library.

The most basic interface is the familiar menu bar which usually appears across the top of the window in most systems. In some systems the menu can appear instead as a small box containing a list of initial choices. The menu bar is an asynchronous or non-modal control which is always available for the user to manipulate. While the user is making choices on the menu bar, the program continues to operate without interruption.



Like most SIMGRAPHICS II constructs, the menu is a hierarchical construct. At the highest level is the menu bar itself. This serves as a container for menus. These are the choices that appear across the menu bar. Contained within the menus are either additional menus or “menu items”. These are the actual choices which the user makes to control some aspect of the program’s operation.

In most cases the menu bar is employed only for the most commonly used or essential aspects of a user's interface with the program. More detailed interaction is typically handled through a dialog box. The dialog box is simply a container object in which an assortment of other controls can be placed. It resizes automatically to fit its contents.

By its nature, the dialog box offers considerable flexibility to the programmer. In its simplest form it can be a short note to the user with an **OK** button to clear the dialog box from the screen once the user has read the note. More complex dialog boxes may fill the user’s screen with a complex arrangement of information and choices. Although the dialog box and its contents are called "controls", they are often used simply as a way to organize and present information to the user.

The controls which can be placed in a SIMGRAPHICS II dialog box, the **`DialogBoxObj`**, will be familiar to the user who has experience with contemporary windowed systems. They include:

`ButtonObj` -	Push to indicate a selection. The button has an optional label.
`CheckBoxObj` -	Presents and receives simple TRUE / FALSE or YES / NO input which is toggled each time the check box is selected.
`TextBoxObj` -	Receives textual input.
`ValueBoxObj` -	Receives numeric input and optionally does range checking on that input.

RadioButtonObj -	Used to indicate a selection which is mutually exclusive among a group of radio buttons in a **RadioBoxObj.**
RadioBoxObj -	The container for a group of radio buttons.
ListBoxObj -	Allows the user to scroll through and select from a list of items. Each item is a **ListBoxItemObj.**
LabelObj -	Used to label controls or present information to the user.
MultiLineBoxObj -	Implements a multi-line text edit box.
ComboBoxObj -	Recieves text input but has an attached "drop-down" list containing a list of alternatives.
TreeObj -	Used to show a list of items hierarchically. Item labels can have a small bitmap icon next to them to identify their category.
TableObj -	Composed of a 2-D array of selectable fields. Each field contains a text string. Tables can have selectable row and column headers and are scrollable if the size of the fields is greater the the table size.
TabObj -	Helps to organize controls into stacked *pages*. Only controls on the top page can be seen.

Dialog boxes can be either modal or modeless. When a modal dialog box is placed on the screen, no other user interactions outside of the dialog box are possible and the MODSIM III program pauses execution.

