National Library of Canada
Canadian Theses Service
Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Service des thèses canadiennes

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-56064-9

Canada

An Object-Oriented Discrete-Event Simulation System With a Graphical User Interface

Thomas S. Wieland

A Thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

April 1990

© Thomas S. Wieland, 1990

Abstract

An Object-Oriented Discrete-Event Simulation System With a Graphical Use Interface

Thomas S. Wieland

It has often been noted by researchers that the programming models of object-oriented programming and discrete-event simulation exhibit a strong similarity. Inspired by the ideas published in [Zeigler 1987], we present the design of a Modular Simulation System (MSS) which employs object-oriented techniques to allow the combination of separately developed simulation modules into a simulation program.

We view a simulation program, or simulator, as a collection of self-contained simulation modules which use messages to communicate with each other about changes of states and other events. These simulation modules are stored in one or more module libraries and the characteristics of modules are described in a module description file.

With the help of the **SimBuild** program, a tool which provides an icon-based graphical user interface to the module libraries, a user can interactively and rapidly create a simulator and configure its simulation modules. The simulator description so obtained is then processed by the **SimMake** program to generate an executable simulator without further user interaction.

In this thesis we present the design of MSS, including that of **SimBuild**, **SimMake**, and other related programs, files and data structures, and module interconnection techniques. The design has been implemented using the object-oriented programming language C++ and the SAMOC discrete-event simulation package. A typical simulator generation is also presented, with a discussion of the generated program and its results.

Acknowledgements

First and foremost, I would like to thank my thesis supervisor, Dr. T. Radhakrishnan, for the support and encouragement he has given me during the course of my studies. I am very grateful for the advice he has offered and for the patience and understanding he has shown towards me.

I would further like to thank Mr. Clifford Grossner for his friendship and for the help he offered on numerous occasions.

I thank Ms. Jean Marjama for implementing the SimBuild program; her help has been most valuable.

To all my friends in the Department of Computer Science, especially those in the L annex, thank you for sharing the good times and for being there during the bad; your friendship has made this a most memorable experience. Special thanks to Dimitrios Livas for drawing some of the figures used in the thesis.

I gratefully acknowledge the financial support given by **Bell-Northern Research, Ltd.** at Nuns' Island, Montreal which has made my studies and this research possible.

Finally, many thanks to my parents and my wife for their confidence and support.

Epigram

Although the envious nature of men, so prompt to blame and so slow to praise, makes the discovery and introduction of any new principles and systems as dangerous almost as the exploration of unknown seas and continents, yet, animated by that desire which impels me to do what may prove for the common benefit of all, I have resolved to open a new route, which has not yet been followed by any one, and may prove difficult and troublesome, but may also bring me some reward in the approbation of those who will kindly appreciate my efforts.

And if my poor talents, my little experience of the present and insufficient study of the past, should make the result of my labors defective and of little utility, I shall at least have shown the way to others, who will carry out my views with greater ability, eloquence, and judgement, so that if I do not merit praise, I ought at least not to incur censure.

Niccolo Machiavelli (1469-1527)
Introduction to *The Discourses*

Table of Contents

1. Introduction 1
    1.1. An Introduction to Simulation 1
    1.2. Simulation and Analysis 2
    1.3. Scope of the Thesis 2
2. The Object-Oriented Approach to Software Design 4
    2.1. The Object-Oriented Approach to System Design 4
    2.2. Features of Object-Oriented Programming 6
    2.3. An Introduction to C++ 10
    2.4. Benefits of Object-Oriented Programming 13
3. The Modular Approach to Simulation 16
    3.1. Discrete Event Simulation 16
    3.2. Monolithic vs. Modular Simulation 19
    3.3. Object-Oriented Programming and Modular Simulation 20
    3.4. Composite Models and Model Hierarchies 21
    3.5. Modeling at Varying Levels of Detail 24
4. An Introduction to SAMOC 27
    4.1. Entities 27
    4.2. Queues 30
    4.3. Synchronization Objects 32
    4.4. Statistics and Report Generation 35
    4.5. Distribution Objects 37
    4.6. Other SAMOC Facilities 38
5. Interactive Generation of the Simulator 41
    5.1. Models, Modules and Module Libraries 41
        5.1.1. Simulation Modules 41
        5.1.2. Simulation Module Library 43
    5.2. Possible Users of a Simulator 44

5.3. The Simulator Development Cycle 47

5.4. A Tool for Interactive Simulator Generation 49

6. The Design and Implementation of the Modular Simulation System 55

6.1. Design and Structure of MSS 55

6.2. Integration of Modules Into a System 59

6.2.1. Design of the Simulation Module Base Class 60

6.2.2. Implementation of the Simulation Module Base Class 66

6.3. Utilities: SimDesc and SimMake 71

6.3.1. SimDesc 72

6.3.2. SimMake 75

6.4. Simulation Design Using MSS 79

7. The Development of a Sample Simulator 82

7.1. Development of the Simulation Modules 82

7.1.1. Messages 82

7.1.2. CPU Module 85

7.1.3. Bus Module 90

7.1.4. Memory Module 91

7.2. Generation of the Module Library 91

7.2.1. Creation of the Module Description File 91

7.2.2. Creation of the Module Object Library 94

7.3. Generation of the Simulator Description 96

7.4. Generation of the Simulator 96

7.5. Execution of the Simulator 98

7.6. Refinement of the Simulator 100

7.6.1. Enhancing the Simulation Modules 101

7.6.2. Router Module 102

7.6.3. Generation of the Expanded Simulator 103

7.7. Future Refinement Cycles 105

8. Conclusion 107

References 110

Appendix A. Source Code for MSS Base Classes 114
A.1. MSS Base Classes 114
A.1.1. Inter-Module Messaging: Classes *msg* and *msg_queue* 114
A.1.2. Simulation Modules: Class *simmod* 116
A.1.3. Implementation of Base Classes 117
A.2. Include and Make Files 123
A.2.1. *C* Include File 123
A.2.2. Makefiles 124
A.2.2.1. Directory Hierarchy 124
A.2.2.2. Compilation Rules 125
A.2.3. Module Generation 126
A.2.4. Simulator Generation 128
A.3. Router Modules 129
Appendix B. Format of MDF, SDF, and SGF Files 133
B.1. Module Description File Structure 133
B.2. Simulator Description File Structure 135
B.3. Simulator Graphics File Structure 136
Appendix C. Listings For Example Simulator 137
C.1. Inter-Module Messages 137
C.2. CPU Module 139
C.3. Bus Module 141
C.4. Memory Module 144
C.5. Module Description File 145
C.6. Simulator Description File 148
C.7. Simulator Source File 149
C.8. Output Generated By Simulator 152
C.8.1. Default Simulator 152
C.8.2. Simulator with Slower Memory 152

Appendix D. Listings For Expanded Example Simulator 154
D.1. CPU Module 154
D.2. Bus Module 155
D.3. Memory Module 157
D.4. Output Generated By Simulator 158

List of Figures

Figure 2.1: Different Views of a System 5

Figure 2.2: Data Encapsulation 6

Figure 2.3: Inheritance and Polymorphism 8

Figure 2.4: Early vs. Late Binding 9

Figure 3.1: System Modeling Concepts: Event, Activity, Process 17

Figure 3.2: Model Composition 22

Figure 3.3: Model Hierarchy 23

Figure 3.4: Levels of Detail 25

Figure 4.1: Entity State Diagram 29

Figure 4.2: SAMOC Class Hierarchy 39

Figure 5.1: Structure of a Simulation Module 42

Figure 5.2: SimBuild Operation 49

Figure 5.3: SimBuild Screen Layout 50

Figure 5.4: Coupling of Modules 52

Figure 6.1: MSS Directory Structure 55

Figure 6.2: Simple CPU/Memory Models 63

Figure 6.3: Relation Between Classes in a Simulator 71

Figure 6.4: Simulator Creation Using MSS 81

Figure 7.1: Example Simulator 82

Figure 7.2: Display of the Generated Simulator 96

Figure 7.3: Expanded Example Simulator 100

Figure 7.4: Possible Use of Router Module 103

List of Tables

Table 4.1: Comparison of *bin* and *res* Classes 34

Table 5.1: Types of Simulator Users 47

Table 6.1: MSS Directory Contents 56

Table 6.2: MSS and UNIX Filename Extensions 57

Table 6.3: MSS system files 59

Table 6.4: MDF Keywords 74

Table 6.5: SDF Keywords 76

Table 7.1: Components of Module Header Comment 86

Table 7.2: Components of a Module Declaration 87

Table D.1: Generated Test Simulators 158

Table D.2: Results of the Simulator Test Runs 159

List of Listings

Listing 2.1: Example of C++ Class Declaration 11

Listing 2.2: Base Classes and Derived Classes 12

Listing 4.1: Declaration of Class *entity* 28

Listing 4.2: Declaration of Class *queue* 31

Listing 6.1: Pseudo Code for CPU and Memory Modules 64

Listing 6.2: Declaration of Class *msg* 67

Listing 6.3: Declaration of Class *msg_queue* 68

Listing 6.4: Declaration of Class *simmod* 69

Listing 7.1: Message Class Declarations 83

Listing 7.2: Declaration of Simple CPU Module 88

Listing 7.3: Implementation of Simple CPU Module 89

Listing 7.4: SimBuild Command Level Prompt 92

Listing 7.5: Adding a Module Description using SimDesc 94

Listing 7.6: Saving the Module Library 94

Listing 7.7: Commands for the Module Makefile 95

Listing 7.8: Generation of the Object Library 95

Listing 7.9: Generating the Simulator 97

Listing 7.10: Output of the Simulator 98

Listing 7.11: Output of Expanded Simulator 105

1. Introduction

1.1. An Introduction to Simulation

According to John McLeod, one of the pioneers of the use of computers in simulation, *"simulation is the use of models as surrogates for the study of existing or hypothesized systems."* As McLeod points out in [McLeod 1988], the above interpretation of the word *simulation* is general enough to include clocks (a simulation of the earth's revolution about its axis), military war-games (a simulation of troop movements), slide rules (an analog simulation of logarithms), and mechanical calculators (a simulation of the rules of algebra) as historical examples of simulations.

Moreover, a model, according to the dictionary a *"formal representation of a theory or a formal account of empirical observations"*, may be a physical representation of a system (e.g. a scaled-down replica) or a schematic diagram of a system (such as a map or a chart), as well as a symbolic description of a system using a mathematical formalism or a computer program. For the scope of this thesis, however, we are concerned solely with computer simulation, i.e. that branch of simulation which describes a system using a model based on mathematical functions and operational rules which are encoded in a computer program.

One of the basic objectives of studying any system, by simulation or otherwise, is to understand how changes occur in the system and, subsequently, how to predict and control these changes. The knowledge gained from a simulation can then be applied to: (i) *optimize* the system's performance, (ii) *evaluate* proposed changes to the system design, or (iii) *predict* future changes in the state of the system. The first two applications are primarily concerned with optimizing the output of a system according to a given set of performance measurements. Here simulation is used to model the behavior of the system and examine its output for various input conditions.

The third application treats the system and its input as fixed and attempts to answer questions about the system's future by observing the changes that occur within the simulation model. This kind of simulation is often used to test one's understanding of a system (i.e. the validity of the model) when direct manipulation of the system being modeled is impossible or infeasible (e.g. in astrophysics), or to predict the reaction of the system to changes in its input where it would take too long to determine it by observation (e.g. in environmental modeling).

1.2. Simulation and Analysis

To study the behavior of a system one may either employ simulation, or one may attempt to describe the system using an analytical model. In principle, the analytical model is always preferable to a simulation as it provides a general solution and thus a precise answer to every possible question one may want to ask of the model. In practice, however, the need to realistically reproduce the system, i.e. to include sufficient detail in the model, very quickly leads to such a degree of complexity in the model as to preclude an analytical solution.

While mathematical queuing theory provides models for many types of queueing systems, only a small subset of these models is known to have an analytical solution. Furthermore, to become tractable, most models, especially those with a known solution, have to make simplifying assumptions about the modeled system which are not realistic in real systems, such as the statistical independence of events occurring in the system [Sauer 1984]. Even the mathematical description of those simple systems for whose model the theory provides a solution method, is sometimes so large and requires such a computational effort for its solution as to be impracticable.

Computer simulation, on the other hand, imposes few, if any, restrictions on the type of system that can be modeled. The degree of detail with which a system is modeled depends for the most part on the time and effort the designer of a simulation is willing to invest. Additionally, computer simulation by its very nature provides flexible and easy control over external influences or the input to the modeled system.

1.3. Scope of the Thesis

In this thesis we present the design and implementation of a Modular Simulation System (MSS) and of a graphical user interface which allows the interactive generation of a simulation program from stored simulation modules.

Chapters two and three discuss the two foundations of our simulation system, namely *object-oriented programming* and *discrete-event simulation*. Chapter two introduces the most important features of object-oriented programming and includes a brief introduction to the C++ programming language. Chapter three introduces the basic concepts of discrete-event simulation and discusses various aspects of this particular kind of simulation.

Chapter four gives a short introduction to SAMOC, the discrete-event simulation package that was used in the implementation of our simulation system.

Chapter five introduces the building blocks used in the construction of a modular simulator, namely *simulation modules* and *simulation module libraries*. It discusses the possible users of the simulation system and the different requirements each class of user has of the system. We then describe an interactive, graphics-oriented program called **SimBuild** to generate a simulator from simulation modules.

Chapter six presents the design of the Modular Simulation System (MSS). It discusses the important issues encountered during the design and implementation of MSS and presents the structure of the system in detail. We also describe two programs, named **SimDesc** and **SimMake**, which assist in the generation of a simulator. Finally, we give an overview of the simulator construction process using MSS.

Chapter seven demonstrates the use of the Modular Simulation System by describing the steps necessary to build a sample simulator for a simple computer system. It also gives an example of how an existing simulation system can be refined by refining some of the modules and reusing some of the previously used modules in a simulator.

2. The Object-Oriented Approach to Software Design

This chapter gives a brief introduction to object-oriented design of software systems. First, we review the basic concepts of object-oriented design, followed by an introduction to object-oriented programming. Next, we briefly describe the C++ programming language and discuss it as an example of a real-world object-oriented programming language. Finally, we summarize the potential benefits of object-oriented programming.

2.1. The Object-Oriented Approach to System Design

A simple definition of object-oriented design (OOD) is given by [Meyer 1988]:

> "Object-oriented design [of a system] is the method which leads to software architectures based on the objects [that] every system or subsystem manipulates."

From this definition, two questions arise immediately, namely: "What is a system?" and: "What is an object?" Unfortunately, these two terms are among the hardest to define in a general sense as they mean different things to different people and in different situations. For now, we will use the following working definition for *system*, which was introduced by Kristen Nygaard, one of the inventors of object-oriented programming [Nygaard 1986]:

> "A system is a part of the world that a person chooses to regard as a whole consisting of components, each component characterized by properties that are selected as being relevant and by actions related to these properties and those of other components."

As this definition points out, exactly what constitutes a system depends on the point of view adopted by the person looking at the system; precisely that is what makes it so difficult to define the term *system*. It also suggests that a system may consist of a hierarchy of components which, from a different point of view, can themselves each be regarded as a system (see Figure 2.1). Finally, it points the way toward what constitutes an object in object-oriented programming terminology by stating that each component of a system, i.e. each object, is characterized by its data ("properties") and functions manipulating its data ("actions").

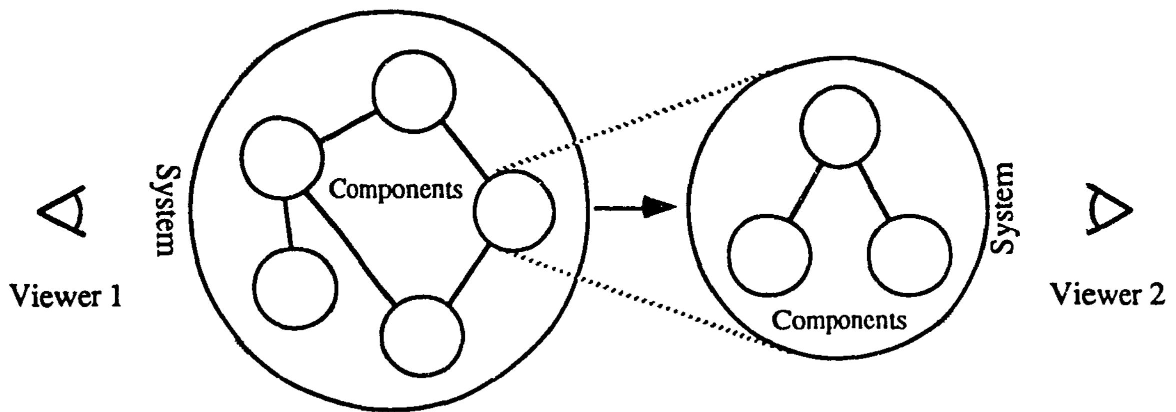

Figure 2.1: Different Views of a System

As implied by its name, and confirmed by the definition above, objects are at the core of any object-oriented methodology. But what exactly are those objects, and how can one find them? There seems to exist no definite method, no algorithm, to determine the relevant objects in a system. The reason for that may be that the answer to these questions is so simple and obvious that it defies a formal specification. Simply put, the objects employed in the design of a system are representations of those objects occurring in the real-world system modeled by the design.

By concentrating the attention of the designer upon the objects manipulated in a system, the object-oriented approach achieves a shift in focus away from the top-down functional breakdown of a system implied by traditional structured programming methodologies and towards the data processed and the data manipulation operations employed by the system. To paraphrase Bertrand Meyer, the designer no longer asks himself what the system as a whole does, but which objects it manipulates, and how. This leads to a more bottom-up approach to system design in which a designer first explores the properties of the individual components of the system before proceeding to the higher level question of what the purpose of the overall system is.

The advocated benefit of the object-oriented approach is that it naturally leads to a modular system design which is flexible and can be easily changed and extended. The techniques used to describe objects and to aid in the construction of these flexible software architectures, are explored in the following section.

2.2. Features of Object-Oriented Programming

The four most prominent features of object-oriented programming are (i) *data encapsulation*, (ii) *inheritance*, (iii) *polymorphism*, and (iv) *late binding*. Of these, data encapsulation is the most fundamental and the one on which the other three features are based and without which they could not work.

In order to describe an object in a formal way, a designer uses classes, and views each object in the system as an instance of a class. A class in object-oriented terminology is an abstract data type which describes an object as a set of properties which characterize the object and a list of services (actions) with which these properties can be manipulated. Taken as an abstract specification, a class does not specify any details of the implementation of these properties and services, but only the interface through which an object of the class can be manipulated. A class thus encapsulates an object's data in a shell which prevents other objects from changing its data in ways other than those intended and provided for by the class designer.

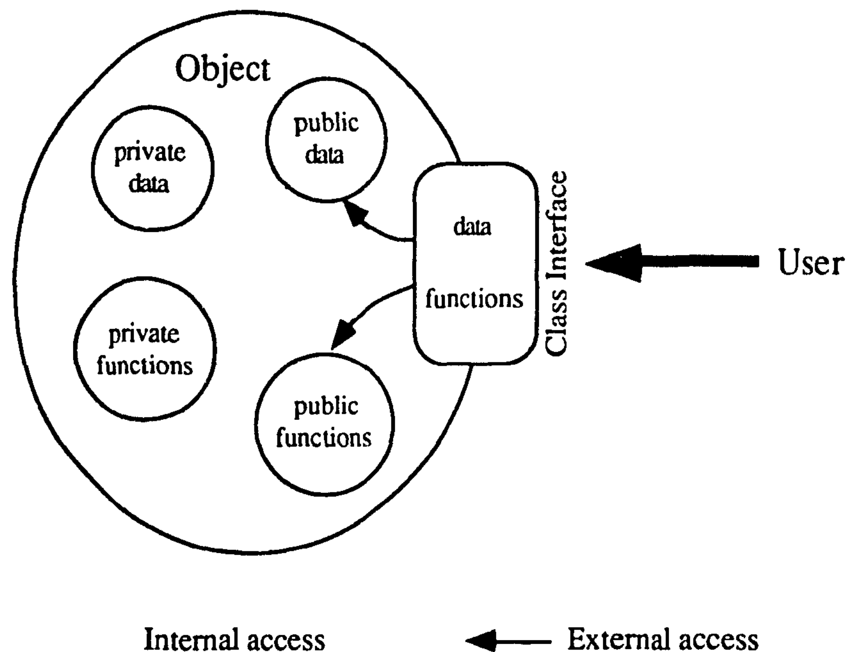

Figure 2.2: Data Encapsulation

For the actual implementation of an object in an object-oriented programming language, the class concept is directly supported as one of the basic constructs in the programming language. The abstract properties of the object and its services are

implemented as variables and functions in that programming language and are listed in the class interface definition. To distinguish them from other variables and functions which are not part of a class, a class' variables and functions are referred to as member variables and member functions. In addition to the variables and functions which are made public in the class interface definition, a class may also possess internal member variables and functions which are necessary or convenient for its implementation. These internal class members are referred to as private members and are accessible only by other member functions, while all public members of a class are accessible from outside the class. Figure 2.2 illustrates these concepts.

Most object-oriented programming languages include support for object creation, initialization, and destruction and for memory management by providing special constructor and destructor functions for each class. A constructor function allocates storage for an object and initializes the object's member variables upon creation; a destructor function is invoked when an object is destroyed and performs any necessary finalizations such as freeing the memory space occupied by the object and any of its dynamically allocated data structures. Constructor and destructor functions are essential to uphold data encapsulation. Without them, the programmer would need to provide external functions to properly initialize and finalize objects; even though they are not class members, these functions would need to access any or all of an object's data. In addition, constructors and destructors significantly reduce the complexity of managing objects for the programmer which is important as many objects may be created and destroyed during the lifetime of a program.

Objects sharing the same properties and the same actions, but possibly different values of their properties, are represented as different instances of one and the same class. Often, however, a group of objects shares certain common features while differing in some other features. In particular, some objects may be specializations of a more general class and may possess additional properties not present in other objects of its general class. Object-oriented programming addresses this situation through a mechanism called inheritance.

Inheritance enables the re-use of existing class designs by allowing a new class to declare itself a descendant of an existing class. Such a derived class inherits all properties and services from its parent class and may then extend or augment them by adding new properties and services of its own (see Figure 2.3). An object that is an instance of a derived class thus possesses the same member variables and functions as an object of its

parent class but also has some others which are particular to itself and not present in the parent class. Multiple inheritance is an extension of this concept which allows a derived class to inherit properties and services from two or more parent classes. As these parent classes may have been developed independent of each other, conflicts regarding the naming of member variables and functions and their usage may result from combining the classes and have to be resolved by the language designer.

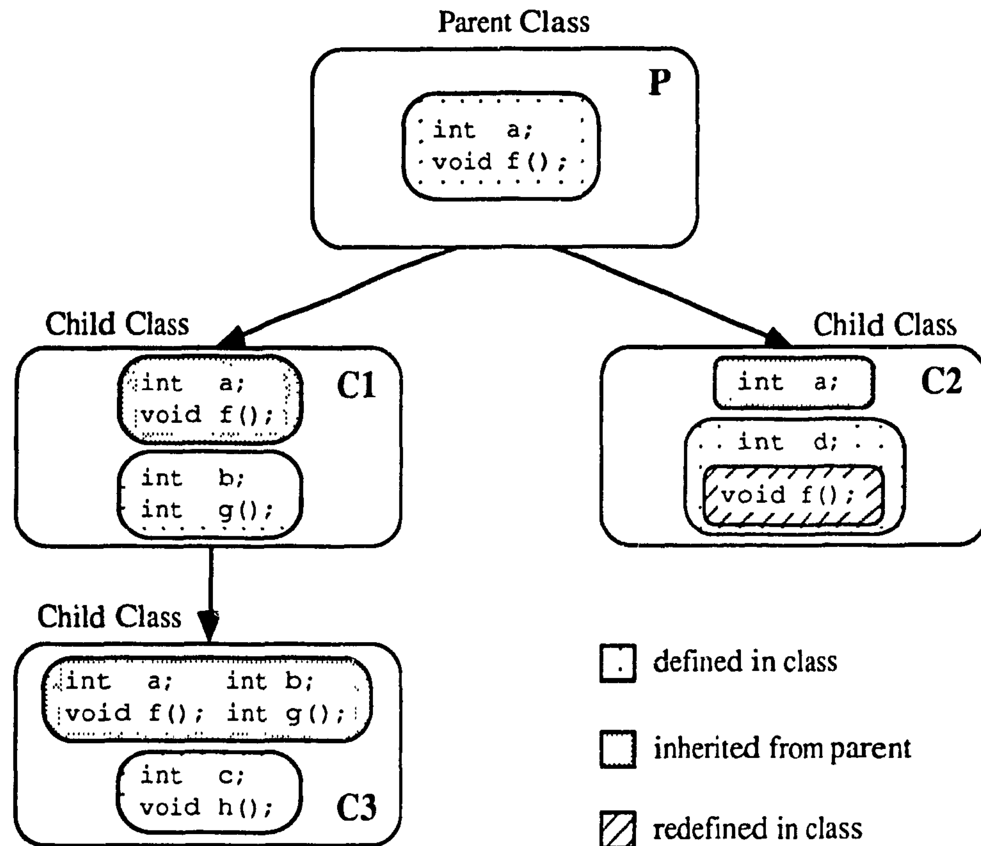

Figure 2.3: Inheritance and Polymorphism

While inheritance solves the problem of describing similar related classes of objects, it does not, by itself, address the situation where a class is a differentiation of a base class. For example, a group of objects may share common features with a class, but may implement a particular service differently from the one already provided by that class. This and similar situations are dealt with by the third essential feature of object-oriented programming, polymorphism.

Polymorphism, as used in object-oriented terminology, occurs in two related forms, one static (i.e. at compile-time) and one dynamic (i.e. at runtime) and is intimately

tied to inheritance and late binding. Polymorphism at compile-time is a form of overloading and allows a derived class to re-implement a service already implemented in its parent class; objects of the derived class will then execute the derived class' version of that service, while objects of the base class still act according to the original definition of the service. At runtime, polymorphism allows an object to reference objects of varying class without having to specify the exact class of the referenced object. What is encoded in the program is only a reference to another object and the service that the referenced object needs to perform. Depending on the language, the referenced object may be of any class (as for example in Smalltalk) or the object reference may be associated with a particular class and the class of the referenced object is then restricted to that class and to its derived classes (as for example in C++).

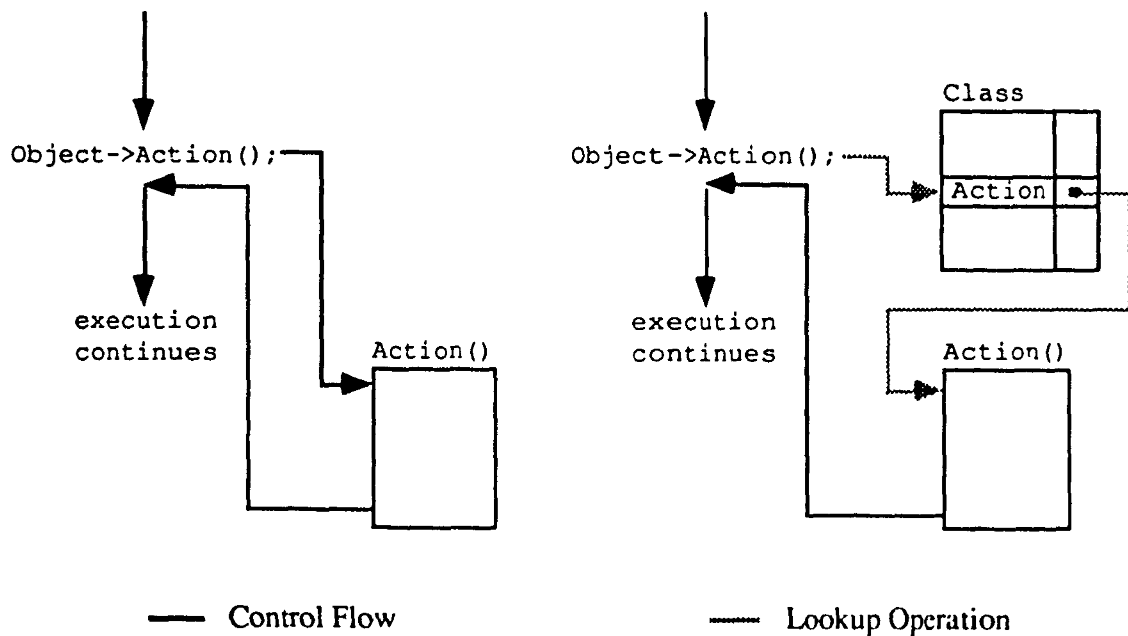

Figure 2.4: Early vs. Late Binding

What makes the dynamic selection of the appropriate function possible is late binding, often also called *dynamic binding*. *Binding* is a term referring to the time at which a decision is made concerning which function is to be executed. In early (or static) binding, the compiler decides which function is to be called at runtime depending on the scope rules of the language and generates the appropriate subroutine call. Late binding defers that decision until the last possible moment, namely to the time a function is actually called during the execution of a program. At that time, the decision which operation to apply to an object is based on the type of the object itself (see Figure 2.4). The exact mechanism by which this decision is arrived at depends strongly on the implementation of

the language, but is generally based on information generated by the compiler for each class or on information provided by the program environment.

2.3. An Introduction to C++

As an example of how the concepts discussed in this chapter can be implemented in an actual programming language, we will now give a brief introduction to the C++ language. There are two reasons for choosing to describe this language; first, it is the most widely used object-oriented programming language currently in use, and, second, the simulation system described in following chapters is based on C++, so familiarity with the language will help the reader later on.

C++ was invented by Bjarne Stroustrup and others at AT&T Bell Laboratories in the early 1980s [Stroustrup 1986]. The object-oriented features of C++ are mostly inspired by Simula 67 [Birtwistle 1973], the first object-oriented programming language. As its name implies, C++ is a superset and extension of the *C* programming language [Kernighan 1988]. It retains the emphasis on code efficiency and the permissive attitude towards the programmer which have made *C* the most widely used programming language for systems programming, while providing additional features to assist the programmer in catching mistakes and in structuring large programs. Historically, C++ has been implemented as a preprocessor producing *C* source code as its output; lately, true C++ compilers have appeared on the market which eliminate the additional compilation and open the door to true C++ programming environments.

The extensions to *C* provided by C++ fall into two categories: non-object-oriented and object-oriented. The former group for the most part introduces stricter rules for function declarations and type-checking to the language, an area in which *C* has been notoriously weak. Other additions include the overloading of function names, default arguments to functions, and a new comment delimiter. *Overloading*, as used in C++, allows the use of the same name for different, but presumably related, functions for the convenience of the programmer; the compiler selects the correct function to call according to the type and the number of parameters used in the function call.

A class declaration in C++ is similar to the declaration of a structure in *C*; the new keyword ***class*** is followed by the name of the class and a list of the class' member variables and functions enclosed in curly brackets. Access to an object's member variables and functions also follows *C*'s way of accessing a structure's fields. As a matter of fact, the C++ preprocessor implements classes using *C* structures. The example in Listing 2.1

defines a class *date* which illustrates several key features of C++; it was taken from pages 136 and 139 of [Stroustrup 1987].

```
class date {
int  day, month, year;
  public:
     void set(int, int, int);
     void get(int*, int*, int*);
     date(char*);
     date(int, int, int);
     ~date();
};
```

Listing 2.1: Example of C++ Class Declaration

According to this class declaration, an object of class ***date*** consists of three private member variables *day*, *month*, and *year* and two public member functions *set* and *get* to set and retrieve the date. The two declarations of the function *date* specify two different constructor functions, one of which (depending on the function arguments) is automatically called when an object of type *date* is created; *~date* is the name of the destructor function, called when the object is deleted. In general, constructor and destructor functions are specified using the class name and no return type, as shown above. The example also illustrates the use of function prototyping in C++, in which the types of all of a function's parameters as well as its return type have to be specified and are type-checked by the compiler.

A class declaration in C++ defines its private members and its interface to the outside world. Following the traditional *C* convention, the declaration is usually contained in a *header file*, while the implementation of the member functions of a class is kept in a separately compiled *source (code) file* which includes that header file. Thus it is possible to provide only the interface to a class, while keeping most of its implementation* hidden from the user by only providing the member functions in compiled form.

To improve efficiency and enable compatibility with regular *C* functions, C++ provides two special constructs not commonly found in other object-oriented languages, namely inline and friend functions. An <u>inline function</u> is a member function whose body is included with the class declaration; whenever such a function is called, the compiler substitutes the code for the body of the function for the function call and no function call takes place at runtime. While there are obvious possibilities for abuse of this facility, its

* Inline functions and macros are declared in the header file in C++, and are therefore visible to the client of a class.

intended and most frequent use is to implement access functions to private member variables, thus encouraging data encapsulation without compromising efficiency. A class declaration may list an external function as a *friend* and thereby give it access to its private members. This facility is necessary for working in a hybrid environment in which parts of a program may be written in C, but still need to have access to the private data of a C++ object. Friend functions are also useful for the implementation of functions which have to manipulate the private data of two or more classes in the same operation.

```
class base {    // by default, members are private
     int  a;    // hidden from all other classes
  protected:
     int  b;    // accessible only to derived classes
  public:
     void c(); // accessible to all classes
};
class der1 : base {
     int  x;
  public:
     // xa() is invalid, as a is private to base
     int  xa()  { x = a; }
     // xb() is a valid inline function
     int  xb()  { x = b; }
};
class der2 : public base {
     int  y;
  public:
     int  ya();       { y = a; }      // invalid
     int  yb();       { y = b; }      // valid
};
// a function outside of the class hierarchy
void f1(base* bp, der1* d1, der2* d2) {
     bp->b=1;   // invalid, b is protected
     bp->c();   // valid, c() is public to base
     d1->c();   // invalid, c() is private to der1
     d2->c();   // valid, c() is public to der2
}
```

Listing 2.2: Base Classes and Derived Classes

A class declares itself a derived class by listing the base class' name in its class declaration; it then has access to all public, but not private, member variables and functions of its base class. A base class can be declared *public*, in which case the public members of the base class are also public members of the derived class; by default, base classes are *private* and their public members become private members of the derived class. As a derived class often needs access to its base class' private members, a class can declare some of its members as *protected*, meaning they are accessible from any derived class, but private with respect to other classes and functions. Friend functions declared in a derived

class are subject to the same access restrictions as the derived class itself. The example in Listing 2.2. above illustrates these concepts.

C++ supports polymorphism and late binding through what it calls, as does Simula 67, virtual functions. By declaring a member function as *virtual*, a base class allows any class derived from it to redefine this function; by default, all other (non-virtual) functions are statically bound and may not be redefined. Any object of a class with virtual functions contains a table of function addresses with one entry for each function declared virtual; each entry in the table points to the implementation of the function appropriate to the class of the object. This hybrid model preserves the efficiency of a regular function call for all non-virtual member functions and altogether eliminates the space and time overhead entailed by virtual functions for classes which contain none. This contrasts with other object-oriented languages, such as Smalltalk, where all function calls are dynamically bound and thus incur the time penalty for late binding whether they make use of it or not. The disadvantage of the hybrid model is that a class designer must anticipate all possible future uses of a class as it is not possible to later redefine a function as virtual in a derived class to enable dynamic binding.

To complement its support of object-oriented programming, C++ also provides type conversion operators and the capability to overload its standard operator symbols, including those for array subscripting and function calls. The two new keywords *new* and *delete* are used to create and destroy class instances: they allocate and disallocate storage space for an object and call the appropriate constructor and destructor functions. Automatic type conversion and operator overloading enable the creation of user-defined classes whose objects can be manipulated with the same ease as variables of the standard built-in types. Expressions involving user-defined objects may thus be written using standard operator symbols and mixed expressions containing both user-defined objects and standard data types are made possible. Additionally, by taking control of storage allocation for a particular class, a designer may implement a storage management system optimized for that class. This may be useful, for example, for a class whose objects are frequently created and destroyed.

2.4. Benefits of Object-Oriented Programming

In summary, let us review the features that make object-oriented programming possible and how they relate to the goals of object-oriented system design. By concentrating the analysis of a system on the data and how it is manipulated, object-

oriented design leads to the identification of system components which can be described as self-contained entities, each consisting of a set of data items and of procedures to manipulate the data. Each system component thus identified is described as an object, an instance of a class. An object encapsulates its data and protects it from outside manipulation except through a defined interface of services; it also hides its internal structure and implementation details and thus reduces its vulnerability to outside interference.

Using inheritance, relationships and commonalities between classes can be made explicit in the class structure by building hierarchies of related classes. Additionally, inheritance and polymorphism enable the re-use and sharing of code between related classes and support the modification and refinement of an existing system design.

Polymorphism together with late binding allows a designer to concentrate on the actions to be performed by the system on a data object without regard to the type of the object at runtime. This allows flexibility in case of later changes or extensions to the system. Additionally, the effects of a change in a class specification can be limited to one place, namely the class itself, and do not necessitate changes to all places in a program where an object of that class has been used.

Object-oriented design and its implementation through classes thus leads to a modular system consisting of independent objects which communicate with each other through explicitly defined interfaces. Polymorphism and late binding together allow an object-oriented system to automatically adapt its operations to the objects it is manipulating. The designer of the system thus does not have to anticipate all possible objects, or combinations of objects, which may be manipulated by the system. This in turn facilitates future modification and expansion of the system and leads to truly flexible systems.

The high modularity of an object-oriented system and the tight encapsulation of its component objects from each other also has additional side benefits with regard to validation and distribution. Proper object-oriented design strives to isolate each class as much as possible from outside interference and to concentrate all of the data and procedures which impact objects of the class within the class itself. The design thus results in a number of self-contained modules, each of which may be tested in isolation from the others and from the remainder of the system. By limiting the number of ways an object can be manipulated, the number of possible test cases is also reduced; by making those ways explicit, the language itself assists in preventing accidental manipulation of an object.

A large part of the work involved in distributing a system across several computers at the medium and coarse grain level, namely the identification of self-contained components, has already been accomplished by an object-oriented design. As the interfaces between the system's components have also been specified already, distributing an object-oriented system should involve far less work than distributing a traditionally structured system.

3. The Modular Approach to Simulation

In this chapter, we give a brief introduction to discrete-event simulation and to the basic concepts involved with this form of simulation. We then outline the difficulties experienced with traditional simulation programs and contrast this to the advantages offered by modular simulation programs. Following that, we examine the close relationship between modular simulation and object-oriented programming. Finally, we discuss structuring techniques for simulation programs which are made possible by the modular approach.

3.1. Discrete Event Simulation

Discrete event simulation (DES), which in our case is synonymous with digital discrete event simulation as we are only interested in simulation programs executing on digital computers, is used to model systems which change state only at discrete points in time. Although DES can be adapted to model continuous systems, other simulation methods such as those based on differential equations are more appropriate for modeling them.

In DES terminology, an event describes a change (or changes) in the state of a system; by definition, an event occurs at a discrete instant in time and the state change described by the event occurs in a discrete step. A set of related operations which together transform the state of an entity within the simulated system is called an activity. An activity is triggered by an event and may result in the generation of one or more future events. All the events and activities relating to a particular entity in the system together form a process, which describes the progress of the entity through the system. Figure 3.1 below shows the relationship between these three concepts.

As changes in the simulated system's state can occur only at discrete moments in time, the simulation program can simply advance the simulation time to the time of the next event, i.e. to the time of the next change in the system's state. The simulation clock thus advances in irregular leaps and bounds, depending on the time intervals between the events occurring in a system. The *current simulation time* is always the time of the currently executed event.

Depending on the concept upon which the algorithm used to advance the simulation time is based, we can distinguish three approaches to modeling a DES system, namely the *event scheduling*, the *activity scanning*, and the *process interaction* approach [Fishman 1973]. In the event scheduling approach, as implemented, for example, in the GASP and

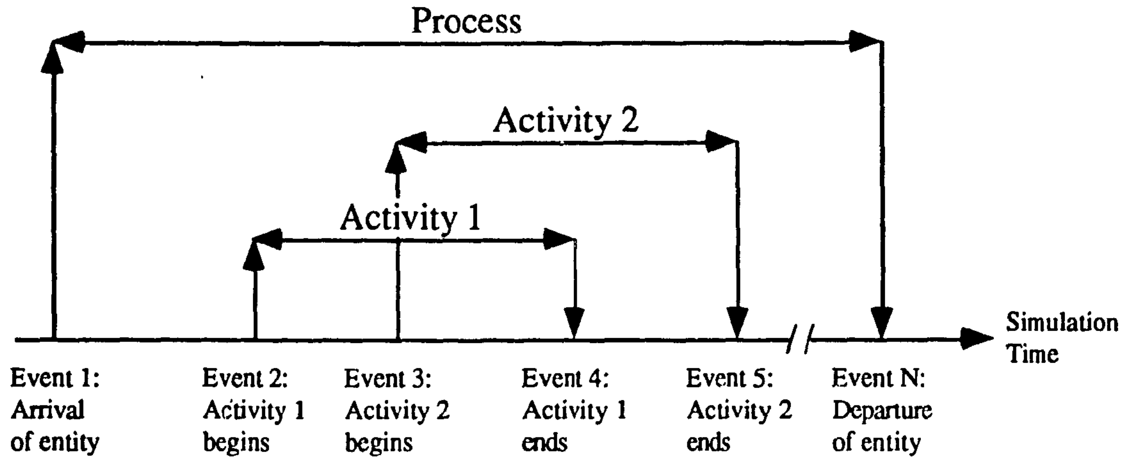

Figure 3.1: System Modeling Concepts: Event, Activity, Process

SIMSCRIPT simulation languages, a simulation program consists of an enumeration of all the events that are possible in the system and, for each event, a description of the actions that take place when that event occurs. Each event is described by a record containing the time and type of the event. As events are generated, they are added to a global event queue which is ordered according to the simulation time of each event; some (usually user-selectable) queueing discipline (e.g.. FIFO, LIFO, with or without priority) may be used to determine the order of events that have the same simulation time. The simulation control program selects the next event from the event queue and passes control to the sequence of statements which describes the actions for that type of event. Each such sequence ends with a "select next event" statement, which returns control to the simulation control program.

In contrast to event scheduling, the activity scanning approach does not have a global event list. Rather than placing the emphasis on the events that occur in a system, it views a system as a collection of activities, where each activity is a set of operations which transform the state of one of the system's components. Each system component whose state can change must have a clock associated with it which records the component's current time. Whenever an event occurs, the simulation control program reviews all activities and determines the ones that can be started or completed. It then scans the clocks of all components in the system to determine the time of the next event and advances the simulation time accordingly. The type of the next event is then determined by logical checking, i.e. by the operation which triggered the event.

The process interaction approach combines features of both the event scheduling and activity scanning approaches; it is used, for example, in the GPSS and Simula 67 simulation languages. This approach views the modeled system as a collection of separate - but related - processes, where each process describes the progress of an entity through the modeled system. A process consists of the sequence of actions which are performed between the creation of an entity (the arrival of the modeled entity in the system) and its destruction (the entity's departure from the system). The actions describe changes in the modeled entity's state, its interaction with other entities and the passage of time, i.e. the amount of time the entity is busy performing a task. Whenever a process needs to synchronize with another process or when it is busy, it is deactivated and an event record is generated which describes the conditions to be met for the re-activation of that process. This event record is then enqueued in the global event queue and the simulation control program selects the next record from the event queue and activates the associated process. In essence, this implements a multi-tasking system in which processes are scheduled by the simulation control program.

Each of the three techniques described above has its own advantages and disadvantages [Fishman 1973]; which one is the most appropriate for a given problem depends on the characteristics of the modeled system. An event scheduling simulation system is probably the easiest to implement due to its conceptual simplicity; however, as each activity in the system must start and end with an event, the number of event records that is generated, manipulated, and destroyed can become large and thus may result in a large system overhead. This is especially true for a system in which only a few arrivals occur but each arrival results in many activities. Activity scanning replaces event record manipulation with faster conditional checks at the completion of each activity. It would thus perform well in the situation outlined above, but would perform poorly in a system with many arrivals (and possibly no, or only few, activities associated with each arrival) because each occurring event requires the system to scan all activities to ensure that all state changes occur.

As process interaction can be regarded as a hybrid of the other two approaches, it shares some of the advantages and disadvantages of both. Depending on the synchronization conditions used for the re-activation of processes, it may exhibit the characteristics of an event scheduling system (e.g. all processes wait unconditionally for activation at a given time) or of an activity scanning system (e.g. processes are reactivated when the states of other processes meet a given condition). In the worst possible case, it may combine the

system overhead of an event list and the manipulation of many event records with the repeated scanning and testing of complex synchronization conditions between processes. In a more typical case, however, it may provide the best of both approaches by combining the simple concept of the event queue with the flexibility provided by conditional activation. At the same time, the number of events that are generated in the system is reduced as the activation and deactivation of processes is combined into a single event. Additionally, the process interaction approach provides greater conceptual simplicity than the other two approaches as it groups all activities and events pertaining to an entity into a single module rather than scatter them throughout the simulation program.

In spite of their different implementations, all three DES modeling techniques are based on two concepts: that of an event and that of a simulation clock advancing in discrete steps according to the sequence of events. A DES simulator-run is usually started by a triggering event or by a startup activity which generates the initial event. After its startup, the simulation continually generates and consumes events until either no events remain to be executed or until some termination criterion, such as a specified simulation time, is reached. At that time, the simulator will typically provide statistics on the number and types of events that occurred during the simulation-run and on the state of the system's queues.

3.2. Monolithic vs. Modular Simulation

Traditionally, simulation programs have been written as large stand-alone programs, where each program created corresponds closely to the system being modelled by it. Even when subroutines or complete programs are already available to simulate certain parts of a system, extensive programming effort and modifications are often required to customize the existing software and adapt it to capture the details of the system under study. The task of changing such a monolithic simulation program is made even harder for programs using the event scheduling or activity scanning techniques as the different events or activities related to a system component, in that case, are scattered throughout the program. Apart from the difficulties in the development and modification of a large simulation program, there are also considerable problems in testing it, especially with regard to verifying that the simulation truthfully captures the characteristics of the modeled system and delivers valid results.

It is desirable, and in accordance with accepted software engineering principles (see, for example, [Parnas 1972] and [Parnas 1985]), to break down a simulation program

into a number of modules where each module models one component of a system. To construct a simulation program for a specific system, one would then only have to combine the appropriate modules. This modular approach makes it possible to develop and test each simulation module in isolation from the others; moreover, by reducing the complexity of each module, it makes it much easier to verify that the simulation module's behavior corresponds to the behavior of the modeled subsystem and to test the validity of its output. Additionally, changes to a module are much easier to make as all relevant data and procedures are located within that module. Once such a simulation module has been constructed and validated, it can then be added to a *module library*; modules in the library can later be re-used in other simulations [Wade 1984].

Given these advantages of modular simulation, why is it not more widely used? One reason for this is that many simulation programs and packages are designed for one specific application (e.g. analyzing a particular network configuration) and their designers do not have to be concerned about future modifications. Another is that traditional simulation languages do not support modularity very well. Even if an attempt is made to design a modular simulation program using one of these languages, the interaction between modules makes it difficult to achieve a high degree of modularity, particularly given the lack of appropriate tools. Finally, to get all possible modules in a system to cooperate, it is necessary that they all conform to the same coding standards and interfaces. Unfortunately, any such standard introduces a certain amount of additional coding into the programming effort and may also involve some runtime overhead. It may thus be perceived as more of a hindrance than an advantage when designing a simulation program.