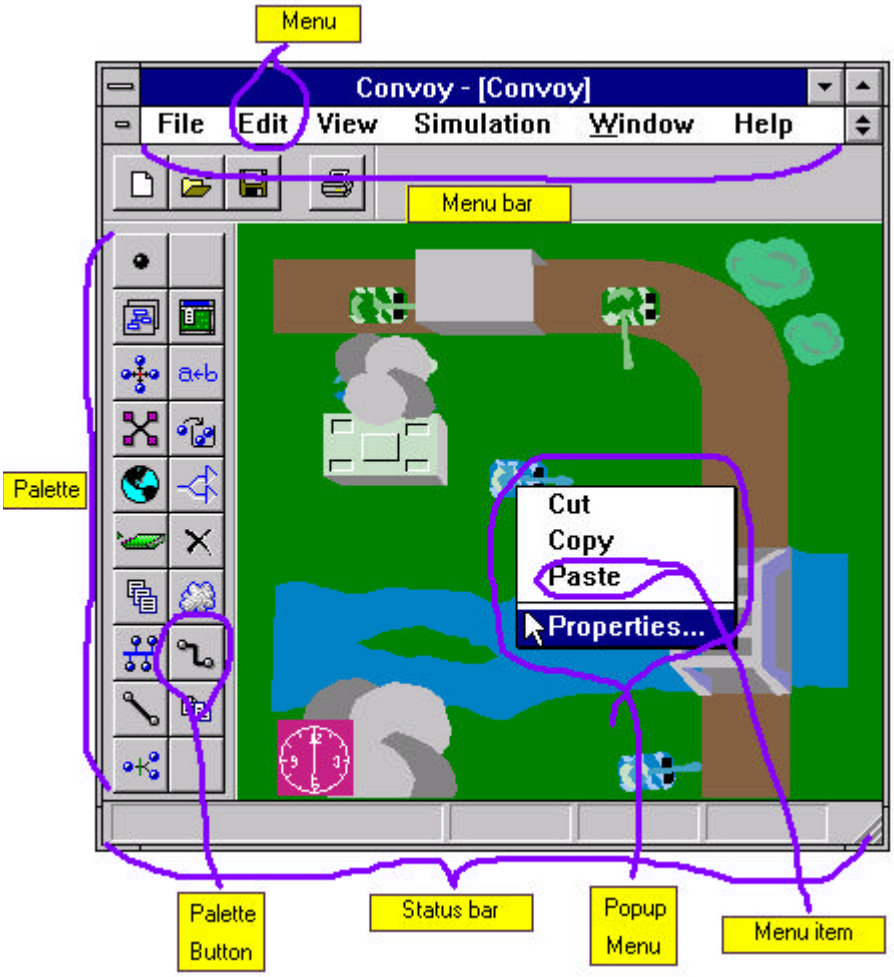
When a modeless dialog box is placed on the screen, the MODSIM III program continues to execute and the user can continue to interact with other controls outside of the dialog box.

12.7 Palettes

A palette (as shown below) contains rows and columns of selectable *palette buttons*. A palette button is a square button containing a bitmap "picture" on its face. These buttons can be used as a short cut for the menu bar, or to show which mode or style the program is currently using. Palette buttons can "toggle" (stay down or up until the next time they are selected), or be "momentary" and pop back up after being pressed.

The palettes themselves can be attached to any side of your window or be "floating" and behave like a dialog box. On MS Windows systems, palettes are "dockable" meaning that they can be detached from one side of the window and reattached to another side with the mouse.

Palette separators can be added to a palette in the appropriate spot to create a gap between palette buttons.