3.3. Object-Oriented Programming and Modular Simulation

Object-oriented programming and simulation share many basic concepts [see, for example, Eldredge 1990 and Larkin 1988]. As a matter of fact, the first programming language to introduce some of the core concepts of object-oriented programming, such as objects, classes and inheritance, was a simulation programming language, Simula 67 [Nygaard 1981, Birtwistle 1973]. The designers of that language, Ole-Johan Dahl and Kristen Nygaard, can be credited with single-handedly inventing object-oriented programming (even though they didn't know it at the time). Simula terminology and constructs have influenced a number of later object-oriented languages, most notably C++.

Looking closer at object-oriented and simulation concepts, the following commonalities become apparent. Both regard a system as consisting of related components

or subsystems; both assign a state to these components and associate a set of functions with them to change their state and to perform activities [Bezivin 1987]. In object-oriented design the central problem is to identify the objects in a system, whereas in simulation it is to determine the components which form the modeled system. Both fields, for different purposes, attempt to break a given system into components and then model the components so identified. Not surprisingly, even though the two fields may use different terminologies, they are really dealing with very similar concepts.

Considering these similarities, it is quite natural to represent the components of a system as objects which respond to a certain set of messages, i.e. events. Especially the process interaction approach to DES modeling fits naturally with the object-oriented paradigm as it already combines the state variables that describe an entity and the activities performed by it into a process. The only element missing for an object to implement a process is a multi-tasking facility; this is actually provided in Simula 67 by the *coroutine* concept and the associated *detach* and *resume* instructions.

In addition, as will be described in more detail later on (see chapter six), object-oriented methodology, in particular inheritance, provides a transparent way to build reusable simulation modules. For example, a class can be designed which provides all the necessary code and declarations to implement a standardized interface through which modules are to be connected. This class provides functions for connecting modules and for communication between modules; the implementation of the interface is hidden within the class and inaccessible to its user. However, by declaring all modules in a simulation as descendants of this base class, the modules inherit the interface and the communications procedures. The designer of a module can then concentrate on the task of creating a simulation of a system component and does not have to concern himself with the interconnection mechanism.

3.4. Composite Models and Model Hierarchies

As we have seen, a DES model is described by its state, as given by its internal state variables, a set of input events which, when they occur, can change the current state of the model and trigger certain activities, and a set of output events, which are generated by state changes and/or input events. Additionally, a model is described by its current simulation time, whether that time is represented explicitly as a clock, as in the activity scanning approach, or whether it is given implicitly by the time of the most recently processed event. A formal way of describing such models was introduced by Bernard Zeigler

[Zeigler 1984a] with the DEVS formalism. According to this formalism, a model M is described as a seven-tuple $<X, S, Y, \delta_{int}, \delta_{ext}, \lambda, \tau>$, where X, Y, and S are the set of external input events, external output events, and model states; δ_{int} and δ_{ext} are internal and external state transition functions; λ is an output event generator function, and τ is the model's simulation time advancement function. We will not discuss the DEVS formalism here in any further detail, but we note that a model can be characterized as a self-contained entity whose sole interaction with its environment is through input and output events, while the model's inner working is hidden from its user.

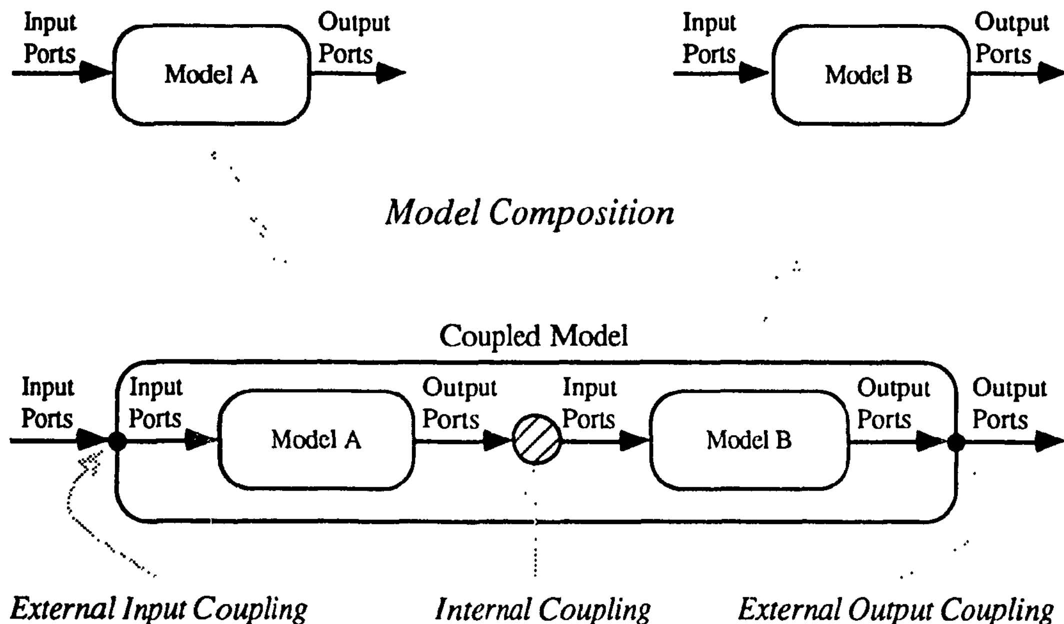

Figure 3.2: Model Composition

This characterization leads us directly to viewing a model as possessing a number of logical input ports through which all input events are received, and a number of logical output ports through which all output events are sent [Zeigler 1984b]. Given this view of a simulation model, there is nothing to keep us from connecting the output ports of one model to the input ports of another. In fact, the coupled model [Ören 1984], constructed from the composition of models, can again be described as having a set of input and output ports and some hidden inner mechanism. In other words, the coupled model can itself be treated like an atomic model, i.e. a model which cannot be decomposed any further, and may be used together with other models to build an even larger model. When we build a coupled model from two (or more) components, we also have to specify a coupling scheme, i.e. a mapping between the input and output ports of the models. Depending on

the ports connected, we can distinguish three types of couplings: the external input coupling and the external output coupling, which connect the ports of the coupled model to the ports of the model's components, and the internal coupling, which specifies how to connect model ports within the coupled model. Figure 3.2 above illustrates these concepts.

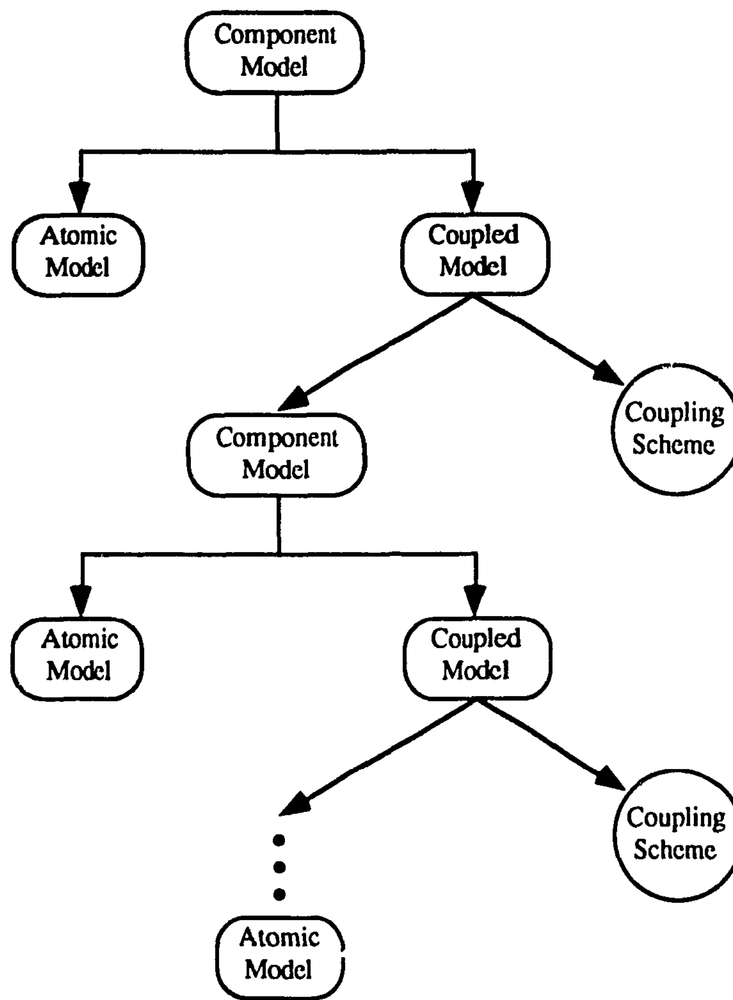

Figure 3.3: Model Hierarchy

As described in the previous paragraph, each component of a model is itself a model, either atomic or coupled. The model coupling process is therefore recursive and naturally leads to a hierarchical decomposition of simulation models [Zeigler 1987]. At the highest level, a simulation can be regarded as one large coupled model which consists of a number of component models and a coupling scheme. Each component model can then be similarly decomposed until only atomic models are left. As illustrated in Figure 3.3, we

thus arrive at a hierarchy of interconnected models which together form a simulation of the chosen system.

By storing atomic and coupled models in a *model database*, we can build a simulation from existing components. Only those models which are not yet available in the model database have to be newly developed. An additional benefit of being able to connect different models regardless of their implementation is that we are able to switch simulation models with the same external interface but a different internal implementation to explore alternative configurations of the modeled system, essentially without any programming effort. Consider, for example, the simulation of a computer network where each node in the network is represented by a simulation model of the appropriate computer type and the network itself is modeled by another model connected to all of the computer models. Provided that all software modules used in the simulation follow the same interface guidelines and provide compatible input and output ports, we can then exchange one type of network for another, e.g. a CSMA network for a token-ring network, by simply switching the appropriate simulation modules.

3.5. Modeling at Varying Levels of Detail

In a hierarchical modeling environment, in which simulation models are selected from a model database and coupled together, a simulation is described by the hierarchy of its connected component models. We can view this hierarchy, or composition tree, as one dimension along which to characterize a simulation; it is mainly concerned with the connections between models and treats the models themselves as atomic entities. Another dimension which can be used to describe a simulation is concerned with the models themselves; it tries to capture the modelling details captured in each model [Bharath 1984]. Up to this point, it was our tacit assumption that all modules used in a simulation program encode more or less the same degree of details. However, this assumption need not be true.

As the level of detail that is included in a model increases, so does the model's complexity and with it the time needed to design, program, and debug the model [Yeh 1979]. Additionally, increased module complexity brings with it increased execution time, which directly affects the duration, and cost, of a simulation-run. It may thus be desirable to combine in a simulation program, modules which model different parts of a system at differing levels of detail. Such an approach may be used during the initial development of a simulation program by first creating a rough model of a system and then progressively increasing the detail of each module [Yeh 1979]. This has a direct analogy in the technique

of stepwise refinement which is well known in software engineering [see, for example, Sommerville 1982 or Fairly 1985]. Another possible use for modules with different levels of detail in the same simulation is to speed up the overall simulation by modeling certain components of the system at a higher degree of detail where detailed answers are needed, and modeling all other components at a coarser level of detail. As an example, consider again the simulation of a computer network. If we are interested in the behavior of the network itself, but not in simulating the processing that occurs at each node, we might use a simple processor module which acts as a transaction generator for the network. Conversely, if we are interested in the detailed operation of each node, we may use a highly accurate model of the node, possibly consisting of several component models, which describes how transactions are generated, while using a less detailed model of the network.

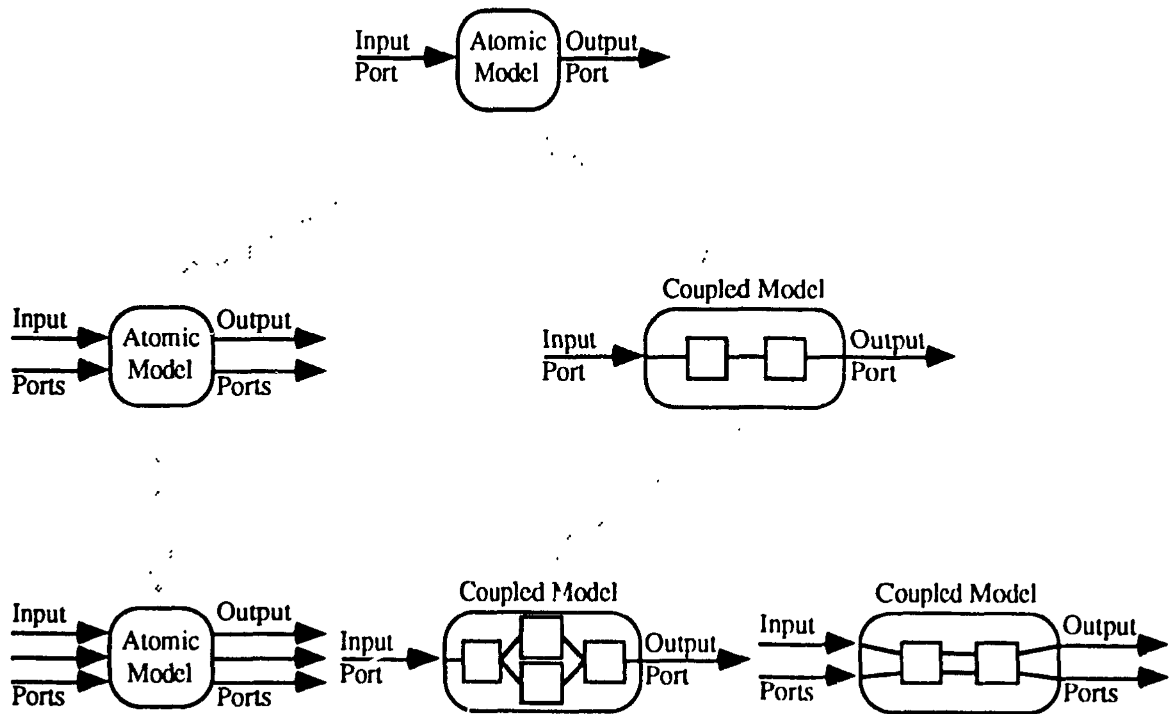

Figure 3.4: Levels of Detail

There are two ways to increase a model's level of detail: we can expand the model to include more details of the modeled subsystem, or we can create a new model consisting of several components. Intuitively, one feels that there should be some relationship between the level of detail in a coupled model and the number of component models it consists of. If all component models are of a similar level of detail, a coupled model of a particular subsystem is clearly more detailed than an atomic model of the same subsystem.

While this is not generally true if models with varying levels of detail are mixed in the same simulation, the number of component models in a coupled model can be regarded as a measure of the model's level of detail. Some possible ways of increasing a model's level of detail are depicted in Figure 3.4.

For an atomic model, the effect of increasing the model's level of detail is to increase the number of states the model can be in. This increase in the number of states not only means that there is an increased number of possible transitions between states, but also that there may be more types of input events that the model has to be able to respond to as some of the transitions may only be triggered by new event types. Thus, the number of input ports of a model, and most likely also its number of output ports, can increase with an increase in the level of detail.

For a coupled model, an increased level of detail may be reflected in an increased number of component models without a change in the model's external interface. Alternatively, the level of detail of some or all of the component models may increase, leading to a larger number of internal connections and most likely also to an increase in the number of the model's ports. A simulation system which permits the combination of models at different levels of detail must therefore have a way to deal with the varying interfaces between modules as models of different levels of detail are exchanged for each other.

4. An Introduction to SAMOC

Previously, we have discussed object-oriented design and have examined C++ as an example of an object-oriented programming language. We then examined the terminology and concepts used in discrete event simulation. In this chapter, we combine those two topics and introduce a discrete-event simulation package written in C++. This package provides a good example of the synergy between the two concepts and it forms the basis for the implementation of the simulation system described in the following chapters.

SAMOC, an acronym which stands for Simulation And Modeling On C++, is a collection of C++ objects, data structures and routines which implement a process-based discrete-event simulation system [SAMOC 1988]. Most of the basic constructs used in SAMOC originated with an earlier package called DEMOS, short for Discrete Event Modeling On Simula, which implemented a DES system for Simula 67 [Birtwistle 1979]. Some additional Simula 67 constructs, most notably coroutines, have also been re-implemented in SAMOC using C++.

Coroutines as implemented in SAMOC, and Simula 67, are a form of cooperative multitasking. In contrast to preemptive multitasking systems, a cooperative multitasking system does not employ a scheduler which interrupts an executing process once its time-slice has expired. Instead, it relies on the currently executing process to voluntarily relinquish the processor and system resources and suspend itself. A cooperating multitasking system may employ a scheduling algorithm to determine which process to transfer control to next, or it may leave that decision up to the process relinquishing control of the system.

4.1. Entities

As does Simula, SAMOC implements a DES system using the process interaction approach discussed in the previous chapter. Processes, therefore, are the main simulation components; each process has a number of attributes and actions. A process is called an entity in SAMOC and is implemented as an instance of the base class ***entity*** or of one of its derived classes. Class ***entity*** implements the basic functions needed for scheduling and manipulation of entities and for event list management. It also declares various other needed attributes, such as an entity's current simulation time or its current state, and functions to access them (see Listing 4.1 for an abridged definition of class ***entity***).

```
class entity : public task {
// Note: class task implements coroutines
// Private attributes and member functions omitted
public:
  entity();                  // constructor, creates coroutine
  ~entity();                 // destructor,  deletes coroutine
// entity attributes and state
  int priority();            // entity's priority  level
  int interrupted();         // entity's interrupt level
  sim_time evtime();         // entity's simulation time
  boolean idle();            // true if entity idle
  boolean active();          // true if entity active
  boolean terminated();      // true if entity terminated
  void terminate();          // terminate entity
  void deletable();          // mark entity as deletable
// queue manipulation functions
  void out();                // remove entity from queue
  void into(queue *q);       // enter  entity into queue
  entity *succ();            // successor   in queue
  entity *pred();            // predecessor in queue
  entity *nextev();          // next entity on event list
  queue *currentq();         // current queue entity is in
  sim_time time_in();        // time queue was entered
// scheduling functions; other scheduling functions
// declared globally and made friends of class entity
  void cancel();                  // cancel entity
  void interrupt(int n);          // interrupt at level n
  void schedule(sim_time t,       // schedule entity at t
              delay_code dc = DELAY_TIME,
              schedule_code sc = IN_ORDER);
// entity synchronization functions
  int wants();               // number of units requested
  void coopt();              // acquire ownership of entity
  entity *owner();           // entity's owner (if any)
  boolean avail();           // true if entity unowned };
```

Listing 4.1: Declaration of Class ***entity***

Entities are the active elements in a SAMOC simulation program; they are given full "life histories" [Birtwistle 1979] in the form of a *script*, or step-by-step listing, of all the actions to be performed by them during their lifetime. User-defined entities are declared as extensions, using inheritance, of the base class ***entity***; they implement the necessary additional state variables and functions not provided by class ***entity***. C++ allows the transparent inclusion of the needed coroutine mechanism by inheritance and, due to dynamic binding, allows SAMOC routines to treat all instances of derived classes as instances of class ***entity***. Thus it is possible for SAMOC to provide services which can be used by all entities, regardless of their implementation details.

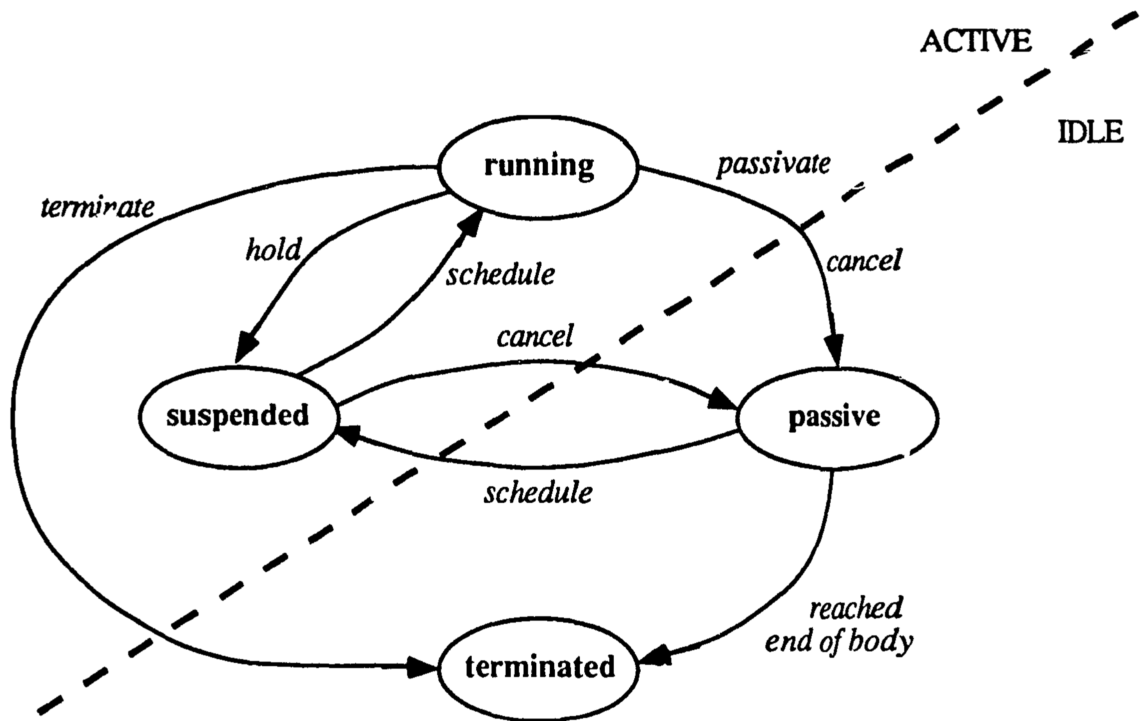

Figure 4.1: Entity State Diagram

During its lifetime, an ***entity*** object is always in one of four possible states: *running*, *suspended*, *passive*, or *terminated* (see Figure 4.1). When an entity is created, its constructor will set up its runtime environment (including allocation of a local stack area), perform any necessary initializations of its member variables, and leave the entity in its passive state. It is up to the entity's creator, the currently running entity, to activate the new entity at the desired simulation time by calling its member function *schedule* with the appropriate parameters. Scheduling an entity will enqueue it onto the event list at a given time and move it to the suspended state. Once the entity reaches the head of the event list, it becomes the running entity and starts executing the actions defined in its member function *body*. Typically, an entity's *body* consists of a large loop statement which terminates when the entity's lifetime is over. The running entity can relinquish control by scheduling a suspended entity (suspending itself in turn), or by passivating itself; it can also cancel previously scheduled entities, moving them to the passive state. When an entity exhausts its actions or voluntarily terminates, it enters the terminated state from which it can never return. SAMOC does not automatically delete a terminated entity (unless flagged accordingly), as it may still hold state information or statistical data which may have to be accessed later on.

An entity's actions roughly fall into one of two categories: the simulation of its own activities and the manipulation of other entities. An entity's activities are reflected by changes in the entity's internal state; to simulate the duration of an activity, an entity suspends itself for the required amount of time, using SAMOC's **hold** function. The time spent in the suspended state represents a period during which an entity remains in the same state and occupies the same resources.

As a result of its activities, an entity may also create and manipulate other entities. One typical example is the creation of the next occurring instance of its own type by the current entity. This is used for instances of entities which repeatedly enter the simulated system, such as jobs entering a queueing system. When an instance of such an entity type first starts executing, it will create its successor and schedule its activation at a time determined by sampling the appropriate distribution. This technique of dynamic entity creation and scheduling allows to keep only one - the next - instance of the entity on the event list which has not yet "officially" entered the system and thus reduces overall system overhead by keeping the size of the event list small. An entity may also manipulate other existing entities by interrupting their current activities, rescheduling them at an earlier or later time, or passivating them by canceling and removing them from the event list.

Any change in an entity's state is effected through one of the scheduling functions provided by SAMOC. Calling one of the scheduling functions indicates the calling entity's willingness to relinquish control. After changing the calling entity's state as determined by the particular function and inserting it into the event list based on the time of its next event, the SAMOC scheduler selects the entity at the head of the event list as the next running entity and activates it at the point where it last relinquished control. SAMOC therefore implements a cooperative multitasking system with a central, non-preemptive scheduler.

4.2. Queues

All currently active entities in a SAMOC program are kept on the system event list, in ascending order of the simulation time of their next event. The event list, while usually not directly manipulated by the user, is a specialized instance of SAMOC's more general queue class which is provided for user-defined entities. Objects of class ***queue*** implement a queue which holds entities, i.e. objects of class ***entity*** and of those classes derived from it, in order of their simulation time. Each entity has a standard attribute *Priority*, initially zero, whose meaning is user-defined. Within each group of entities with the same simulation time, objects of higher priority are enqueued in front of all objects with lower

priority, regardless of their arrival time in the queue. For entities of the same priority, or for all entities if the *Priority* attribute is unused, ***queue*** implements a simple FIFO queue.

Entities in a SAMOC simulation program frequently have to keep track of other entities, for example in the case of a master entity with several subordinated idle entities (i.e. entities not currently on the event list). Class ***queue*** provides a uniform mechanism to accomplish this. Also, as ***queue*** is intimately linked with ***entity***, it makes sense to provide it as part of the simulation package, rather than putting the burden on the user to implement a general queueing class.

The classes ***queue*** and ***entity*** together also provide an interesting example of the kind of change in design perspective that object-oriented programming can bring about. Class ***queue*** by itself provides only member functions to determine the first and last entities in the queue and the current length of the queue, plus some reporting and statistics functions which are themselves inherited from yet another class, ***tabulate*** (see Listing 4.2). All the other functions normally associated with a queue, such as entering objects into and deleting objects from the queue and determining an object's predecessor and successor, are in fact defined in class ***entity*** (see Listing 4.1).

```
class queue : public tabulate {
  // Note: class tabulate implements report functions
  // Private attributes and member functions omitted
public:   queue(char *title, boolean report = TRUE);
// Queue manipulation
  int length();               // current length of queue
  entity *last();             // last  entity in queue
  entity *first();            // first entity in queue
// Reporting functions
  void list();                // list entities in queue
  void reset();               // reset  queue statistics
  void report();              // report queue statistics
  char *heading();            // heading for statistics
};
```

Listing 4.2: Declaration of Class ***queue***

The design perspective adopted here is that what is manipulated is not the queue itself, which is treated as the passive object in the transaction, but the ***entity*** object. Hence, it is the entity which is enqueued and dequeued, and it is the entity which determines its predecessor or successor, not the queue. Naturally, it would not be a good design decision to implement a general queue type in the above described manner because then each object that could possibly be entered into a queue would have to implement the necessary queueing functions. However, in a system such as that provided by SAMOC,

with its specialized queue type and where an entity is expected to spend most of its time entered into some queue, this design is advantageous. It then also makes sense to have an entity provide a pointer to its current queue and the (simulation) time it entered into the queue. The latter makes it very easy to generate statistics regarding an entity's wait times in queues.

4.3. Synchronization Objects

Class ***queue*** is used to model the simplest case of synchronization between entities, that of client entities waiting in line until some server entity processes their request. Often, however, we need to model more specialized cases of synchronization. For these cases, SAMOC provides three classes of special-purpose synchronization objects, namely the classes ***bin***, ***res***, and ***waitq***. As these three classes are implemented as regular C++ objects, generating and manipulating a synchronization object involves far fewer activities than performing the equivalent function on an ***entity***, for which the system has to set up its own runtime environment. Modeling entity interaction using a synchronization object thus results in a much smaller simulation system overhead than modeling the same interaction with an entity. Synchronization objects are therefore the preferred way to "model minor simulation elements where we don't need to give full rôle descriptions, but essentially record how much of a modelled resource is currently available." [Birtwistle 1979].

All three classes of synchronization objects are derived from class ***queue*** and manage their private queue of entities. They therefore inherit all their queue manipulation functions (*length, first, last*) and reporting functions (*reset, list, report* etc.) from class ***queue***. Entities in a synchronization object's queue are blocked waiting for a resource to become available; during the time they are enqueued, entities are passive (not on the event list). Once the awaited resource is available, the synchronization object schedules those waiting entities, whose requests can now be fulfilled, at the current simulation time.

Objects of class ***bin*** (short for storage bin) are used to model systems in which a producer/consumer relationship exists between two groups of entities. In such a system, one group of entities produces a needed resource and entities in the other group consume that resource. To synchronize the consumer and producer entities, the producers deposit one or more tokens representing resource units into a common storage area (the "bin") whenever they finish a production cycle; consumer entities then request a number of those tokens from the storage.

Instances of ***bin*** are created holding a certain initial number of tokens, as determined by their creator. Producers add more tokens to a ***bin*** using the member function *give*, while consumer entities request tokens using the function *take*. As implemented in SAMOC, there is no upper limit to the number of tokens in the bin, so producer entities are never blocked. Consumer entities, on the other hand, are blocked when they request a greater number of tokens than is currently available. Waiting consumers are unblocked once the requested number of tokens is available.

Res (short for resource) objects model the competition between entities for a limited pool of resources. They are created with a certain limited number of resource items available and synchronize mutually exclusive access to these resources between the entities contending for them. An entity requests use of a number of units of the resource by calling the member function *acquire*; if the attempt fails because an insufficient number of units are available, the requesting entity is blocked. Once an entity has finished using units of the resource, it returns them to the pool using *release*, which may unblock any waiting entities. As the maximum number of units of the resource it represents is fixed at its creation, a ***res*** object verifies that neither the number of units an entity attempts to acquire, nor the total number of units in the pool after an entity releases its units exceeds the specified maximum number of units. If either case is detected, a simulation runtime error is generated which identifies the offending entity.

Other than in producer/consumer systems, where different entities deposit and remove tokens from the shared pool, the same entity which requested a resource is expected to return it again to the resource pool after use. This conceptual difference can be used by a simulation designer to decide how to model a synchronization object in cases which may be modeled using either of the two classes. Class ***res***, however, makes no attempt to ensure that resource units are only released by the same entity which acquired them. Table 4.1 lists some of the commonalities and differences between the two classes.

Both ***bin*** and ***res*** classes also provide member functions *avail* and *obtainable* which allow an entity to determine whether it would be blocked if it requested a certain number of units of the resource. This allows an entity some flexibility in its resource usage pattern. A call to *avail* returns the number of resource units currently available for use; *obtainable* returns true if the calling entity can acquire the specified number of units. The difference between the two functions is that the result returned by *avail* is the same for all callers, while *obtainable* takes into account the calling entity's priority and those of the entities blocked in the synchronization object's queue. Only if the caller's priority is greater

Feature	***bin***	***res***
Synchronization type	producer/consumer	mutual exclusion
Initial number of units	>= 0	> 0
Maximum number of units	unlimited	fixed at creation
Number of units generated during program execution	unlimited	none
Runtime error on request if...	$m^* < 1$	m < 1 *or* m > limit
Runtime error on rclease if...	m < 1	m < 1 *or* m+available > limit

Table 4.1: Comparison of ***bin*** and ***res*** Classes

than that of all blocked entities or if the queue is empty will *obtainable* return a true result; this reflects the fact that even though a sufficient number of units may be available, the entity may not be able to acquire them because other entities may get to reserve them first.

Waitq objects are used for direct synchronization between two or more cooperating entities. It is very difficult to implement a general method for a number of cooperating entities to not only perform the same task at the same time, but to also ensure that they advance their simulation time synchronously. Hence, SAMOC provides the ***waitq*** synchronization method which assumes a master/slave relationship between cooperating entities. To quote Graham Birtwistle, the designer of SAMOC (and before that of DEMOS), *"we arrange for one of the entities to dominate and let it treat the other as a resource to be coopted, retained as a passive slave throughout the period of cooperation, and then be released for independent progress at the end of this period of cooperation."* [Birtwistle 1979]

Consequently, a ***waitq*** object is implemented using two queues, one for master and one for slave entities. An entity assumes the rôle of a slave by enqueueing itself into the slave queue using the member function *wait*. Similarly, master entities acquire a slave entity by calling *coopt*, which will return a pointer to the entity at the head of the slave queue. If no entity is available from the slave queue, master entities are blocked and added

---

* *m* stands for the number of units requested or released

to the master queue. Appearance of an available slave entity (or of a master in the case of a blocked slave) will activate the master entity and coopt the slave entity for the master. The *wait* and *coopt* functions therefore work together to synchronize pairs of master and slave entities. All other functions, such as those used for determining the availability of slaves, are already implemented in the regular class ***queue***. To make these functions available to the users of ***waitq*** objects, member functions *master_queue* and *slave_queue* are provided which return pointers to the object's master and slave queues.

In the case of more than two cooperating entities, we must allow one master to adopt the required number of slave entities. As ***waitq*** allows a master to coopt only one slave at a time, the master must enqueue itself for a number of times until it has acquired the required number of slaves. Unless this master entity has a higher priority than all other masters, it cannot make any assumptions about being able to adopt all of the needed slaves without interference from another master entity. A design with several masters needing more than one slave at a time therefore has to be carefully evaluated by the simulation designer as it has the potential for deadlocks.

Finally, SAMOC also provides a member function *coopt* in each ***entity*** object which allows an entity to directly gain control of other entity. *Coopt* performs the same actions for an entity as it does for the ***waitq*** member function of the same name, namely it removes the entity from any queue it is on and sets the entity's owner to the calling entity (actually, ***waitq***'s *coopt* is implemented using ***entity***'s *coopt*). Using *coopt* and the member function *owner*, which returns an entity's current owner (if any), any number of user-defined synchronization schemes can be implemented.

4.4. Statistics and Report Generation

As mentioned earlier, all ***queue*** objects, and, by inheritance, the queues in all ***bin***, ***res***, and ***waitq*** objects, collect some basic usage statistics. Additionally, SAMOC provides five data collection classes which can be used to collect user-defined statistics. All data collecting classes, explicit as well as implicit (such as in ***queue***), are derived from the base class ***tabulate*** which implements the shared functionality for report generation.

The member functions implemented by ***tabulate*** are *title*, *heading*, *reset*, *reset_at*, and *report*. *Title* and *heading* return the name of the data collection object (set at its creation) and the heading used in reports for objects of the respective type. *Reset* allows the user to reset the statistics for an object to an initial state; *reset_at* returns the last simulation time at which the object was reset (initially the time of the object's creation).

Finally, *report* generates a one-line summary of the statistics collected by an object. Functions such as *report* and *reset* are declared as virtual functions to allow their redefinition in derived classes as needed.

Resetting an object's statistics is useful as most simulations undergo a startup phase before the simulated system reaches an equilibrium. Statistics gathered during the startup phase normally would distort the results of the simulation [Law 1983, Wilson 1978]. By resetting statistics collection once the system reaches equilibrium, more realistic results can be obtained.

The five data collection classes provided by SAMOC fall into two categories: three simple collection classes and two classes which use the facilities provided by the simpler classes to analyze the collected data. All data collection objects can be given a name at their creation which will be printed on any reports; additionally, a flag *Report* may be set to suppress report generation for an object (by default, all objects generate a final report). This suppression facility is useful if we want to generate our own statistics printout and do not want it cluttered by system-generated reports. To record an observation, all data collection objects provide an overloaded member function *update* (*update* is an overloaded, rather than a virtual function, as the different classes require different numbers and types of arguments).

The simplest data collection class is ***count***, which just keeps a running total of recorded incidences of some user-defined event. Each observation updates the total by the number of incidences recorded during the observation period. ***Tally*** objects are used to record time-independent data sequences. In addition to keeping a running total of observed values, they also record the number of observations and the minimum, maximum, average, and standard deviation of the observed values. Creating similar statistics to ***tally*** (except for the running total), ***accum*** objects are used to record time-dependent data. They are updated with a value pair <t,v> to indicate that the value v was observed for t time units.

Class ***histo*** collects time-independent data using a ***tally*** object, but also generates a histogram of the collected data in addition to the standard report generated by ***tally***. The upper and lower limits and number of cells in the histogram have to be specified at the time the ***histo*** object is created; *update* is the same as for ***tally***. The histogram report shows the boundary values for each cell, the number of observations in each cell, and their relative and cumulative frequencies; the number of observations is also displayed graphically in the form of a bar chart.

Regre objects are used to perform a linear regression analysis on a sequence of (x,y) data pairs, i.e. to determine the straight line which best fits (in a least-squares sense) the recorded data points. If the analysis is successful, the report will display the number of observations, the average values for the x and y values, the slope and y-intercept for the regression line, and the correlation coefficient and some other statistics concerning the quality of the fit. An attempt is made to discover invalid or insufficient data; as such, at least six data pairs have to be recorded for an analysis to be attempted and degenerate data (such as values clustered around a point) are reported.

4.5. Distribution Objects

As any other simulation package, SAMOC provides a number of random number generators which produce pseudo-random numbers distributed according to a particular distribution. In most other simulation packages, the random number generator for each distribution is shared globally by users throughout the simulation system. To prevent predictable patterns in the generator's output, its users are often required to maintain different seed values to be used for generating different random number "streams". This method forces the users of the random number generator to explicitly update and manipulate the current random number seed, two operations which tend to be error-prone [Kleijnen 1986]. It also leaves the choice of the initial seed value for each stream up to the user, who usually selects a statistically unsound value. This results in short cycles and noticeable patterns in the random number generator's output (see [Park 1988]).

By employing object-oriented principles, SAMOC's designers have provided a random number generator which is both statistically sound and simple to use. It is also flexible enough to allow a user to manipulate the seed values, if he so desires. Random number generators are implemented as *distribution objects*, each of which is automatically given a seed value at creation and produces a stream of random numbers from a particular fixed distribution.

The base class for all distribution objects is class ***dist***, which uses a global "seed generator", separate from the random number generator itself, to initialize each distribution object's seed value. Each distribution object then uses a global random number generator routine with known good statistical properties, supplying its own current seed as a parameter, to obtain a value from a standard uniform distribution. The value so obtained is then transformed so that the result conforms to the desired distribution. Class ***dist*** also implements all functions common to all distribution objects, such as those used to set the

current seed value and to generate a report on the number of uses of the distribution object. A public attribute of dist, *Antithetic*, is also common to all distribution objects and is used by the global random number generator to generate antithetic, or complemented, values. For example, for a standard uniform generator (i.e. on the interval [0..1)), the antithetic random number will be 1-x (where x is the random number normally generated). This facility is useful to prevent any patterns in the generated numbers from biasing the results of a simulation.

Derived from class ***dist*** are the ten distribution classes provided by SAMOC. Each class implements a member function *sample* which returns the next random number according to the distribution implemented by the class. There are three groups of distribution objects, those that generate boolean, integer, and floating point values. In each of these groups is a class (***b_const***, ***i_const***, and ***f_const***) instances of which will return a fixed value determined at their creation. These classes are useful during simulation development when non-repeatable random number sequences would make debugging more difficult. Once the correct coding of a simulation has been verified, the constant generators can easily be replaced by the real random number generators.

Distribution objects of class ***draw*** return a **true** value with a certain probability which was determined when each object was created. Similarly, instances of classes ***randint*** and ***uniform*** return integer and floating point numbers which are uniformly distributed over an interval determined at their creation. ***Poisson*** objects return Poisson-distributed integers with mean λ, where λ is the expected number of arrivals (e.g. in a queue) per unit time. ***Normal***, ***negexp***, and ***erlang*** objects return floating-point values from normal, negative exponential, and Erlang distributions, respectively.

4.6. Other SAMOC Facilities

With the plethora of functions provided by a package like SAMOC, it is essential that as much functionality as possible is automated "behind the scenes", to reduce the burden on the programmer. This is especially important for casual users and for beginners. In this respect, SAMOC's implementation in an object-oriented language goes a long way to reduce its complexity and make life easier for the programmer.

By deriving all of the user-accessible classes, except for class ***entity***, from a single base class, ***tabulate***, SAMOC's designers have provided a consistent interface to these classes (Figure 4.2 shows a diagram of the class hierarchy). Additional commonalities between groups of classes could also be expressed using sub-hierarchies such as those

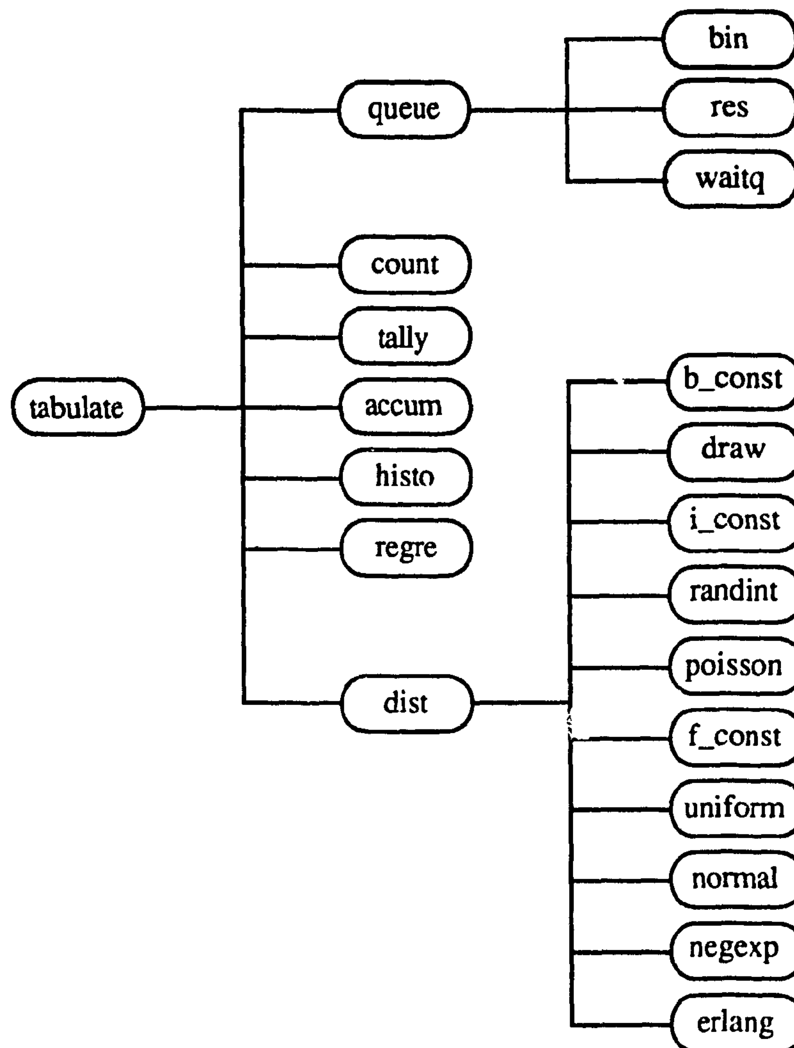

Figure 4.2: SAMOC Class Hierarchy

based on ***dist*** and ***queue***. By using constructors to initialize class members and by restricting the interface to a set of selected functions, much of the internal complexity of the system is hidden from the user, and remains hidden until the user explicitly requests access to the internal mechanisms. Moreover, judicious use of default values for member function arguments reduces the complexity of most function calls and sets values which make the objects behave the way they are expected to under most circumstances.

To further reduce the details a programmer has to worry about, all objects are linked into one of SAMOC's global report queues according to type. All objects on report queues (implemented as instances of class ***report_q***) automatically generate their default report upon termination of the simulation, unless explicitly overridden by the programmer.

Additionally, using global functions **report** and **reset**, all objects on report queues can generate a report, and can be reset under control of the simulation program. The programmer can tell the system not to add a newly created object to a report queue; by default, all objects are part of a report queue, ensuring that only a minimum effort is required to create a simulation which produces useful output. Custom and semi-custom report generation is also well supported by access to the headings used for reports for each class and by a variety of utility functions provided for printing and formatting.

Finally, SAMOC supports the debugging of simulation systems through a ten-level trace facility and various error and warning messages. Traces can be turned on and off individually for each level; five of the trace levels are defined by the system and can be used to selectively trace the operation of entities, queues, and synchronization objects. The remaining five trace levels are user-defined and can be used to implement higher-level diagnostic routines. SAMOC also performs various consistency checks at critical locations in the system code to detect inconsistencies which might have been generated by incorrect usage of system calls.

5. Interactive Generation of the Simulator

In chapter three, we gave an introduction to discrete-event simulation (DES) and explored the advantages of modular simulation programs. Based on this earlier discussion, we introduce in this chapter a way of providing simulation modules as building blocks for a simulator. We also consider the different types of users who may be employing a simulator and we describe the development cycle of a simulator. Finally, we describe a tool for the interactive generation of a simulator based on simulation module libraries.

5.1. Models, Modules and Module Libraries

Earlier we introduced a DES model as an abstract representation of a system or subsystem. As explained in chapter three, we can divide the actions such a model performs into receiving events through its input ports, sending events through its output ports, and performing some internal operations which consume input and generate output events and alter the model's state. To distinguish between an abstract model and its implementation in a simulation programming language, we call the implementation of a model a module. Modules are combined into module libraries to form the foundation of a modular simulation system.

5.1.1. Simulation Modules

As a module is the direct implementation of an abstract model, it is characterized by the same features as the model, namely by its input ports, its output ports, and a set of internal state variables and state transition functions. As with the abstract model, input and output ports, or more specifically, event input ports and event output ports, serve to connect the various modules which are part of the same simulator. Each port of a module has a certain event type associated with it and a port of a given type can only receive or send events of that type. Typed ports are necessary for two reasons: to specify the events which are expected as inputs to the module and which are generated by the module, and to allow module users to specify meaningful connections between the modules of a simulator. Without typed ports there is no efficient way to ensure that the correct ports are connected, i.e. that a module receives only the events it was designed to process.

In addition to the event input and output ports and the internal state variables inherited from the abstract description of a simulation model, we also define control input ports and measurement output ports for each module. Figure 5.1 shows the overall

structure of a simulation module. Typically, there are a number of parameters in a module which affect its operation during a simulation run. Control input ports allow the simulator designer to configure a module by changing some or all of its operating parameters when the module is incorporated into a simulator.

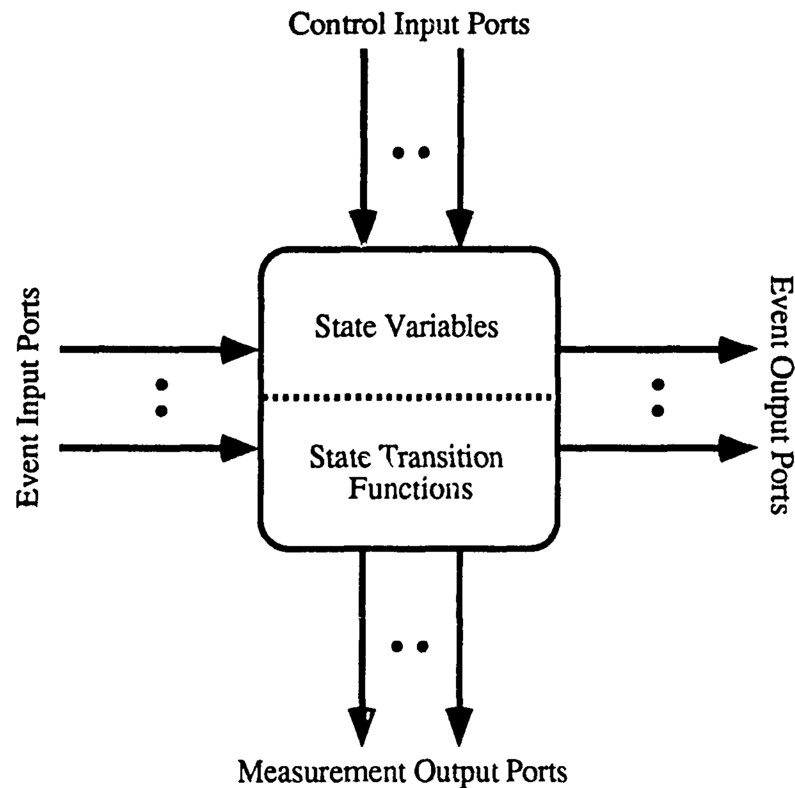

Figure 5.1: Structure of a Simulation Module

For example, in a memory subsystem, the width of the access path to the memory and the cycle time of the memory are two parameters which affect the amount of data that can be transferred to and from the memory system in a given time-span. A generic simulation module of such a memory system can be provided by allowing these two parameters, among others, to be configured using the module's control inputs. Control input ports increase the flexibility of a simulation module, and therefore its value for later re-use.