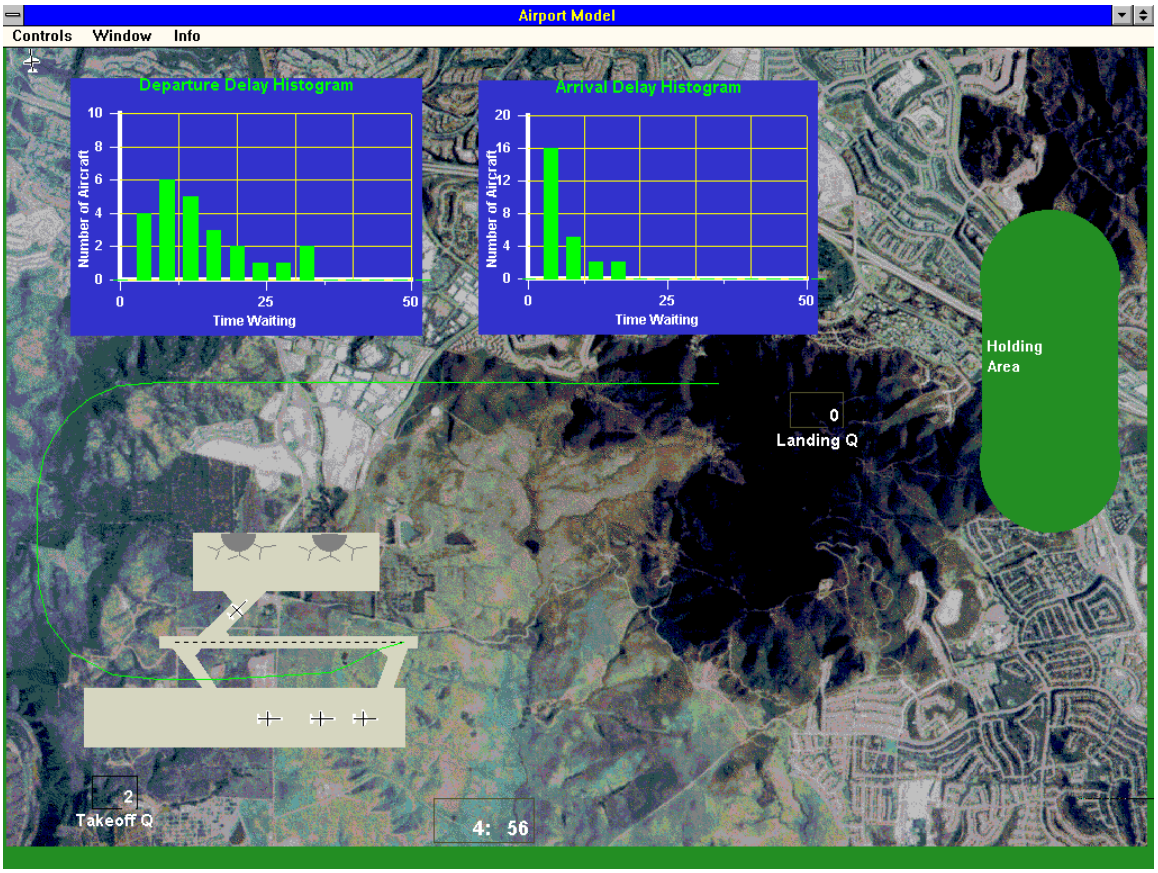

Menu
Convoy - [Convoy]
File
Edit
View
Simulation
Window
Help
Menu bar
Palette
Cut
Copy
Paste
Properties...
Palette Button
Status bar
Popup Menu
Menu item

13. An Animated Airport Model

The simple airport model presented its results as a set of dry statistics at the completion of a model run:

```
Simulation Run is Finished at SimTime 1466.567241
Mean interarrival rate used for arrivals & departures: 6.000000
# of arriving AC: 243    # of departing AC: 230
# of go arounds = 4
Max # AC waiting to takeoff & Mean # waiting to takeoff: 25 - 10.467890
Mean time spent departing: 60.910749
Max # AC waiting to land & Mean # waiting to land: 6 - 1.344961
Mean time spent landing: 6.098950
```

The statistics were averaged over the entire run but lacked information about variance in the data. They also provided no feel for the interaction of the objects in the model. We might learn something if we could observe the model in action. This is the motivation for an animated model.



It would also be advantageous if the user could specify more of the operating parameters and perhaps be able to adjust them while the model is running.

Finally, it would be nice to have the statistics presented in a more meaningful manner. Often, histograms will tell much about the data being observed that would not be apparent when presented as a simple mean and variance.

The design goals for the animated airport model took into account all of the above factors and added a few more goals and constraints:

- It had to be based on the simple airport model
- It should be useful as a demonstration model
- It should have a fairly accurate appearance
- Its animation speed should be adjustable so that results can be seen in "real time" or X times real time.

The resulting model has about 815 lines of code as opposed to 340 in the simple version. Surprisingly, it is not animation which accounts for most of the code inflation but the a d-dition of considerable user interaction.

The requirement to make appearance accurate resulted in the construction of a graphical queue object which was fairly complex. This could have been omitted from the model with absolutely no effect on model results or usability.

13.1 The Model Design Process

To add animation to the program we first had to decide on a background against which to present the model. In this case an aerial photograph was chosen and scanned as a bit-map. It was imported into SIMDRAW and used as the backdrop for the remainder of the work. SIMDRAW was used to draw in an airport runway, taxiway and terminal facilities. It was also used to draw the aircraft, the paths to be followed for takeoff and landing and the delay histograms. Finally, SIMDRAW was used to design the menus and dialog boxes which make up the user interface.