Another use of control inputs is to select or de-select certain features of a module. For example, we might be able to direct a module to perform the simulation of the subsystem it represents at a specified level of internal detail. This would allow the designer of a simulator to trade off simulation time versus the level of modeling detail. Other uses of

control input ports include the selective gathering of statistics about a module's operation, such as number and types of events received, average duration of requests issued by other modules etc.

Measurement output ports represent the flow of data generated by a module's statistics collection facilities during a simulator run. In its simplest form, a measurement output port signals that some particular data is collected by the module which will be printed after the simulator run has terminated. A more advanced form of the measurement output port may allow the simulator designer to discard some or all of the collected results, or to re-route them to a file for later analysis. Other ports may be used to feed the collected statistics to a separate program to perform analysis or plotting, either concurrently with the simulation or after its termination. These facilities allow a simulator designer to control and limit the amount of information that is generated as the result of a simulator run.

As described in the preceding paragraphs, a module interacts with its environment through event input and output ports, control input ports, and measurement output ports. The event input and output ports serve the dual purpose of specifying the connections between modules when configuring a simulator and of providing the inter-module communications mechanism at simulator runtime. Control and measurement ports, on the other hand, are only provided for configuration purposes and have no connectivity requirements at runtime.

A module's internal state variables and state transition functions model the behavior of a real-world system through transitions in the module's state and govern the creation and consumption of events While the operation of some or all of the state transition functions, and thus certain characteristics of a module, may be modified by the user of a module through the control input ports, the module's internal mechanism is implemented as a program written in a simulation programming language and is not accessible from the outside. As will be shown in greater detail in chapter six, the code needed to implement the port mechanism is inherited from a base class which provides the necessary data structures and port access functions. This ensures compatibility between all simulation modules in a simulator and reduces the coding effort needed to implement a module.

5.1.2. Simulation Module Library

A simulation module library, or SML, is a collection of simulation modules. Each simulation module in a library is implemented as a C++ class. A module implementation therefore consists of three parts: a C++ class declaration, C++ source code which

implements the module's member functions, and a compiled object which holds the executable version of the module.

In addition to the three parts of a module required by C++, we also generate a module description file (MDF); the exact format and contents of this file are given in the next chapter. Briefly, the MDF contains such items as the module's name, the filename of its class declaration, source, and object files, a specification of the module's ports, and other similar information. The module description is needed to allow the interactive generation of a simulator description (covered later in this chapter) and for the generation of an executable version of the simulator (described in chapter six).

Ideally, a SML should be implemented as a "true" library with its own archiver and library maintenance utilities. This has the advantage that all parts of a library can be combined into one file and thus not only conceptually, but also physically, form one entity. An intermediate step would be to use a general-purpose archiving utility, such as the UNIX archiver *ar*, to create the SML. The disadvantage of combining the files for all modules in a SML into one archive is that the tools needed to generate a simulator, such as the C and C++ compilers, are not aware of the special-format library. They are thus unable to access those parts of a module, such as the class declaration, that are needed for the generation of a simulator. We have therefore chosen to set up a number of directories, each to hold one specific component of a simulation module. The combined contents of all these directories can be viewed as constituting all the simulation module libraries known to the system.

A SML is described by a collection of module descriptions which have been combined into one MDF. As each module description in turn includes the filename used for the module's class declaration, source code, and object files, which are stored in known directories within the file system, all components that constitute the SML are accessible by reading and interpreting the information in the description file.

5.2. Possible Users of a Simulator

So far, we have always referred to the person using a simulation module or a simulator in rather vague terms as "the user" of the module or simulator. Now it is time to take a step back and consider what different types of "users" there are, and what their purposes for "using" a simulator might be. We can discern three types of users, the module developer, the system modeler, and the system administrator or manager. The three differ in the amount of technical expertise that is required of them and in their purpose for using a simulator (see also Table 5.1).

The developer of a simulator module needs the highest level of technical knowledge of all of three types of users. Developers must be familiar with programming in general, and with the simulation programming language and the design and interface standards of the simulation system in particular. They also should have a certain amount of knowledge of simulation and modeling techniques to be able to assess the impact of design decisions on overall simulator performance. A developer's main task is to design and code the modules that are going to be used in a simulator. Another task which is best taken care of by users of this type is to maintain the module libraries. Library maintenance involves keeping an inventory of available simulation modules for the benefit of other users, incorporating newly developed modules into existing libraries, and updating existing modules as the systems they model undergo change.

The modules needed for the construction of a planned simulator are given to the developer in the form of a module requirement specification (**MRS**). The MRS details the functionality that must be incorporated in a simulation module and lists its input/output requirements; it also provides the algorithms needed to implement each module. A module developer then decides, based on the knowledge of available modules and SMLs, which of the given modeling requirements can be met by existing simulation modules and for which new modules will need to be implemented. Once the necessary new modules have been created, they are individually tested to verify their conformance to the module specification. When testing has been completed, they are incorporated into existing SMLs or grouped into a new library and are then available for use by other users.

A system modeler's main task is to build a simulator which accurately models the system under study. Necessarily, this type of user needs to be familiar with the system and also needs detailed knowledge of the technical discipline which covers the system. A good grasp of modeling techniques, needed for the translation of his or her expert knowledge of the system into an appropriate model, is also required of the system modeler. Once models of the system and its subsystems have been designed, they are passed to the module developer in the form of a MRS. The system modeler subsequently generates the simulator by selecting a module connection scheme and by choosing a module configuration which reflects the configuration of the modeled system.

Once a simulator has been generated, the task of system modelers is to arrive at a sequence of simulation runs which will provide the information necessary to answer the questions which prompted the simulation study. Using their knowledge of modeling techniques and information about the modules, they will vary the configuration of the

system and of the modules to explore different designs of the modeled system. This includes varying the types of component modules of the simulator, as well as their number and internal configuration. The system modeler then uses the results of the simulation to examine the advantages and disadvantages of different system configurations and to arrive at the "best" system configuration.

The administrative or managerial users employ a simulator as a tool which assists in the running or optimization of the system modeled by the simulator. They are not concerned with the details of the simulator's implementation and need no detailed technical or programming knowledge (cf. [Treu 1988]: "*..the user [of a simulation interface] should not be expected to be a simulation expert or even familiar with the technical simulation details and terminology*"). Typically, these users will not generate the simulator themselves, but will use an existing simulator to explore the effects of fine-tuning the operating parameters of the working system. For this purpose, it is sufficient to view the simulator as a set of connected building blocks, each of which models one component of the system.

Using their knowledge of the existing system and its operating parameters, system administrators can set up the simulator to reflect a desired state of the real system and can then modify some of the components' operation and observe the effect of the change on the system. Similarly, managers may want to determine whether, for example, the addition of a server to a local area network improves overall response time enough to warrant its acquisition. For this, they would modify the existing simulator by adding a module representing the server at the desired location and compare the results of the simulator run to those for the existing system. Users in this group employ the simulator to answer "what-if" questions about an existing system, not as a tool for designing a new system from scratch.

Even though we have treated the three types of possible users of a simulator as distinct groups, it should be obvious that in reality there will always be at least some degree of overlap between the different functions. To mention just two examples, a system modeler may very well have sufficient programming knowledge to also design and implement the needed simulation modules. Similarly, the leader of a system design group may use a simulator as a rapid prototyping tool to explore different configurations of a proposed system, while also being involved in the design of the models themselves. What we wanted to note here is that each of these groups has certain unique requirements when using a simulator which need to be recognized and supported by the simulation system. We shall see shortly how many of these requirements are met by the graphical interface to

the simulation system. Table 5.1 summarizes the different characteristics and requirements of each type of user.

	Module Designer	System Modeler	Administrator, Manager
Area of Expertise	Programming, Simulation System	System's Domain, Modeling	System Operations
Programming Knowledge	Detailed	Some	None
Modeling Knowledge	Some	Detailed	None
System Knowledge	None	Detailed	Some
Purpose of Simulator Use	Testing	System Development	System Optimization
Level of module detail known	High	Medium to High	Low or None
Simulator Design	No (testing only)	Yes	No
Simulator Configuration	No (testing only)	Yes (module connections and attributes)	Yes (module attributes only)
Need for Interactive Interface	Low	High	High

Table 5.1: Types of Simulator Users

5.3. The Simulator Development Cycle

Most software engineering models identify the following four phases in the software development cycle [e.g. Luqi 1989, Hekmatpour 1988]:

(i) Requirements Definition

(ii) Software Design

(iii) Implementation

(iv) Maintenance

The development of a simulator follows a similar life cycle. The initial modeling requirements for a simulator are determined by the system to be modeled; these requirements are then refined until the requirements for all component models in the simulator have been determined. The software design for each module is given in its module requirement specification. This design is then implemented and the resulting modules are subsequently integrated into executable form in the simulator. Experience with the simulator usually leads to demands for changes in, and extensions to, the original requirements documents and thus the development process is restarted from the beginning ([Parnas 1986] and others argue that there can be no satisfactory initial requirements document). This process continues until the simulator is sufficiently refined to produce the desired data, at which point it enters the maintenance phase. Here, changes are triggered by changes in the modeled system or by errors that are discovered in a model's design or in its implementation in a module.

The development of a simulator in our modular simulation system differs from the above process in the implementation and refinement phases; these differences are due to the possible re-use of existing modules in a simulator design. Thus the implementation phase proceeds in two distinct stages. In the first stage, we implement the needed simulation modules and incorporate them into SMLs. In the second stage, we implement the simulator itself by choosing appropriate modules from the SMLs and specifying a connection scheme between them and by configuring each module. Once a first simulator has been developed, it can then be easily refined and expanded by adding other simulation modules. The refinement process may continue at the level of the simulation modules, either by modifying existing modules, or by replacing a module by one or more others at a finer level of detail.

The modular design of the simulator also lends itself to an alternate design approach: By using existing simulation modules as building blocks, we can rapidly construct a prototype [Taylor 1982] simulator of a system. This assumes, of course, that most of the components needed for the simulator already exist as modules in some form. The modules used in the prototype need not be the exact modules required in the final design, but need only approximate the functions of the desired modules. Instead of creating a detailed design document for all parts of the simulator before even considering its implementation, we can use the prototype simulator as an aid during the requirements and design phases. This approach may save valuable time by uncovering flaws in the design early on [Luqi 1989, Hekmatpour 1988] and it can provide early estimates of the

performance of the modeled system. Moreover, those modules that already accurately model system components can be retained in the final version of the simulator and can be easily combined with newly developed modules. This avoids one of the most serious drawbacks of rapid prototyping, namely that of the dual development effort needed for the prototype and the production system.

5.4. A Tool for Interactive Simulator Generation

At this point, it should be quite clear that most of the work of designing and building a simulator involves the manipulation of simulation modules. A tool which assists all types of users with manipulating modules is therefore desirable, especially as there may be a large number of available modules, which may be held in several different SMLs. Such a tool is provided by the **SimBuild** utility, which forms a graphics-oriented windowed environment for the interactive generation of a simulator.

SimBuild provides an intuitive interface for simulation modules by representing each module as an icon on a bit-mapped graphics screen. The user selects a module by clicking on its icon using a mouse-controlled pointer. Module instances can be interactively created, edited, connected, and destroyed by selecting a module's icon and choosing the appropriate action from a menu. To determine the attributes and the number and types of input and output ports of a module, **SimBuild** reads the information stored in the module's description file. That same information is also used to perform consistency checks on the connections between modules and range checking on the module attribute

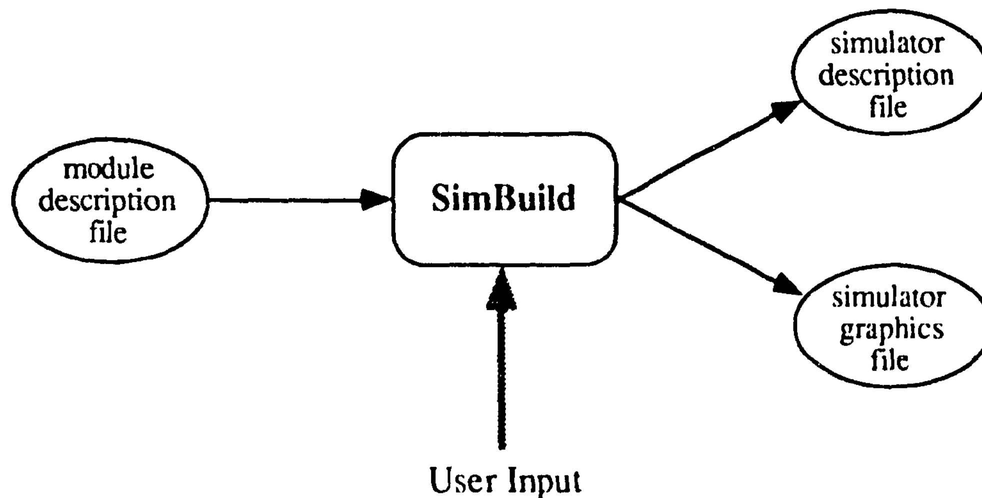

Figure 5.2. **SimBuild** Operation

values entered by the user.

A simulator is generated by creating instances of each of the simulator's component modules and by connecting them as required. Once a simulator has been generated, **SimBuild** saves a description of the simulator in a simulator description file (**SDF**) for future processing (see Figure 5.2). The SDF specifies the module instances in a simulator and their logical connections; its exact contents are detailed in chapter six. For its own internal use, **SimBuild** also records the simulator's graphical configuration, i.e. each module's icon position and connections, in a simulator graphics file (**SGF**). Previously generated simulators can subsequently be loaded into **SimBuild** again for further editing or expansion of the simulator.

SimBuild's main display is the simulator window, a scrollable bit-mapped graphics window (or *canvas* in SunView terminology) which shows the current configuration of the simulator. This main display is augmented by two control panels and two text windows (see Figure 5.3). Command buttons on the file control panel and the module control panel are used to select actions operating on files and modules. The library information window and the module information window display information about the currently selected module library and the current module type, respectively. There is also a one-line status window which displays messages about the current operating mode and error messages. Each of the windows and their functionality will be described in further

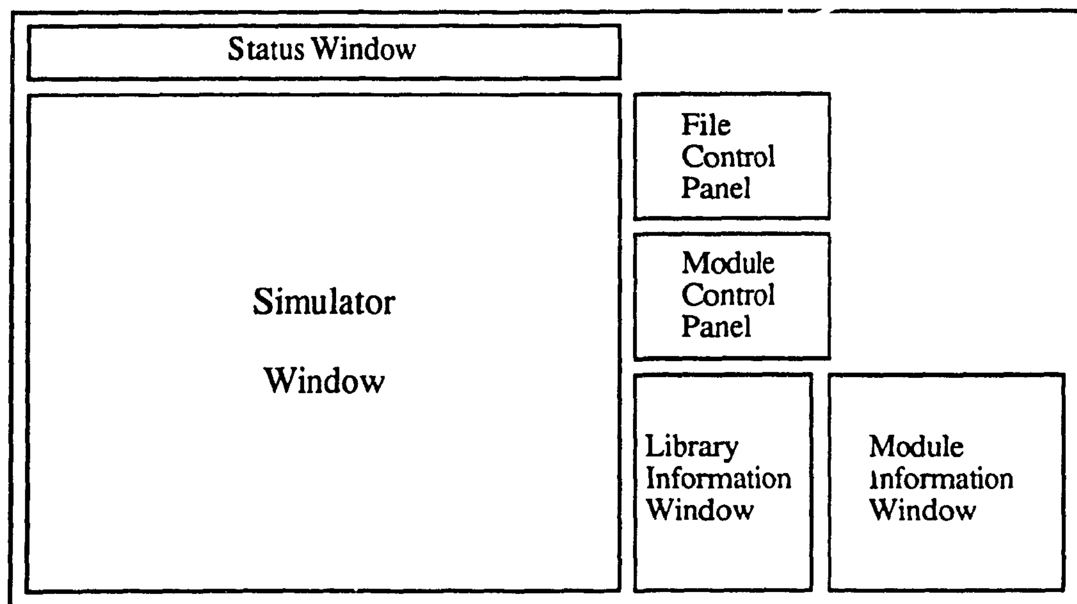

Figure 5.3: **SimBuild** Screen Layout

detail in the paragraphs to follow.

When **SimBuild** first starts up, the simulator window is empty as no simulator has been generated yet. Two items in the file control panel display the directory and filename of the currently selected module library, represented by its module description file, and that of the current simulator description file. The filenames initially selected are defaults which can be changed by the user before loading the module library or simulator description. Three panel buttons allow the user to *load* a simulator description from the named file, *save* it to that file, or *delete* the current simulator description. Selecting the *delete* operation will bring up a pop-up window requesting confirmation by the user before proceeding with the deletion. Similarly, all other actions which have potentially disastrous effects if they are selected by accident require user confirmation in **SimBuild**.

Another set of three buttons in the file control panel allows the user to *load* a module library, to *examine* individual modules in the library, and to *add* a coupled module to the library. *Loading* a module library makes it the current library, i.e. the library from which modules can be selected. It also updates the information in the library and module information windows. The library information window displays a list of the modules contained in the current module library. Each of the module names in the list can be selected by clicking on it to make the corresponding module the currently active module. When a library is loaded, the first module in the library is selected as the current module by default.

The module information window displays information about the active module. This display is generated from the module's entry in the module description file and lists the changeable parameters of the module and their default values, as well as the module's input and output ports. By editing the parameter values in this window, the user may set the default values for all instances of the module that are created during the session. The edited values are not saved permanently in the module description file, however.

By selecting *add a coupled module* from the file control panel, a user may combine a number of modules into a coupled module and add that module to the current library. The modules that are to be combined are designated by clicking on their icons in the simulator window. **SimBuild** will first determine the internal and external connections of the new coupled module. Internal connections involve only modules which are going to be components of the coupled module. The coupled module's external connections, i.e. those connections whose source or destination is not within the coupled module, determine the number and types of external input and output ports of the coupled module. **SimBuild**

will then replace the icons of the designated modules into one icon for the coupled module and update the display of any connections between this and other modules accordingly. Finally, **SimBuild** will generate a module description for the coupled module and add it to the current module library. The newly generated coupled module can then be acted upon like any other atomic module; for example, instances of the coupled module can be created in one step rather than having to create separate instances of all component modules.

For an illustration of this process, consider the system on the left side of Figure 5.4. Five modules (V, W, X, Y, Z) and their connections through ports (A, B, C, D, E) are shown much as they would appear in **SimBuild**'s simulator window. By clicking on the module icons X, Y, and Z, the user specifies these three modules as the components of a coupled module (indicated by the grey line around them in Figure 5.4). The connections between the port pairs B and D, as well as the one between ports C of modules Y and Z are internal to the coupled module; the remaining connections are external connections of the coupled module. The coupled module (indicated by a double outline in Figure 5.4) therefore has three ports A, C, and E, which connect it to the two other modules.

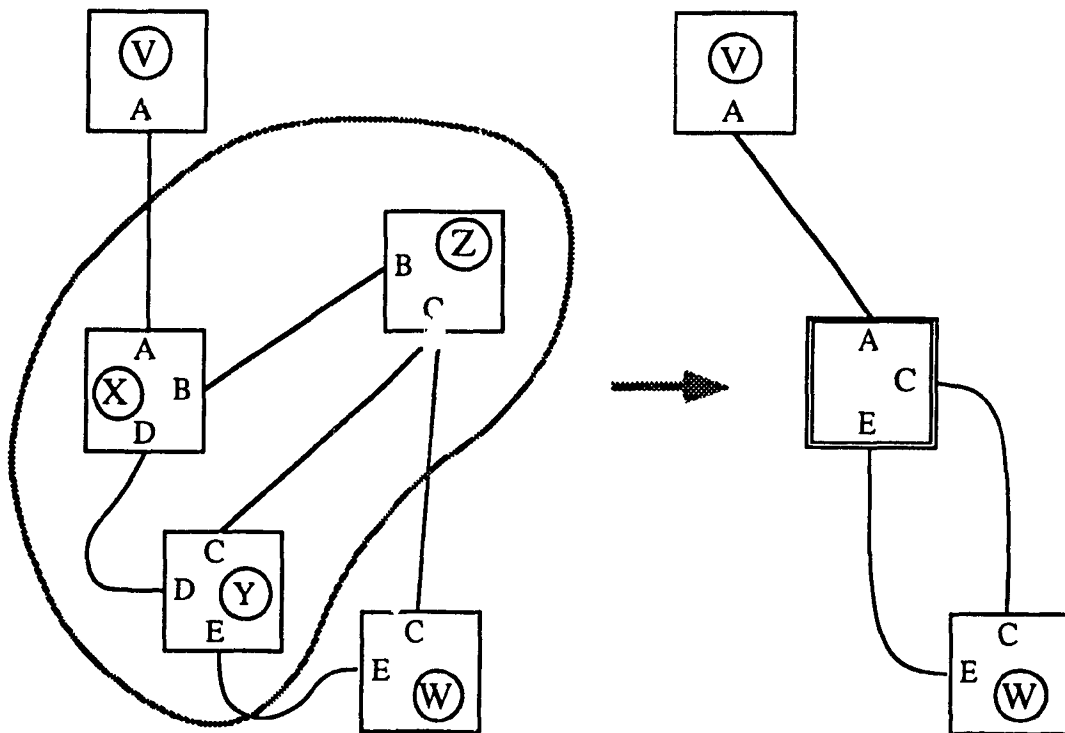

Figure 5.4: Coupling of Modules

The buttons on the module control panel allow the user to manipulate individual module instances. All buttons on this panel are <u>modal</u>, that is, by clicking one of the

buttons the user enters the mode described by the button's action. For example, clicking on the *Add a module* button will enter module creation mode. In this mode, an instance of the active module will be created at the current pointer position whenever one of the mouse buttons is clicked while the pointer is in the simulator window. The actions selected from the module control panel are mutually exclusive, i.e. selecting one button while in another mode will cancel the previous mode and enter the mode indicated by the selected button. Selecting any button on the file control panel also cancels the current mode.

Three of the buttons on the module control panel allow the user to create, examine and edit, and destroy module instances. As already described above, a module instance is created by clicking in the simulator window while in module creation mode. The instance of the module is represented by its icon and possesses the default parameter values of the currently active module. To create instances of other modules, the user must activate that module by clicking on its name in the library information window before creating the instance. To edit the parameter values of a module instance, the user selects the Examine module button and then selects the desired module. This will bring up a pop-up window which displays the current values of the module instance's parameters and the source and destination modules of any input and output connections that have already been defined for it. By editing the values in this window, the parameters for that instance (and that instance only) can be changed. Deleting a module not only deletes the selected module instance from the simulator, but also automatically deletes all connections defined between this and other modules. Thus deletion of a module can possibly lead to a "disconnected graph", i.e. ports of some of the modules may not be connected any longer.

Expanding a module from the module control panel reverses the aggregation of several modules into a coupled module. **SimBuild** will expand the coupled module into its component modules, creating an icon for each, will display the internal connections between the modules, and will adjust the display of external connections to properly indicate their source or destination module. By expanding a coupled module, the user regains direct control over its component modules, which may now be manipulated like any other module in the simulator. The expansion of a coupled module is limited to the selected instance of the module; it has no effect on the definition of the coupled module stored in the module library or on other instances of this coupled module, which may still be used as before.