After this work was done, code was written to connect this graphical world with the pr o-gram's existing code. The most important item was to import **`DynImageObj`** so the air-craft object could inherit from it the ability to be animated.

```
FROM Animate IMPORT DynImageObj;
```

When the aircraft executes its **`ObjInit`** method, we added code so that it could load it graphical representation from the graphics library:

```
ASK  METHOD ObjInit;  { generic }
BEGIN
  INHERITED ObjInit;
  LoadFromLibrary(lib, "plane");
  ASK Window TO AddGraphic(SELF);
  SetAutoRotation(TRUE);
  startTime := SimTime();
END METHOD;
```

We also changed the design of the aircraft object family by making it a descendent of the **DynImageObj**. There are few other changes to the aircraft type declarations. Most of the fields and methods needed to animate them are inherited from **DynImageObj.**

```
AircraftObj = OBJECT(DynImageObj)
  taskSpeed : REAL; { speed for takeoff or approach }
  startTime : REAL; { sim time at which AC starts Land or TO }
  OVERRIDE
    ASK METHOD ObjInit;
    ASK METHOD StopMotion; { logic to reset Motion flag }
END OBJECT;

TakeoffObj = OBJECT(AircraftObj);
  TELL METHOD Takeoff;
  OVERRIDE
    ASK  METHOD ObjInit; { set takeoff performance attributes }
END OBJECT;

LandObj = OBJECT(AircraftObj);
  landPriority : priorityType; { normal or goAround }
  TELL METHOD Land;
  TELL METHOD GoAround;
  OVERRIDE
    ASK  METHOD ObjInit;  { set landing performance attributes }
END OBJECT;
```

The next most obvious change is that the aircraft no longer simply waits for time to pass while it pretends to takeoff or land. The new code actually makes the aircraft move from one point to another. Simulation time passes while the aircraft moves.

Here is the aircraft's landing code from the simple model:

```
WAIT DURATION taskTime
END WAIT;
TELL Controller ClearOfApproach;
IF (runway <> clear)  { is runway clear? }
  GoAround;
  RETURN;
END IF;
runway := inUse;
WAIT DURATION ovhdTime  {  roll out time }
END WAIT;
TELL Controller ClearOfRunway;
DISPOSE(SELF);
```

Now examine the aircraft's landing code from the animated version of the model. In this version, every action of the aircraft as it approaches, lands, rolls out and taxis off the ru n-way is modeled

```
SetSpeed(taskSpeed);           { approach / landing speed }
SetAutoRotation(TRUE);   { align in direction of movement }
WAIT FOR SELF TO MoveTo(3.5, 2.0);
END WAIT;
TELL Controller ClearOfApproach;
IF (runway <> clear)  { is runway clear? }
```

```
    GoAround;
    RETURN;
  END IF;
  ASK ArriveGen LogCompletion(startTime);
  runway := inUse;
  SetSpeed( taskSpeed * 0.7);         { initial rollout }
  WAIT FOR SELF TO MoveTo(2.5, 2.0);
  END WAIT;
  SetSpeed( taskSpeed * 0.3);         { final rollout }
  WAIT FOR SELF TO MoveTo(1.8, 2.0);
  END WAIT; SetRotationSpeed(-2.8);
  WAIT FOR SELF TO RotateTo(1.4); { -090 degrees as radians }
  END WAIT;
  WAIT FOR SELF TO MoveTo(2.0, 2.2);
  END WAIT;
  TELL Controller ClearOfRunway;
  SetSpeed( 10.0 / 60.0);  { all planes taxi at 10 Kts }
  WAIT FOR SELF TO MoveTo(2.6, 2.8);
  END WAIT;
  DISPOSE(SELF);
```

Notice that the speed of the aircraft is first set, and then the aircraft is told to move (from its present position) to a new position. The positions in this model are expressed in naut i-cal miles on an X-Y grid. When the aircraft is ready to taxi off the runway, its rotation speed is set and we wait while it turns and starts taxiing. Not only does this code result in an accurate animation of the aircraft, but it also adds to the accuracy of the model since the time taken to complete each separate action is accounted for automatically by the sy s-tem.

Glossary

activity: A **WAIT** statement in a **TELL** method. The place in an object's **TELL** method where simulation time elapses.

base type: The immediate ancestor or the immediate underlying object type of an object type.

behavior: A method of an object implements the object's behavior.

BNF: Backus-Naur Form. The notation used to describe the syntax of the language. Note that this notation is not capable of describing semantics or semantic limitations of the syntax.

class: As in "object class". Meaning is the same as "object type". The term is used to describe the type definition for an object in languages which have a weak notion of types such as C++.

component: Either a field or method for an object

conflicting methods: This occurs when two or more of the base types in a multiple inheritance have a method with the same name.

derived type: An object type defined in terms of one or more existing object types; each of these types is a base type

dynamic binding: The type of each operand and operation is determined at run-time; most object-oriented languages, including MODSIM, are based on dynamic binding. MODSIM uses dynamic binding only for field references and method calls, not for other operations such as **+**, **-**, **AND**, etc.

encapsulation: Packaging the fields which define the state of an object and the methods which define its behaviors within one object definition.

enumeration: A user-defined ordered set of literal values
e.g. `workday = ( Mon, Tue, Wed, Thu, Fri )`

field: One of the variables associated with a particular object or record type.

function method: A method which returns a value. Only **ASK** methods can return a value. Therefore, **TELL** methods cannot be function methods.

group: A structure used to associate objects. Examples are: **StackObj**, **QueueObj**, **RankedObj, BTreeObj**. Comparable to SIMSCRIPT II.5's SET or a Smalltalk Collection.

inheritance: The definition of one object type in terms of another, already-existing object type.

instance: One particular object of an object type.

invoke: To call a procedure or method. To cause a procedure or method to execute.

member: An object which is contained within a group.

message: The name of a method; "sending message A to B" is an equivalent way of saying "ask object B to perform method A" or "perform method A with object B".

method: A routine which describes an object's behavior. Similar to a procedure. However, a method is **always** associated with an object.

object: A dynamic data structure that includes an associated list of methods.

ordinal type: A type which has a known ordering. In other words, given one value which belongs to the type, it is possible to state what the next or previous value would be. The following are ordinal types: **`INTEGER`**, **`CHAR`**, **`BOOLEAN`**, enumerations and subranges.

pass by reference: When a parameter in a parameter list is shared by both the invoking and the invoked routine. Parameters with the **`INOUT`** and **`OUT`** qualifier are passed by reference.

pass by value: When a copy of a parameter in a parameter list is made and passed in to the invoked routine. Parameters with the **`IN`** qualifier are passed by value.

private property: A property with a scope limited to the methods of an object type or derived object types. If a field or method of an object is declared to be private, it cannot be accessed or invoked except from the object itself.

process: Process-based simulations allow methods of objects to describe a series of related activities rather than being limited to defining simply one event per method.

proper method: An untyped method that has no return value. Can be either a **`TELL`** or **`ASK`** method.

property: A characteristic or attribute of an object type. Specifically either a method or field of the object type.

public property: A property of an object that is available for use outside the methods of that object type.

qualified inherited call: In a multiple inheritance, an invocation of an inherited method of a specific base type, as in:

```
INHERITED FROM SomeObject aMethod;
```

record: A data structure which consists of a collection of fields which may be variables of differing types

reference type: Each object type has a reference type, which is used to define variables that reference a specific object of that type - analogous to a pointer type in other languages.

routine: A general term for a sub-routine, procedure, function or method.

scalar type: A type which has only one element or component part and can be used to scale, measure or quantify things. The following are scalar types: **`INTEGER`**, **`REAL`**, **`CHAR`**, **`BOOLEAN`**, enumerations and subranges. An example of something which would not be a scalar type is an array, record or object type.

SELF: Built-in reference variable which is defined within every method. It allows reference to the object instance from within its own methods.

shared variable: A variable which is shared by all the methods of a particular object type. In other words a variable defined outside the scope of an object so that it will be visible to all instances of that object type. Usually a shared variable is defined globally, within a module.

strong typing: The type of each operand, parameter and operation is fixed at compile-time. MODSIM III, Ada, Pascal and Modula-2 are characterized by strong typing.

TELL method: A proper method which is executed asynchronously. It can elapse simulation time. If it has a parameter list, only **`IN`** parameters are allowed. **`WAIT`** statements are allowed in **`TELL`** methods.

time-elapsing method: A **`TELL METHOD`** which contains at least one **`WAIT`** statement.

underlying type: If type **`A`** is derived from type **`B`**, or from some type which is in turn derived from **`B`**, then **`B`** is said to be an underlying type of **`A`**.

Virtual object type: An object type which is not used directly. It exists only to serve as a base type from which other objects are derived. It often serves as an “anchor” to relate two other object types.

Index

G

I

L

M

N

O

P