The two final buttons on the module control panel are used to *create* and *delete* connections between the simulator's modules. To create a connection between two

modules, the user first selects the source and then selects the destination module. For source modules with more than one possible output port, **SimBuild** will request the desired output port in a pop-up window. It will then establish the connection between the two selected modules, provided the destination module has an input port of the appropriate type. If there is no such input port on the destination module, an error message will be displayed in the status window and the connection request will be cancelled. **SimBuild** thus allows only the establishment of valid connections between modules. A connection is deleted by clicking on it anywhere; the deletion has no effect on any other connections or modules, i.e. **SimBuild** makes no effort to deduce logically related connections which should be deleted together.

As we have seen, **SimBuild** provides a graphical environment for the interactive generation and configuration of a module-based simulator. It allows the convenient display and selection of modules from SMLs and thus improves the speed and efficiency of the simulator generation process. By introducing consistency checking for the connections between modules, **SimBuild** also ensures that the generated simulator is syntactically correct.

The intuitive user interface provided by the direct manipulation of module objects on the screen allows far faster generation of a simulator than a text-based interface would [Barth 1986]. It also encourages experimentation and the exploration of alternatives by taking the drudgery out of the simulator creation process ([Kramer 1989] deals with related issues with regard to use of software modules). While developers of simulation modules will not use **SimBuild** as their main working environment (their main task, after all, is to implement simulation modules, not to generate a simulator), it is still useful to them as it allows the rapid generation of a test environment for newly generated modules. On the other hand, **SimBuild** greatly improves the working environment of the system modeler and the administrative user, whose tasks are mainly concerned with manipulating simulation modules.

6. The Design and Implementation of the Modular Simulation System

As discussed in the previous chapter, **SimBuild** enables its user to interactively create a simulator description using modules from module libraries. **SimBuild** is one component of the Modular Simulation System, or **MSS**, presented in this chapter. After introducing the object-oriented, modular simulation environment provided by MSS, we describe and discuss the implementation of the base class for simulation modules which provides the software structure required of simulation modules for integration into MSS. We then describe the two remaining major components of MSS, the utility programs that assist the user with module description and simulator creation. Finally, we give an overview of the simulator design process using the facilities provided by MSS.

6.1. Design and Structure of MSS

The Modular Simulation System is designed to run under the UNIX operating system and uses some of the utilities and systems programs provided by the UNIX environment. Simulation programs constructed using MSS also use the facilities provided by the SAMOC simulation package described in chapter four. MSS is structured following the UNIX philosophy of providing a flexible and open environment with a set of interacting tools, rather than using the approach of single master program with a predetermined,

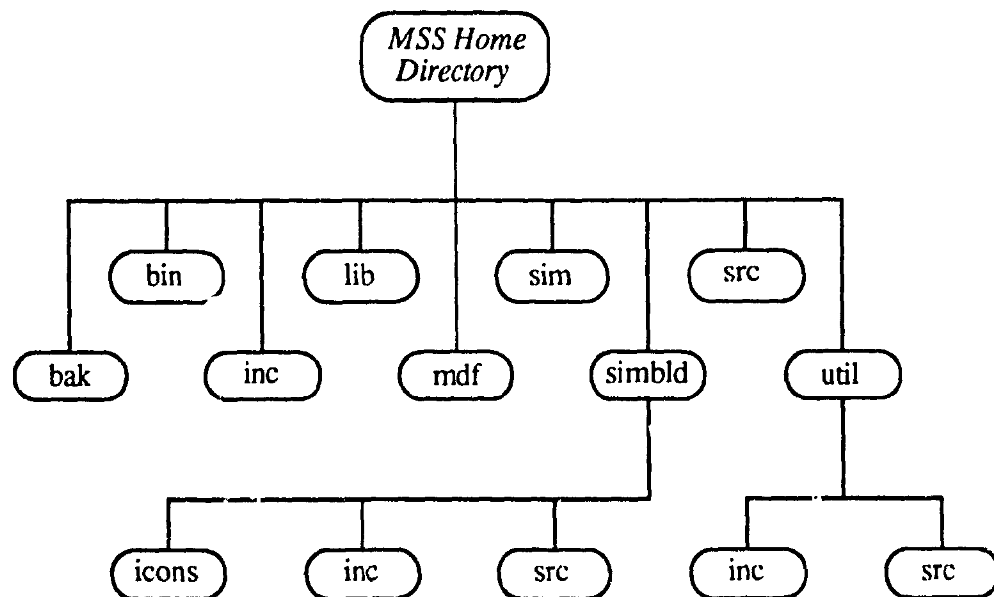

Figure 6.1· MSS Directory Structure

non-extensible functionality. As such, MSS consists of utility programs which are separately invoked from the UNIX command interpreter and which communicate using files with a standard file format. A set of class definitions and coding guidelines provided by MSS supports the creation of inter-connectable simulation modules which can be combined to produce an executable simulator.

To organize its components in a consistent way and to coordinate their operation, MSS defines a hierarchy of directories (see Figure 6.1) in which each subdirectory is assigned a specific function and holds files of a specific type (or types). MSS programs can then rely on finding the needed user and system files in one of those known directories and can provide default pathnames for user-generated files. The location of user files which reside in a directory different from the default one may be specified by giving the complete pathname of the file. Table 6.1 gives a short overview of the contents of each standard MSS directory; the exact use of each will be discussed further in the following paragraphs.

Directory	Contents
bak	Backup of system and user files
bin	Executables for MSS utilities and simulators
inc	Simulation module header files
lib	Simulation module object files and libraries
mdf	Simulation module description files
sim	Simulator description files and source files
simbld	SimBuild special files and source code
src	Simulation module source files
util	Source code for MSS utilities (except SimBuild)

Table 6.1: MSS Directory Contents

The home directory for the Modular Simulation System, i.e. the directory which forms the root of the MSS directory tree, may be located in a particular user's directory or in a system directory known to and accessible by the user community. The home directory also serves the additional function of providing some global information files which are used by all system components. All MSS utilities and related files (such as UNIX *make-files*) use the pathnames given in the global files to access the different MSS directories. We can thus easily configure the system by editing the information in the global files. The contents of the global files is listed in Appendix A.

As mentioned in the previous chapter, a simulation module, even though a single conceptual entity, is described by several different files. These files are stored in separate directories in MSS, with the convention that files belonging to the same module share the same basename, modified by a common extension (see Table 6.2 for a list of the filename

Extension	Description
.hxx	C++ header file; holds one or more class definitions
.cxx	C++ program file; holds implementation of member functions for one or more classes
.o	Object code file; holds compiled member functions for one or more simulation module classes
.a	Archive file; holds a collection of object code files in a format suitable for use by the linker
.make	Simulator makefile generated by SimMake
.mdf	Simulation module description file; holds descriptions for one or more simulation modules in a format usable by **SimBuild** and **SimMake**
.sdf	Simulator description file; holds specification of a simulator in a format usable by **SimMake**
.sgf	Simulator graphics file; holds graphical layout of simulator description (used internally by **SimBuild**)

Table 6.2: MSS and UNIX Filename Extensions

extensions defined by the UNIX system and by MSS). The class definition for a simulation module is kept in its header file, which has the extension "**.hxx**" (mandated by the C++ compiler); this header file is kept in the *inc* directory of include files. The *inc* directory also contains certain other header files provided by MSS, such as the one for the base class of all simulation modules. The implementation of a module's member functions is contained in the module's source file (extension "**.cxx**", again mandated by the C++ compiler), which resides in the *src* module source directory.

As C++ provides separate compilation of source modules, each module has an associated object file ("**.o**"), which is stored with its source file until incorporated into a module library. The *lib* directory contains object libraries for those modules which are part of a module library and the library of MSS provided objects. For reasons discussed in chapter five, these libraries are maintained using the UNIX *ar* archiving utility so that they can be accessed directly by the UNIX system linker. Finally, each module's description, needed for the configuration and generation of a simulator, is kept in a (simulation) module description file (extension "**.mdf**") in the directory *mdf*. This file is generated by the **SimDesc** utility program described later on and contains the description of all the simulation modules which form a module library.

As described in the previous chapter, the **SimBuild** utility program, based on the information stored in module description files and on user input, generates a simulator description file, which specifies the simulation modules and the connection scheme for a simulator. This file (extension "**.sdf**"), together with the simulator graphics file (extension "**.sgf**"), is kept in the *sim* directory. This directory also holds the simulator source file and the make file ("**.make**") produced by the **SimMake** utility for each simulator.

The executable object files for simulators, as well as the executables of the MSS utility programs are kept in the *bin* binaries directory. The directory *util* contains subdirectories with the C++ header and source files for the MSS utilities. Similarly, the *simbld* directory holds the header and source files for the **SimBuild** program and some additional files, such as icon definitions, needed for its operation. The *bak* directory is used for a local backup of MSS system and user-generated source files; this is intended as an insurance against accidental deletion, not as a permanent backup mechanism.

Most of the files in the MSS directory tree are generated by the activities of MSS users. In addition, there are also certain MSS system files, namely the MSS utilities and the class declarations for simulation files, which are provided with MSS. The names and a

short description of each file is given in Table 6.3; the remaining sections of this chapter will discuss them in further detail.

File	Description
simdesc	The **SimDesc** utility program
simbuild	The **SimBuild** utility program
simmake	The **SimMake** utility program
Makefile	Makefile to automate compilation and installation of MSS components (several exist in different directories)
Make.Path	Definition of MSS directory structure (for use with makefiles)
Make.Rules	Compilation rules and compiler options (for use with makefiles)
MSS_Path.h	C header file definitions of MSS directory structure
msg_queue.hxx	Declarations for classes *msg* and *msg_queue*, which implement communications between simulation modules
simmod.hxx	Declarations for class *simmod*, the base class of all MSS simulation modules, and other modules supplied by MSS
simmod.cxx	Definitions for member functions of classes *msg*, *msg_queue*, *simmod*, and other MSS-supplied simulation modules
mss.mdf	Module descriptions for MSS-supplied simulation modules
mss.a	Object code archive for member functions of MSS-defined classes

Table 6.3: MSS system files

6.2. Integration of Modules Into a System

As mentioned before, MSS provides a base class from which simulation modules inherit all the necessary inter-module communication mechanisms. In traditional (non-object-oriented) software system designs, interface standards between different modules are designed, often by members from different design groups, and then implemented sepa-

rately in each module. This method has two disadvantages: the implementation of the interface may differ between modules, and changes to a particular interface necessitate changes in all modules using that interface. The first can lead to difficulties when two modules attempt to communicate using different implementations of the interface if their respective designers have interpreted the interface standard differently. The second problem, caused by the explicit coding of the interface in each module, leads to a high degree of maintenance activity, which is clearly undesirable and error-prone.

Implementing an interface standard as a class avoids the aforementioned problems. First, as the class is designed by one designer or group, there is a single standard implementation. Thus, the problems arising from different interpretations of a design document are avoided as all modules, using inheritance, share the same implementation. Second, as the exact implementation of the class is hidden, changes to it are transparent as long as its access functions remain unchanged. Thus, no code changes are necessary in the client modules if the internal organization changes.

By requiring all simulation modules to be inherited from a provided base class which implements a standard interface between modules, MSS can ensure that all simulation modules in a simulator use the same connection mechanism. All connections between modules are therefore compatible in principle, which makes it possible to use an automated system to generate the code to set up the connections within a simulator. Additional restraints imposed on the validity of module interconnections, such as the rule that only ports of the same type may be connected, further restrict the number of possible connections in practice.

6.2.1. Design of the Simulation Module Base Class

Our goal in this section is to explore some of the issues arising in the design of the inter-module communications mechanism to be used by MSS and to arrive at a design for a class which implements this mechanism in an efficient way. This class is then used as the base class for the implementation of all other simulation modules.

Simulation programs are designed and implemented using facilities provided by the SAMOC simulation package which was described in chapter four. As simulation modules are the main components of a simulator, each representing one component of the modeled system, they are implemented as SAMOC entities. Each simulation module can therefore be scheduled independently from other modules. When designing modules for a simulator

based on SAMOC, we must thus always keep in mind that the modules are to cooperate in the fashion of independent processes in a multi-tasking system.

As seen in chapter three, we can describe a simulation model as an object which interacts with its environment through a fixed set of input and output events. A simulation module implements typed input and output ports which correspond to each of the possible input and output events; the type of each port determines events of which type can be received or sent by the port. Events are represented by typed messages which are exchanged between module ports. The inter-module communication mechanism implemented by MSS should follow this model of typed, unidirectional ports exchanging typed messages. Each module thus would need a set of input ports, one port for each event type the module can receive, and a set of output ports, one for each type of event the module generates.

Unfortunately, when trying to implement this simple design, several problems arise, some having to do with the simulation facilities provided by SAMOC, and others with the overall simulation environment that we are designing for. The design environment causes problems in the implementation as we plan to design modules which are self-contained, re-usable entities. We cannot assume, and in fact do not know, anything about the context each module is going to be running in or about the connections to and from a module's ports that will exist. Thus, even though we know the type of port a module's output port is to be connected to, we do not know on which entities those ports reside, nor, in general, how to address a particular port on another entity.

One solution to this problem is to define a centralized message dispatcher module which receives all outgoing messages and routes them to their destination modules. While this approach has been shown to work before [Messerschmitt 1988], it does not solve by itself the problem of how to address a module's port. Either the central router has to know each possible module's port configuration, making modifications of the router necessary every time a new module is implemented, or it must restrict the possible port types, e.g. limit communication to untyped messages (again, see [Messerschmitt 1988]). Furthermore, using a central router introduces additional overhead, both for routing and for task switching, as all messages now have to pass through the router.

Another problem, familiar in all multi-tasking systems, is that modules with more than one input port which are waiting for an incoming message need a mechanism to suspend themselves pending the first arriving message on any port. As in any multi-tasking system, we do not want a module to waste computation time by polling its input ports in

such a situation. While some operating systems provide a mechanism to wait for the first of several input events (cf. the UNIX *select* system call), SAMOC only provides synchronization mechanisms for waiting for one event at a time. A module with more than one input port which suspends itself waiting for a message on one port thus would not be activated if a message arrives at another port.

To avoid the problems associated with more than one input port, the actual design of the base class for simulation modules deviates from the conceptual model. Each module only possesses one input port, which can accept messages of all types. Messages exchanged between modules are still typed, but the type of a message is now determined by a type field in the message, rather than by the type of the port through which it passes. The receiving module then uses the message type at runtime to interpret the message. Having only one input port solves the synchronization problem for a module awaiting messages and the problem of addressing the destination port. With a known destination port address, modules can exchange messages directly, eliminating any routing overhead. So, even though input and output ports are still part of a module's description so that we can verify a valid connection, the implementation of a module has only one input port and no output ports (as the messages are sent directly to the destination module's input port by the sending module).

Another issue that arises in the design of the module communication mechanism is the protocol to be used for exchanging messages between modules. Modules use messages to trigger events in other modules, analogous to the way a simulation model triggers events in other models by its actions. In many cases, however, a module will not know the amount of time a certain action will take when executed by another module. We therefore need to find a way to synchronize the actions of two or more independent modules, without knowing, at the time each module is designed, the number or types of the modules involved because of the modular design of the simulator.

To illustrate this synchronization problem, consider simulating a computer system in which the central processor accesses the system memory to obtain instructions and data. In this example, we consider only the memory's access time as a variable and are not concerned with the inner details of the CPU and other modules. We will examine three different cases which are relevant in this context, namely a simulation using: (i) only *one module*, (ii) *two* connected *modules*, and (iii) *three or more* connected *modules*.

Case (i): One module

The simplest possible simulation of the sample system would consist of a single combined CPU/memory module which models memory access simply as a certain delay incurred by the CPU on every access (see Figure 6.2a). Changes in the memory's access time would be modelled directly as a change to the delay parameter; obviously, no inter-module communication or synchronization is needed in this case.

Case (ii): Two modules

We can design a simulator of the example system consisting of two modules, one for the CPU, and another for the memory module (Figure 6.2b). In such a simulation, the CPU module cannot be aware beforehand of the access time used by the memory module. It must therefore use a message to notify the memory module of the access request; the CPU module thus generates an *access request* message and transmits it to the memory

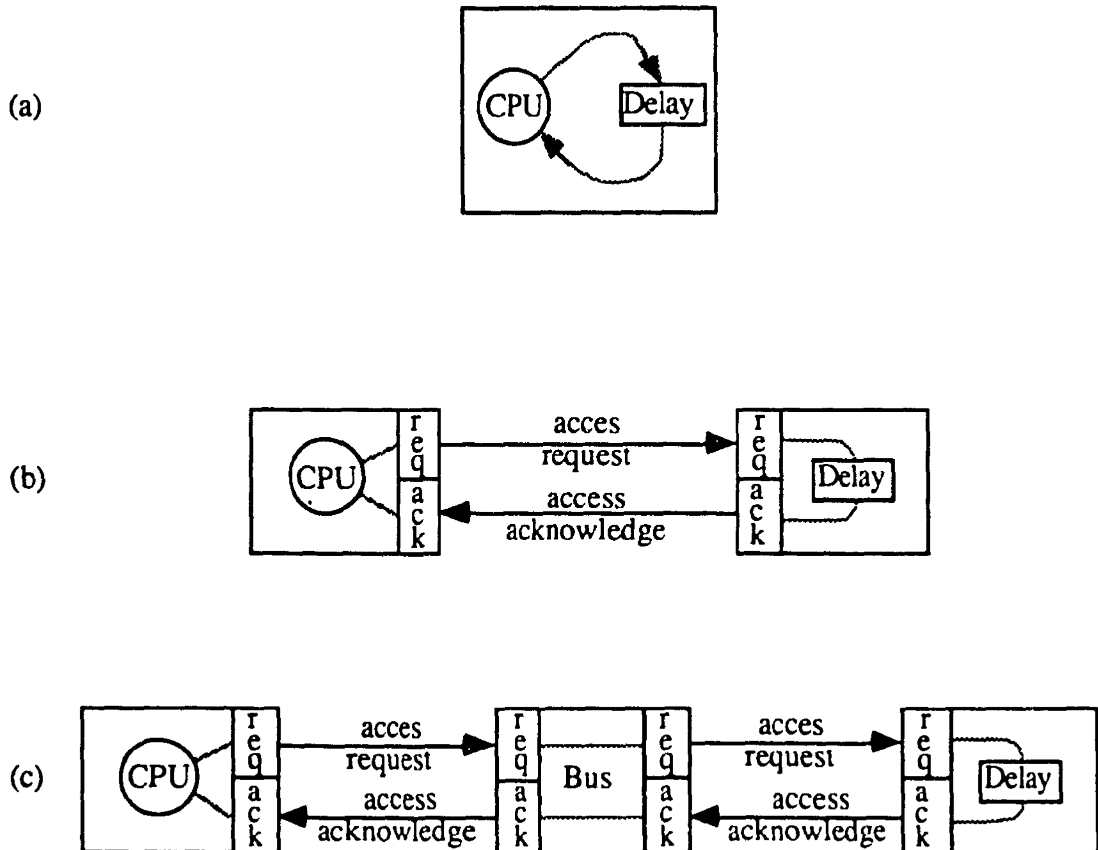

Figure 6.2: Simple CPU/Memory Models

module. The memory module again models the access time as a (user-configurable) delay time which has to elapse before the simulator can continue with the execution of the CPU module.

Consider what happens in this case if both modules are implemented as SAMOC entities of the kind described in chapter four. Assuming the CPU module is currently active, then sending the access request to the memory module will not stop execution of the CPU module, nor will it result in the activation of the suspended memory module. We must find a way to ensure that any further execution of the CPU module stops as the CPU attempts the memory access (barring such events as overlapping the memory access with some CPU-internal operation), and that the memory module becomes activated. We must further give the memory module a way to re-activate the CPU module once its access time has passed, i.e. the accessed data is available at the CPU.

We can accomplish both tasks by using a protocol which uses a pair of messages for each memory access, say an *access request* message passing from the CPU to the memory module, and an *access acknowledge* message passing back from memory to the CPU. For the CPU module, we block any further activity after the sending of the *access request* message by waiting for the appearance of the *access acknowledge* message on the proper input port (see Listing 6.1a for the pseudo code of the CPU module). The memory module (pseudo code in Listing 6.1b) starts its existence in a suspended state, waiting to receive an *access request*. Once this message is received, the module is activated and, after any internal computations that may be necessary, suspends itself again for the time required for accessing it. Once this time has passed, it is re-activated and returns the *access acknowledge* message to the CPU module and again blocks waiting for the next message.

```
(a) CPU Module
     WHILE forever DO
          // internal computations
          SEND access request message
          WAIT for access acknowledge message
          // internal computations
     ENDWHILE;

(b) Memory Module
     WHILE forever DO
          WAIT for access request message
          // internal computation
          HOLD for access time
          SEND access acknowledge
     ENDWHILE;
```

Listing 6.1: Pseudo Code for CPU and Memory Modules

While there may be simpler ways of modeling such a system, we must keep in mind that our goal is to develop **reusable** simulation modules. Therefore, we cannot make any assumptions about the hardware structure that is being modeled or about the specific connection scheme between modules. By implementing the modules using the message passing mechanism with the request/acknowledge protocol shown in Listing 6.1, we can create simulation modules which are general enough to be re-used for modeling different systems.

Case (iii): Three or more modules

For this case, consider the system depicted in Figure 6.2c. The CPU module is now connected to the memory module via a bus module, possibly with other hardware modules also contending for bus access. Even though this system is more complex than the one in the previous case, we are still able to use the same two simulation modules to model the processor and memory. The CPU module's output port is now connected to the bus module, which transmits any *access request* message it receives to the memory module. Similarly, the memory module's acknowledgement is routed through the bus module to the CPU. As the destination module of each message is not determined by the code in the sending module, but given by the connections between modules' output and input ports, no changes are necessary in either the CPU or memory modules. The bus module alone is aware of the correct destination for each message and takes care of the necessary message routing. Similarly, contention for the bus and any associated increased memory access times for the CPU, are all modeled in the bus module and are transparent to the CPU module.

We view each pair of messages, such as the *access request* and *access acknowledge* messages described above, as subtypes of the same message type. The message type determines the purpose of the message, e.g. the *access* message type in the example is used to model memory access, while the subtype indicates the direction of the message, e.g. a *request* is received by the memory module and an *acknowledge* is returned by the memory module to the sender. Another point of view is to consider each message pair as the two sides of a token which is used to activate exactly one of a set of (functionally) related simulation modules. As more than one token can be in circulation at any one time, more than one simulation module can be active at a time.

As we expect this *request/acknowledge* protocol to be useful in many cases, the communications mechanism of MSS provides some special functions for this type of inter-

module communication. For example, a *request* message can easily be converted into the matching *acknowledgement*. However, these are additional functions provided for the convenience of the user and their use is not mandatory. Thus communications between modules are not limited only to this protocol, but may take any other form found desirable by module designers.

6.2.2. Implementation of the Simulation Module Base Class

Simulation modules, being the major components of every simulator, are implemented as SAMOC entities. Consequently, class ***simmod***, the base class of all simulation modules, is derived from SAMOC's class ***entity***. It implements the inter-module communications mechanism and provides the necessary functions to send and receive event messages. Additionally, it provides a set of member functions which outline the framework that each simulation module should follow in order to be compatible with other MSS simulation modules.

The communication mechanism is implemented using two additional classes, ***msg*** and ***msg_queue***. Class ***msg_queue*** implements a priority linked list for inter-module messages, represented by instances of class ***msg***. Classes ***msg_queue*** and ***msg*** are themselves derived from a generic linked list class provided by SAMOC; they share a relationship similar to the one described for classes ***entity*** and ***queue*** in chapter four.

Access to these MSS-defined classes is provided through their header files in the MSS include directory. Files **simmod.hxx** and **msg_queue.hxx** give the class declarations for class ***simmod*** and for the classes ***msg_queue*** and ***msg***, respectively. The complete class declarations and the definitions for all classes provided by MSS are listed in Appendix A.

Class ***msg*** (see Listing 6.2) provides member variables for the fields required by each message, such as message type, identification of the sender and destination module, message priority, and message subtype. It provides access functions for these variables, as well as the member functions required by its base class ***pr_list_el*** to manipulate messages. A message's type and priority can only be set once, when the message is created. As messages of different types may have additional member variables specific to their type, changing the message type is not allowed. On the other hand, the source and destination modules of a message may vary, so we have to provide a way of changing the sender and destination identification fields. Four additional member functions (*set_req*, *set_ack*, *is_req*, *is_ack*) support the request/acknowledge protocol described above and

give access to a message's subtype. Note that Listing 6.2 gives only an abbreviated class declaration; many of the listed functions are implemented as C++ inline functions for efficiency.

```
class msg : public pr_list_el {
  int       Priority;    // message priority
protected:
  // all messages provide the following fields
  msg_type  Mtype;          // type of this message
  mod_ID    SendID;         // ID of sending module
  mod_ID    DestID;         // ID of destination module
  boolean   ReqAck;         // T if request, F if acknowledge
public:
  msg       *prev();        // prev message in queue
  msg       *next();        // next message in queue
  msg_queue *current_list();   // queue the message is in
  void      insert(msg_queue* mq, boolean before=FALSE);
  void      remove();       // remove messages from msg queue
  double    rank();         // for ordering messages by priority
  // access functions for member variables
  int       prio();         // priority
  msg_type  type();         // message type
  int       get_send();     // get sender ID
  int       get_dest();     // get destination ID
  void      set_send(mod_ID);  // set sender ID
  void      set_dest(mod_ID);  // set destination ID
  void      set_ID2(mod_ID, mod_ID);      // set both IDs
  void      swap_ID();            // exchange sender/dest IDs
  // support for request/acknowledge message protocol
  void      set_req();      // make it a request message
  void      set_ack();      // make it an acknowledge message
  boolean   is_req();       // TRUE if request
  boolean   is_ack();       // TRUE if acknowledge
  void      print();              // print type & priority
  msg(msg_type mt, int pr = 0);
};
```

Listing 6.2: Declaration of Class ***msg***

Class ***msg_queue*** is used to hold messages (see Listing 6.3). It only has to provide member functions to determine its current length and to retrieve the first and last messages in the queue as the remaining functionality is already implemented in ***msg*** or in its parent class ***pr_list***.

A simulation module's input port is implemented in class ***simmod*** by a pair of member variables (see Listing 6.4). One, *mqueue*, is of class ***msg_queue*** and is used to hold all messages received by the module; the other, *inqueue*, is an instance of SAMOC's class ***bin*** and is used for module synchronization purposes. A message received by a module is enqueued into *mqueue* and at the same time a token is deposited into *inqueue* to

indicate the presence of a message. Thus, if the receiving module has been waiting for a message (by calling member function *wait_msg*), it is activated and inserted into the event list at the current time. Once it resumes execution, the module can then determine the type of the received message and act accordingly.

```
class msg_queue : public pr_list {
public:
// inherited from class pr_list:
// void       print();     // print elements on queue
// boolean    empty();     // TRUE if queue is empty
  // adapted from class pr_list:
  int      length();      // current queue length
  msg*     first();       // first message in queue
  msg*     last();        // last  message in queue
  msg*     getfirst();    // get the first msg or NULL
};
```

Listing 6.3: Declaration of Class ***msg_queue***

Class ***simmod*** provides a number of member functions to receive and send messages. To receive a message, the module calls *wait_msg*, which will attempt to *take* a token from the bin *inqueue*. If no token (i.e. no message) is available, this will suspend the module until a message is received. After processing a message, the module may test for further messages waiting in the input queue using the member function *msg_avail*, and may use *get_msg* to retrieve the next message from the queue (if it exists). All three functions by default retrieve a message any type from the input port queue; an optional argument may be supplied to apply the operation only on messages of a certain type. Thus it is possible for a module to wait until a message of a specific type has been received, or to retrieve all messages of a particular type from the input before processing other messages. An additional member function *only_msg* can be used if only messages of a single type can be processed by a module; it will lead to a runtime error if any other message is received by the module.

To send a message to another module, simulation modules use their member function *send_msg*. This function will put the message directly into the input queue of the receiver module and deposit a token in the receiver module's *inqueue* bin; two pointers, one to the message to be sent and the other to the receiver module are passed as arguments. ***Simmod*** also provides higher-level send and receive functions which support the request/ acknowledge protocol. Functions *request* and *reqack* can be used to issue a request message and wait for its acknowledgement; while *request* will return an error indication if it receives any message other than the expected acknowledgement, *reqack* will generate a

runtime error and should therefore only be used if no other message can be received. *Send_ack* returns an automatic acknowledgement given a request message.

Class *simmod*'s member functions *init_port*, *init_attr*, *body*, and *report* provide the interface between a simulation module and a simulator's main program; the purpose of each function is described below. Module classes derived from ***simmod*** must re-implement these functions and adapt them as needed as ***simmod*** only implements the minimal needed functionality. As the number and type of arguments to some of these functions varies depending on the requirements of each module class, they cannot be implemented as virtual functions, but must be redefined as overloaded functions in the derived classes. This also has the beneficial side-effect of being more efficient than an implementation as a virtual function.

```
class simmod : public entity {
  bin       *inqueue;    // count incoming messages
  msg_queue *mqueue;     // holds incoming messages
protected:
  mod_ID    ID;          // module ID
public:
  // member functions for manipulating messages
  int      msg_length();                        // length of mqueue
  void     send_msg(simmod*, msg*);             // send msg to a module
  msg*     wait_msg(msg_type mt = ANY_MSG);     // await next message
  msg*     get_msg(msg_type mt = ANY_MSG);      // get next message
  msg*     only_msg(msg_type);                  // wait for typed msg
  boolean  msg_avail(msg_type mt = ANY_MSG);    // TRUE if msg available
  // support for request/acknowledge protocol
  msg*     request(simmod*, msg*);           // send request, get ack
  msg*     reqack(simmod*, msg*);            // send request & wait ack
  void     send_ack(simmod*, msg*);          // return acknowledgement
  // Fatal error handler: prints routine, error message, message
  void       mod_error(char* rt, char* em, msg* mess=NULL);
  simmod();
  ~simmod();
  void init_port(mod_ID id);  // output port initialization
  void init_attr();         // module attribute initialization
  void body();              // simulation time behavior
  void report ();           // custom final reports
};
NEW (simmod, 1024);
```

Listing 6.4: Declaration of Class *simmod*

Due to requirements imposed by SAMOC, which arise from the implementation of the coroutine calling mechanism, constructors and destructors for classes derived from class ***entity*** may not possess any arguments. Similarly, SAMOC requires each entity to provide a parameterless procedure *body* which implements the runtime behavior of the

entity. ***Simmod*** implements the constructor and destructor functions, which create and destroy the ***bin*** and ***msg_queue*** objects for the input port. Similarly, it declares dummy *body* and *report* functions.

As it is not possible to pass arguments to the constructor function when a module is created, we have to implement additional functions to properly initialize the module's member variables. While this defeats one of the reasons for having a constructor function, namely transparent object initialization, it turns out that it also simplifies the overall simulator design. Each simulation module has two such initialization functions, named *init_port* and *init_attr*. As the name implies, *init_port* is used to initialize a module's connections to other modules, i.e. its output ports, and *init_attr* initializes a module's user-configurable attributes. The values used to initialize a module's attributes are selected the by the user in **SimBuild**.

The parameters supplied to *init_port* are the module identification number (supplied by **SimBuild** in the simulator description file) and, for each of the module's output ports, a pointer to the destination module. Module identification numbers are used to set a message's sender and destination ID fields; they may be used by other modules to route a message to its desired destination. The pointers to the destination modules are used by a module's *send_msg* function to access the destination module's input queue. The connection scheme set up using *init_port* is the one configured by the user in **SimBuild**; however, it is not until the simulator is actually running and until a module instance has been created that we can access a module's address and thus construct a pointer to it. By creating all the modules of a simulator before calling their *init_port* functions, we can easily initialize the destination module pointers; if this had to be done in each module's constructor, it would be very difficult, maybe even impossible, to find an order of creating modules without encountering forward module references.

The final function prototype provided by ***simmod*** is *report*, which may be used by a module to generate a customized report in addition to the report that is automatically generated by SAMOC upon the termination of a simulation run.

Classes ***simmod*** and ***msg*** together provide the functionality needed to connect simulation modules and to exchange messages between them. As they do not provide any additional features, a user will not generate instances of either class, but will only use them as prototypes from which to derive simulation modules and message types which incorporate the needed information. Figure 6.3 shows the relationship between classes declared by SAMOC, by MSS, and by the user.

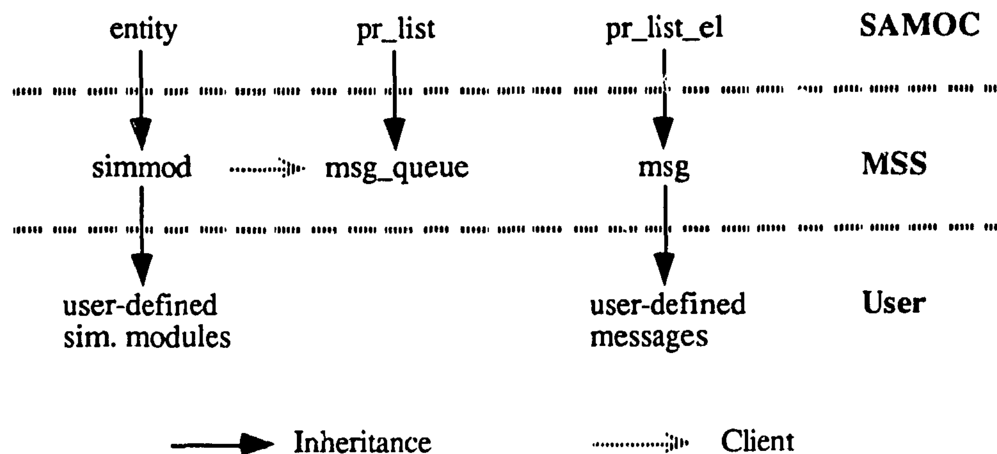

Figure 6.3: Relation Between Classes in a Simulator

An alternate design for the module and message base classes which was considered initially was to model both as entities. The advantage for this would have been that we could have used the facilities provided by SAMOC, such as class ***queue*** or the ***waitq*** synchronization mechanism, instead of implementing similar functionality in classes of our own. However, as a large number of messages may exist during the runtime of a simulator, these would have put an additional burden on the simulation system both in terms of additional event list and entity management and in terms of increased statistics gathering. We therefore decided to implement messages and message queues as new classes (still making some use of the code in SAMOC's priority linked list class) to reduce the system overhead. Also no valuable measurements are lost as the ***bin*** objects used to implement the input port keep some statistics and further measurements are easily implemented within each module.

6.3. Utilities: SimDesc and SimMake

So far we have discussed how to structure simulation modules and how to implement the inter-module communication mechanism. We have also shown how we can construct a simulator description interactively using the simulation modules. In that discussion, we have already mentioned that **SimBuild** takes information from a module description file and generates a simulator description file. We now describe how to generate the former, and how the latter is used to actually create an executable simulator.

6.3.1. SimDesc

A simulation module is described by three files: its header file, which holds the C++ class declaration of the module, its source file, which implements the module's member functions, and its module description file (**MDF**), which lists information about the module in a well-defined format. The MDF is a textfile and the information stored in it constitutes some of the documentation for a simulation module. However, the information is also intended to be read and interpreted by MSS utility programs and thus has to follow a rather fixed format. While a module description file could conceivably be generated using a text editor, it is much easier and less error-prone to generate it using the **SimDesc** program.

SimDesc is a simple text-oriented, interactive utility program. Its three main objectives are to generate new module descriptions, to update old descriptions, and to maintain a number of module descriptions within the same file. The module descriptions combined in the same file constitute a module library.

When starting a session with **SimDesc**, it is assumed that the user has a written module specification at hand or is otherwise familiar enough with the attributes of a module to describe them correctly. It should be remembered that the information stored in a module description file is the only information about the module that MSS is able to interpret. If this information is erroneous, there is nothing MSS can do to prevent the incorrect use of the module; features that may be implemented in a simulation module, but are not specified in the module's descriptive entry, are likewise ignored by the system.

When **SimDesc** is started, the user may specify a filename as an optional argument on the command line. If a filename is given, **SimDesc** will search for a module description file of that name in the *mdf* directory and, if found, read it in. **SimDesc** will then enter its main command loop, from which the user can add, edit, list, and delete individual module descriptions, obtain a list of all modules in the file, save the current or read a new description file, or quit the program.

Adding a new module description proceeds interactively, with **SimDesc** prompting the user for each component of the module description in turn. Similarly, when editing a module description, the user is shown the current value of each component and may then replace it with a new one. Listing a module description displays the information about a module as it will be saved to disk. Listing all module descriptions in a file shows the

names and position of all modules; the user may then select an individual module either by its name or its position in the file. Deleting a module, as well as reading in a new module description file, will require a confirmation by the user as both operations destroy existing information. Saving the current description file will write all module descriptions to the existing or a new module description file.

The module description file generated by **SimDesc** is stored as a textfile so it can be printed out for documentation and debugging purposes. The file is line oriented, with comment lines indicated by the hash symbol ("#"); comment lines and blank lines in the file are ignored when the information in the file is interpreted by one of the other utility programs. The module description file is divided into variable length sections, each section describing one simulation module. To separate module descriptions from each other and to distinguish the different fields within each module description, a keyword scheme is used. A keyword is an uppercase string, preceded by a percent sign ("%") at the beginning of a line; Table 6.4 lists the currently defined keywords. This format was chosen as it is flexible enough to incorporate future extensions without problems (by defining additional fields and keywords), easy to read by humans, and still fairly easy to interpret for a machine.

The creation and modification date fields, as well as the author field, appear at the beginning of the module description file and are used for documentation purposes. Creation and modification dates are automatically maintained by **SimDesc**, while the author field has to filled in by the user.

The symbol table is an optional header section which is calculated by **SimDesc** when the module description file is saved to disk. Following the keyword **%SYMTAB**, the number of module entries in the symbol table is given, followed by a one-line entry for each module. Each line holds the name of the module in the description file and the start of the module's descriptive entry, measured both in lines from the %BOF marker and in bytes from the start of the file. This is intended to allow efficient use of module description files by the **SimBuild** and **SimMake** utilities, by allowing direct positioning to a desired entry (using the UNIX *lseek* system call), rather than scanning the entire file for a module's description.

The **%BOF** (beginning of file) and **%EOF** (end of file) keywords bracket the module descriptions in a file. The %BOF keyword also signals the end of the file's symbol table (if it exists) and is followed by the number of module descriptions in the file. Similarly, the **%MODBEG** (beginning of module) and **%MODEND** (end of module)

keywords bracket the descriptive entry for each module. The %MODBEG keyword is followed on the same line by the name of the module described in the following entry. The first few lines of the module descriptive entry list information such as the module's filename, its icon type (or icon filename, used by **SimBuild**) and similar information needed by MSS to access the other files that describe the module.

Keyword	Field Contents
%CDATE	Creation date of module description file
%MDATE	Last modification of description file
%AUTHOR	Author(s) of module descriptions
%SYMTAB	Start of description file symbol table
%BOF	Start of module descriptions
%EOF	End of module descriptions
%MODBEG	Start of a module description
%MODEND	End of a module description
%INPORT	Start of input port definitions
%OUTPORT	Start of output port definitions
%CONTROL	Start of control specifications
%MEASURE	Start of measurement specifications
%ATTRIB	Start of attribute definitions
%INIT	Start of module initialization definitions

Table 6.4: MDF Keywords

The information stored in the **%INPORT** and **%OUTPORT** sections specifies the number of input and output ports a module possesses and, for each port, lists its name

and type. This information is used by **SimBuild** to verify proper connections between modules and may also be used to label or otherwise mark the connection type on the graphical display of the simulator. The structure of the information in the **%CONTROL** and **%MEASURE** sections is as yet undefined, these fields await future expansion of the system's functionality.

The **%ATTRIB** section gives the number of attributes of a module which are user-configurable and describes each of them. The entry for each attribute consists of two lines; the first line lists the attribute's name, its type, and its default value. The second line lists either the lower and upper limits for the attribute, or the number of possible attribute values and their values. An attribute can be either a text string, an integer value, or a floating point value. The information listed in this section is used by **SimBuild** to display the module's attributes and to validate user input.

The **%INIT** section lists the number and values of the arguments that have to be passed to a module's *init_port* and *init_attr* functions to properly initialize the module. This section consists of two sub-sections: the first giving the number and names of the output ports, and the second listing the number and names of the user-configurable attributes. Some of the information in this section could be deduced using the preceding output port and attribute sections. It is nevertheless necessary to list them again here as, for example, the order in which output port destinations are passed to the *init_port* function may not be the same as the order in which the output ports are listed. Information in this section is used by **SimMake** to generate the correct module initialization calls.

This completes the discussion of the file format of the module description file. For the exact layout and definition of each component of the module description file, see Appendix B; a sample module description file can be found in Appendix C.

6.3.2. SimMake

The last step in the generation of a simulator is performed by the **SimMake** utility. It uses the information in the simulator description file (**SDF**) created by **SimBuild** to generate the main program of the simulator and a makefile which allows automatic compilation of the simulator by the UNIX *make* facility. To determine the characteristics of the modules used in the simulator, **SimMake** interprets the information in the module description files of the required modules.

The simulator description file uses a file format that is very similar in its layout to that of a module description file. However, while a module description file's entries describe a class of simulation modules, the entries in a simulator description file specify the instances of simulation modules which are actually used in a particular simulation. The simulator description file is stored as a textfile, so that it can be printed out for documentation and debugging purposes. It is divided into variable length sections delimited by keywords of the same format used for module description file. The keywords currently defined for the simulator description file are listed in Table 6.5 below.

Keyword	Field Contents
%BOF	Start of simulator description
%EOF	End of simulator description
%RUNTIME	Runtime parameters for simulator main program
%LIBRARY	Start of a simulation module library
%MODBEG	Start of a module instance specification
%MODEND	End of a module instance specification
%OUTPORT	Start of module output port specifications
%ATTRIB	Start of module attribute specifications

Table 6.5: SDF Keywords

As in a module description file, the keyword **%BOF** signals the beginning of the simulator description (there is no symbol table); it is followed by the number of instances of simulation modules in this simulator. **%EOF** marks the end of the simulator description; any information after the end of file marker is ignored.

The **%RUNTIME** section is intended to hold specifications to the **SimMake** utility on how to set up the simulator's main program. This will be used to select configuration options for the main program, for example, whether the program should prompt the user for the desired length of the simulator run or whether this value should be read from the command line. Eventually this section will be generated using a *Set runtime parameters*

menu in **SimBuild**. At the moment, this menu has not been implemented due to lack of pertinent experience with running simulators.

The section marked by the **%LIBRARY** keyword sets the currently selected module library. The keyword is followed by a line giving the filename of the module description file. This section is generated by **SimBuild** to indicate the library that is selected at the time an instance of one of the modules in that library is created. The library section marks the beginning of a set of module instance specifications whose descriptions can be found in the same module library. Any number of library specifications can be given in a simulator description for the same or different libraries, no ordering or sorting by libraries is assumed by **SimMake**. For efficiency reasons, however, modules from the same library should be grouped together

In a simulator description file, each section enclosed between a pair of **%MODBEG** and **%MODEND** keywords describes a specific instance of a module. The module identification number following the module's name on the %MODBEG line is generated by **SimBuild** to uniquely identify each module instance of a simulator. This number is used in the simulator description file for output port specifications and, at simulator runtime, for the default name given to modules. If another name is to be used as the name of the module instance, that name is given in the line following the %MODBEG line. This enables the simulator designer to specify descriptive names for modules (e.g. "Master Processor" instead of "CPU212") which will later be used in the reports generated by the simulator.

The **%OUTPORT** section describes a module instance's connections to other module instances as defined by the simulator designer in **SimBuild**. For each output port of the module, it lists the name of the output port and the module identification number of the receiving module. Similarly, the %ATTRIB section specifies the values for a module instance's attributes that the simulator designer selected in **SimBuild**.

SimMake is invoked from the UNIX command line with the filename of the simulator description file to be processed (it will prompt for the filename if it is not specified). **SimMake** will then proceed to read in and interpret the information in the simulator description file. Following this, it will read the required module descriptions from each of the specified module libraries. At this point, **SimMake** has all the information needed to generate the simulator program.

The source code of the main program of a simulator always follows a predictable scheme. First, it has to include a number of standard C++ header files with declarations of the classes provided by SAMOC and MSS and the header files with the class declarations for user-defined simulation modules. The main program proper, always called *main* in UNIX systems, starts out by declaring a pointer variable for each of the modules of which instances are created and then dynamically creates an instance of each module, saving its address in the associated pointer variable. It then has to initialize each module's output ports and attributes to the values specified in the simulator description file. The simulation is started by scheduling all simulation modules at the current time and by suspending the main program for the duration of the simulation. Once the main program regains control, it produces the customized report for each module and then terminates the simulation, which also produces a report on the statistics gathered by SAMOC.

For the generation of the simulator program, **SimMake** uses a set of pre-defined program fragments and then creates the additional statements necessary to construct a working simulator. The program fragments include standard text such as the one for the inclusion of header files described above and other standard parts of the simulator program; they are kept in a text file and are therefore easily editable. The names of the required pointer variables are synthesized using the module class name and the module instance's identification number; each module's class name (in uppercase) is used as the default name for the simulation module instance.

Once code for the creation of module instances has been generated, **SimMake** proceeds to generate the calls to each module's initialization routines. By matching the output port declaration from the module's description to the output name and destination module ID given in the simulator description and by utilizing the module pointers created earlier, **SimMake** generates the proper sequence of function parameters for each module instance's *init_port* function. A similar process, using the module description as a template which is filled in with the values given in the simulator description, then generates the appropriate calls to initialize each instance's attributes. Finally, **SimMake** generates the necessary calls to SAMOC to schedule all modules and the standard text which determines the duration of the simulation run, suspends the main program for the required time, and terminates the simulation. For an example of a simulation program generated by **SimMake**, see Appendix C.

In the next step, **SimMake** generates the makefile which will automatically compile the simulator. A <u>makefile</u> is a textfile which contains instructions interpreted by the UNIX

make utility which specify how to compile and link a program. Without going into too much detail, the instructions recognized by the make utility are dependency rules, transformation rules, and macro definitions. A dependency rule lists a *target*, i.e. a file that is to be generated, and its *dependencies*, the files required for the generation of the target file. Transformation rules list the commands that are used to generate a target file. There are two forms of transformation rules: default transformation rules based on file type, and explicit transformation rules in the form of a list of commands following a dependency rule which are to be executed to generate the target file. The default rules list the commands to be used to transform a file of a particular type (as determined by its filename extension) into one of another type, e.g. how to convert a C source code file (extension ".c") into an object file (extension ".o"). Macro definitions can be used to specify such items as default directories to search for include files, compiler flags to be used for all compilations, or the names of a group of related files.

The format of the rules contained in a makefile is well documented and lends itself well to automatic generation. As in the generation of the simulator source code, **SimMake** can make use of pre-defined fragments of text for standard parts of the makefile such as the pathnames of the directories in which the different source files reside or the default transformation rule for C++ source files. Using the names of the module header and source files given in each module description, **SimMake** generates a dependency rule which specifies the files on which the simulator depends and the commands used to compile and link an executable version of the simulator.

When **SimMake** has finished execution, it will have generated the C++ source code for the simulator described in the simulator description file and a makefile which will automatically compile the simulator. If no further changes are necessary to the simulator source code, the user can then invoke the UNIX *make* utility to generate the simulator.

6.4. Simulation Design Using MSS

As described in chapters three and five, each component of a system is modeled as a separate module which connects to other modules exclusively through message ports. The necessary port and message structures are defined by MSS and are encapsulated in the class *simmod*, which serves as the base class for all user-designed simulation modules. A simulation module designer constructs a new module by defining member variables and member functions which extend and specialize the functions of this base class to fit those of the modeled entity.

Creating a new simulation module involves a number of steps. Once the design of the new module has been completed, the design has to be implemented by creating a C++ class declaration for the new module class and a C++ source file which contains the implementation of the module's member functions. If the newly designed module uses a new message type, an appropriate message class has to be derived from the MSS-defined class msg and its definition should be included in the same source file as that of the simulation module using it.

Once the implementation of a module has been completed, its designer has to create a module description as the next step. Using **SimDesc**, the new module's description can be added to an existing module description file or a new description file can be created to start a new module library. To make the module available for use in the MSS system, the module's source file has to be compiled and the resulting object file has to be moved into the object library directory. If the module description has been added to an existing module description file, its object file should also be added to the existing module object library.

When modules for all the different components of a system are available, the simulator designer can then use those modules to create a simulator. Using **SimBuild**, a graphical representation of the simulator is interactively constructed by selecting the appropriate modules from one of the module libraries and by defining the necessary connections between them. As needed, the simulator designer may also change the default values of module attributes to adapt them for the simulated system. The simulator description resulting from the designer's work is then saved in the simulator description file generated by **SimBuild**.

The simulator designer next invokes the **SimMake** utility which, based on the information stored in the simulator description file and various module description files, will generate the simulator's main program and a makefile to compile the simulator. If necessary, the generated simulator program can then be modified by hand. This may be necessary, for example, to specify a module's stack size which differs from the default used by SAMOC. Finally, the UNIX *make* utility is used to compile the executable version of the simulator. Figure 6.4 illustrates the simulator creation process and shows the created files and their usage in the system.

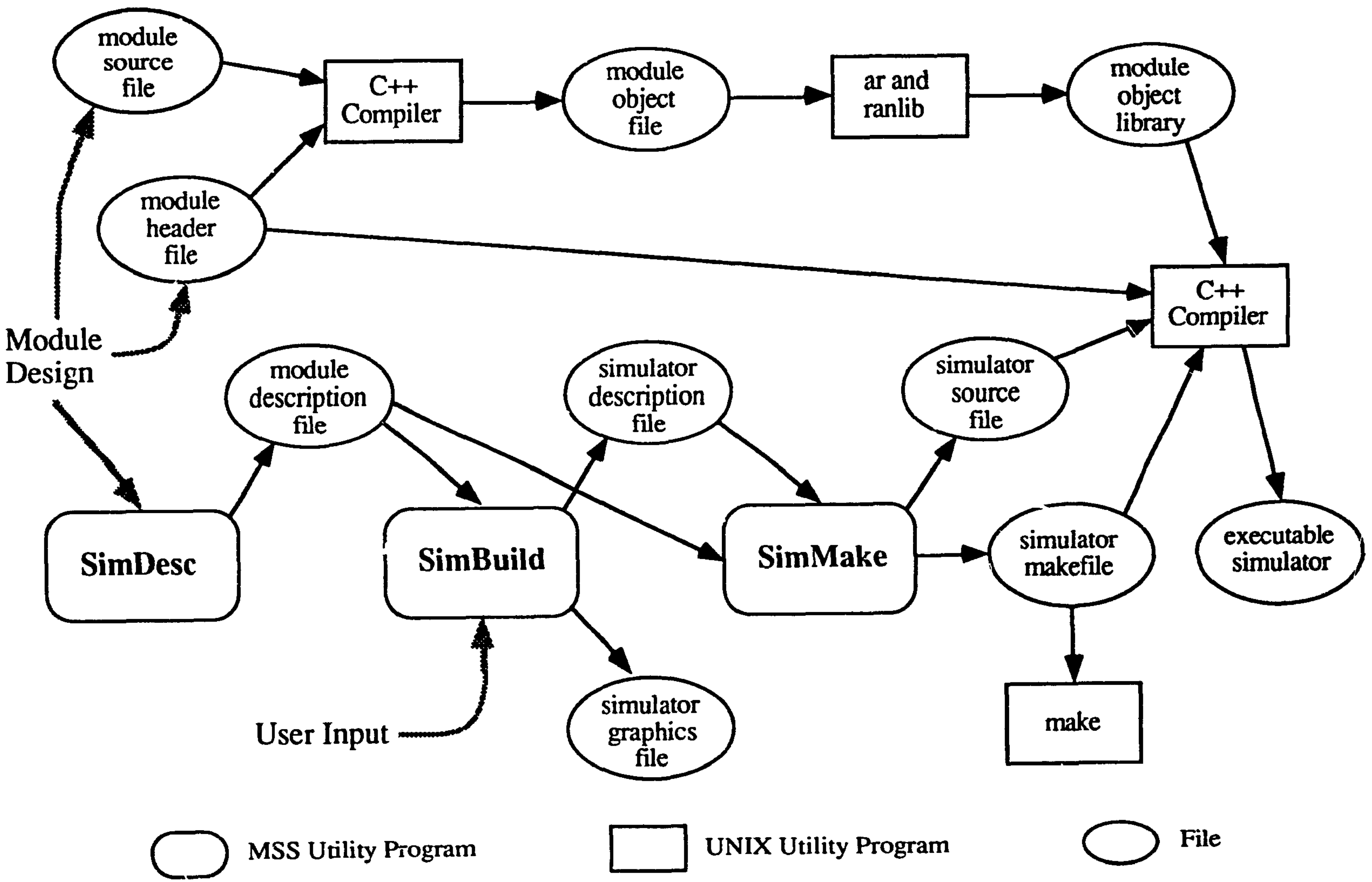

Figure 6.4: Simulator Creation Using MSS

7. The Development of a Sample Simulator

In this chapter we discuss the development of a sample simulator using MSS. This discussion serves the dual purpose of showing the process of simulator development in more detail, and of further elaborating on the documentation and coding guidelines and standards mentioned earlier.

We will show two iterations of the simulator development cycle described in chapter five. First, we develop a very simple simulator with fixed, non-random events to demonstrate the basic design process. In the second iteration, we expand the existing simulation modules to include random events and further refine their operation; we then use these modules to generate a more interesting simulator. The full text of the files created during this sample session is listed in Appendices C and D.

7.1. Development of the Simulation Modules

To generate a simulator, we first have to create the necessary simulation modules. For this example, we assume that no modules other than the base modules supplied with MSS exist. The simulator we want to develop is for a very simple computer system similar to the one used as an example in chapter six. Our first simulator will consist of just three modules: a central processor module, a memory module, and a bus module which connects the processor to the memory (see Figure 7.1). The development of these three modules is described in the following sections.

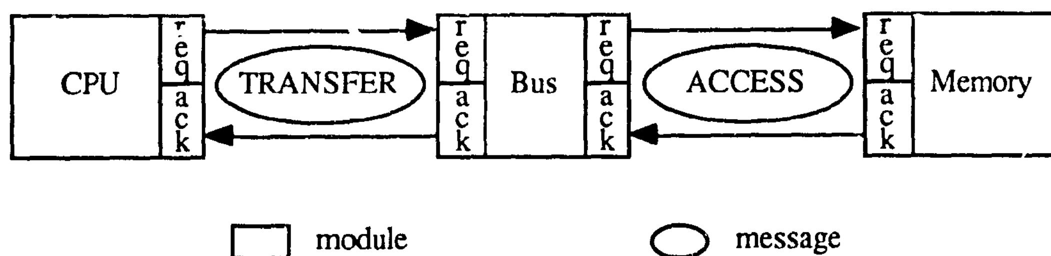

Figure 7.1: Example Simulator

7.1.1. Messages

As can be seen in Figure 7.1, there are two message types used in the simulator, *transfer* and *access*, both of which use the request/acknowledge protocol described in chapter six. A *transfer* message is used by a CPU module to request the transfer of a number of bytes to or from a memory module. The number of bytes to transfer is

determined by the CPU module and may vary. An *access* message is used by the bus module to transfer one word of data to or from the memory, where the word size is determined by the width of the bus.

The MSS base class for messages, ***msg***, already provides fields for the message type and for the identification of sender and destination modules. To define transfer and access messages, we need to expand this base class to include the additional message fields required and we have to add functions to set these additional fields. Both *transfer* and *access* messages are used to read data from or write data to the destination module. *Transfer* messages additionally specify the memory module address and the length of the data to transfer.

```
//***************************************
//  access message: Bus <-> Memory
//   - access request:     Bus -> Memory
//   - access acknowledge: Memory -> Bus
const msg_type  ACCESS = 1;

class acc_msg : public msg {
public:
  boolean   Read;    // Read (T) / Write (F) access
  // initialize an access request
  void  init_acc(Mod_ID send, Mod_ID dest,
                boolean rd=TRUE);
  acc_msg(msg_type mt=ACCESS, int pr=0) : (mt,pr) {}
};

//***************************************
//  transfer message: CPU <-> Bus
//  - transfer request:     CPU -> Bus
//  - transfer acknowledge: Bus -> CPU
const msg_type  TRANSFER = 2;

class tx_msg : public acc_msg {
public:
  long      Adr;    // start address of memory access
  long      Length; // number of bits to transfer
  // initialize a transfer request
  void  init_tx(int send, int dest, long len,
               boolean rd=TRUE, long adr=0);
  tx_msg(msg_type mt=TRANSFER, int pr=0) : (mt,pr) {}
};
```

Listing 7.1: Message Class Declarations

We therefore design a class ***acc_msg***, derived from the base class ***msg***, and a class ***tx_msg***, derived from ***acc_msg***. ***Acc_msg*** implements access messages, expanding ***msg*** objects by a field *Read* that flags read or write access. Class ***tx_msg***

implements transfer messages and expands ***acc_msg*** by two fields *Adr* and *Len* for the address and length of the data to transfer (see Listing 7.1). To avoid the confusion that often occurs because of varying definitions of data sizes, the length of transferred data is always given in bits; this also is advantageous for the communication with modules which simulate serial or network devices. We also define two constants **ACCESS** and **TRANSFER** which are used to indicate each message's type.

As Listing 7.1 shows, the additional message fields have been declared as public member variables, they are thus accessible to all other classes. The alternative would be to declare them as private (or protected) variables, in which case we would have to provide member functions to access and set these variables from another class. Either approach is valid, with the first one involving less coding and the second one being more purely object-oriented.

Both classes also provide a method to initialize a message instance, providing appropriate defaults where possible. As both message types use a request/acknowledge protocol, only one instance needs to be created for each originating module, which is reinitialized with the new values every time it is sent. No default is provided for the sender and destination fields to force the sending module to specify these two fields; similarly, there is no default value for the length of a TRANSFER message. Default values are provided for the remaining arguments of the initialization functions, however. The arguments for each function are arranged to put those most likely to remain at their default value at the end of the argument list to make their specification unnecessary for most function calls.

As the full listing (in Appendix C) shows, the initialization functions were implemented as inline functions to reduce the overhead incurred for using them. In general, we try to use the C++ inline facility for often-used calls to speed up the simulation wherever possible. In our case, as all member functions are declared inline, there are no source or object file corresponding to the header file declaring the message types.

The file **msgs.hxx**, which contains the declaration of the two message types used in our example simulation, also illustrates some of the additional coding guidelines mentioned earlier. Thus it starts with a one-line comment giving the name of the file and a short description of the file's contents. As many files may exist in a large simulation system, this convention enables a simulation designer to quickly gain an overview of the different modules present in the system.

All include files used in MSS define a symbol based on their filename, i.e. the filename in all uppercase letters, preceded and followed by an underscore character. This symbol is used to allow conditional inclusion of the file by the C++ preprocessor, depending on whether it has been previously included during the compilation or not. Not only does this speed up the compilation process by preventing unnecessary file accesses, it also is necessary to avoid some hard-to-trace compile and runtime errors which can occur in a C++ program due to multiple inclusion of class declarations with inline functions.

Each class declaration for a message type should be preceded by a comment field which details the use of messages of this type. This comment should state the intended use for each field of the message, the type of the sending and receiving modules, and any special processing requirements to be provided by those simulation modules. Any further assumptions and information about how this message should be used are also stated in this comment.

7.1.2. CPU Module

The CPU module is based on the basic, well-known processor model in which the processor fetches and decodes an instruction, fetches the necessary operands, and executes the instruction. It is implemented in class ***cpu***, which is derived from MSS base class ***simmod*** and declared in file **cpu1.hxx**; the source code for its member functions is stored in file **cpu1.cxx**.

As described above, file **cpu1.hxx** begins with a one-line header stating its name and contents (see Listing 7.2 below). Following that, we include the header files for class ***simmod*** and for the message classes used in the simulation. The declaration of a simulation module is preceded by a comment block describing the module's attributes and characteristics. This comment should follow the format given in Table 7.1 to enable users to get a quick and systematic overview of the module.

After stating the function of the module, e.g. that this is a simple CPU module, the comment field gives the <u>time base</u> used by this module. SAMOC provides a type *sim_time* which represents the simulation time; this is a double floating point number restricted to ten significant digits. While the interpretation of the time unit represented by the simulation time is up to the simulation program, all modules in a simulator must share the same interpretation to work together correctly. In our case, by choosing a simulation time unit of one nanosecond, the shortest time separating two events which do not occur simultaneously is

one nanosecond, while the maximum runtime of the simulator is approximately 10 seconds (10,000,000,000 ns). This choice of a time base is appropriate for the intended simulation.

The next component of the header comment for class ***cpu*** lists the user-configurable attributes of the module. In our case, we have only one attribute, the cycle time of the CPU clock. We also state the default value used for this parameter; this value can be changed using **SimBuild**.

Component	Description
Function	Module function
Time Base	Module time base
Attributes	User-configurable attributes
Characteristics	Assumptions and limitations of this module implementation
Statistics	Additional statistics gathered and reports generated by the module
Simulation Model	Description of simulation model this module is based on

Table 7.1: Components of Module Header Comment

Next, we describe the characteristics of this implementation of the CPU module. This section lists the assumptions made in the design of the module. As ours is a very simple design, all instructions are of a fixed size of one word (16 bits) and have a constant execution time of four clock cycles. Similarly, each instruction only has one 16-bit operand and no data is written to memory (i.e. all operands are read-only).

The two remaining components of the header comment should list any additional statistics generated by the module and describe the simulation model on which the module is based. In our case, no statistics other than the ones are gathered automatically by SAMOC are provided. The description of the simulation model just lists the four steps of the CPU instruction execution cycle (instruction fetch, instruction decode, operand fetch, instruction execution).

Similar to the header comment, the declarations in a simulation module class should also follow a certain standard scheme (see Table 7.2). We mark the beginning of each of the different declaration sections with an appropriate comment to visually separate it from the preceding fields.

Section	Description
Ports	Module output ports
Messages	Message objects for messages originating from this module
Attributes	User-configurable module attributes
Distributions	SAMOC distribution objects used by module
Statistics	SAMOC statistics objects used by module
Internals	Other variables used by the module implementation
Required Functions	Member functions required by the module interface
Optional Functions	Optional member functions

Table 7.2: Components of a Module Declaration

Our first concern is to define the necessary output ports and messages; in this simple module, we only have one output port, which leads to the bus module. As described in chapter six, output ports are implemented as pointers to the destination module. Each output port pointer is declared as a pointer to an object of the base class ***simmod*** so it will be assignment-compatible with all possible modules derived from that class. For each type of message which is sent or received by a module, we need to create a pointer to reference the message object. For those messages originating from a module, we also need to create an instance of the appropriate message object; this is done in the module's constructor.

Module attributes are declared next, followed by the declarations of the distribution objects used by the module to simulate randomness in its behavior. In the case of the simple CPU module, we have only three constant distributions, one each for instruction size and execution time, and for operand size. Even though it would have been simpler to

use constants for this purpose, we used the SAMOC constant distribution objects as we will later refine this module to include random behavior. Any SAMOC statistics objects used by the module to generate customized reports and member variables used in the implementation of the module are declared in the following two sections. In our module, there are none.

```
//*********************
//  Simple CPU module  *
//*********************
//
// Time Base: 1 time unit = 1 nanosecond
//
// Module Attributes:
//   - CPU clock cycle time (default: 100 ns)
//
// Module Characteristics:
//   - fixed instruction execution time (4 cycles)
//   - fixed instruction size (1 word)
//   - fixed operand size (1 word)
//   - all read, no write transfers
//
// Additional Statistics: none
//
// Simulation model:
//      1. Instruction fetch
//      2. Instruction decode
//      3. Operand fetch
//      4. Instruction execute
//
class cpu : public simmod {
// PORTS
  simmod    *Bus_port;   // pointer to bus module
// MESSAGES
  tx_msg    *Tx_msg;     // pointer to transfer msg
// ATTRIBUTES
  sim_time  Cycle_time;  // CPU clock cycle time
// DISTRIBUTIONS
  i_const   *I_time;     // instruction execution time
  i_const   *I_size;     // instruction size (words)
  i_const   *O_size;     // operand size (in words)
public:
  cpu();
  ~cpu();
  void  init_port(mod_ID id, simmod *bp);
  void  init_attr(sim_time ct=100.0);
  void  body();
};
NEW (cpu, 1024);
```

Listing 7.2: Declaration of Simple CPU Module

The module's interface, given in the public part of its declaration, consists of the functions required by the interface defined by class ***simmod*** and of optional functions which may be implemented by the module's designer. The constructors and destructors of a simulation module dynamically create and de-allocate the objects declared earlier, such as messages and distributions. The member functions *init_port* and *init_attr*, are used to initialize a module's output ports and attributes; they are typically implemented as inline functions. If the module collects additional statistics for a custom report, a function *report* must be declared as one of the optional functions. Other optional functions may be defined by the module designer as needed. Also note the statement immediately following the class declaration, which defines a macro called **new_cpu**; a macro of this form is needed for all classes derived from SAMOC class ***entity*** and replaces the standard C++ function new for those classes. This macro is used to set up the private stack for each ***entity*** object.

The function *body* determines a module's runtime behavior and its implementation is kept in a module's source file. A shortened listing of the implementation of class ***cpu*** is given below in Listing 7.3). The complete implementation is given in file **cpu1.cxx** and in Appendix C. In general, we can expect the body to consist of an endless loop inside which all actions of a module take place. Unless the module ceases to exist during the simulation runtime, it never exits this loop.

```
void cpu::body()
{
  while ( TRUE ) {
    //*** I-fetch
    Tx_msg->init_trans(ID,UNKNOWN,16*I_size->sample());
    Tx_msg = (tx_msg*) reqack(Bus_port, Tx_msg);
    //*** I-decode -- no time used
    //*** O-fetch
    Tx_msg->init_trans(ID,UNKNOWN,16*O_size->sample());
    Tx_msg = (tx_msg*) reqack(Bus_port, Tx_msg);
    //*** I-execute
    hold(Cycle_time*I_time->sample());
  }
}
```

Listing 7.3: Implementation of Simple CPU Module

As can be seen from the listing, the CPU module is a straight-forward implementation of the four-phase CPU model described earlier; each pass through the loop executes one instruction. To fetch an instruction, we initialize the transfer message to transfer 16 bits and send it to the bus module; the module then waits for the return of the message from the bus module (we have omitted the code which does error checking to verify that the

correct message has been received). We then repeat this process to fetch the instruction's operand. Note that we do not need to specify the memory address or read/write mode as they are supplied by default values. As we do not know the ID number of the destination module, we supply the value **UNKNOWN**, which is defined by MSS; this ensures that trying to use the value as a module ID will result in a runtime error (the sender and destination ID fields are used by the bus module in a specific manner as described later). Finally, we execute the instruction, modeled by suspending the CPU module for the required number of clock cycles.

7.1.3. Bus Module

The task of the bus module is to simulate the transfer of data between the CPU and memory modules. The bus thus has two input and two output ports, one each to and from each type of module. As mentioned earlier, we assume in this simple model of a computer system that all data transfers are of a fixed size of 16 bits; each transfer request from the CPU results therefore in exactly one memory access request (this may not always be so, as we shall see later). The only attribute for the bus module is the bus cycle time, which determines how long it takes to transfer one data item.

The bus module's activity can be divided into three stages: waiting for an incoming message, which may be either a *transfer request* or an *access acknowledge*, processing the message, and transferring the data. A bus cycle is started by the reception of a *transfer request* message from a CPU module; if another transfer is currently in progress, the request will be queued, otherwise an access request message is sent to the memory module. Once an *access acknowledgement* has been received from the memory module, the transfer request is acknowledged and sent back to the CPU module. Data transfer is simulated by suspending the bus module before sending or after receiving the memory access message, depending on whether a write or read access has been requested.

For the simple simulation described in Figure 7.1 it is impossible to receive a *transfer request* while an earlier one is still being served. We provide for this possibility anyway, as we may later wish to construct a simulator which uses two CPU modules connected to the same bus and memory.

The bus module initializes the destination field of the memory access message with the ID number of the CPU module that sent the transfer request. This ID number is then returned in the same field in the *access acknowledgement* and can be used to determine the identity of the CPU module which issued the transfer request. We can use this to verify the

validity of the received message and may use it later on to implement a bus module which can deal with several pending *transfer requests*. A listing of the bus module can be found in Appendix C.

7.1.4. Memory Module

The memory module also follows a fairly simple operational model. Each memory access request received from the bus results in a delay as determined by the memory module's access time; after the delay has elapsed, an *access acknowledge* message is returned to the bus to indicate the end of the memory access cycle. Only one access request can be active at a time.

As the implementation of the memory module is straight-forward, it will not be further discussed; the listing of its header and source files (**mem1.hxx** and **mem1.cxx**) is given in Appendix C.

7.2. Generation of the Module Library

At this point, we have generated the simulation modules which are needed for the simple simulator described above. To implement the modules, we have used the base classes provided by MSS as a starting point and have extended their functionality as needed. We now have to create a description of each module in a form that can be used by the simulation system, and we have to combine the created modules into a simulation module library. These two steps are described in the following two subsections.

7.2.1. Creation of the Module Description File

To create the module description file (or MDF), we invoke the **SimDesc** program; if we do not specify a filename on the command line, we will create a new module library. To obtain a list of the available commands, we can type a question mark at **SimDesc**'s command level prompt (see Listing 7.4; user input is printed in **boldface**; "%" is the UNIX command prompt); to verify that the module library is indeed empty, we can type "**l**" to obtain a list of the defined modules.

We now proceed by adding the necessary descriptions for our three modules; the order in which the descriptions appear in the file is of no importance. After choosing to add a module description, **SimDesc** will prompt for the required information in the order required by the MDF file format.

```
% simdesc
SimDesc Version 1.1 (15-Mar-90)
Enter '?' for help
==> ?
Valid  module commands:
         a        add a module description
         d        delete a module description
         e        edit a module description
         l        list modules in library
         s        show a module description
         r        read  a module library file
         w        write a module library file
         h        help (this text)
         q        quit SimDesc
==> l
Current library file: <none>
No modules in library
==>
```

Listing 7.4: **SimBuild** Command Level Prompt

The module description and each of its major components may be preceded by a comment block. The comment preceding the formal module description should give a short description of the module and its features. The comment blocks preceding each component of the module description should document features which are not immediately obvious from the formal description. These comments are ignored by the MSS utilities, but may be useful as additional module documentation. Listing 7.5 shows the actual dialogue generated by **SimDesc** that resulted from the addition of the CPU module to the library (ellipses "..." indicate repetitive parts of the dialogue which have been omitted).

```
=> a
Enter an empty input line to abort
Class name of module > cpu
Filename of module files > cpul
Valid module icon types are: B, C, D, G, K, M, N, R, S
Module icon type (or filename)> C
Enter text for comment block (max. 80 chars);
end input with empty line
> Simple CPU module with non-random events
> and fixed instruction & operand sizes
> Time base is 1 nanosecond
>

Enter INPUT PORTS (max. 10); empty line to exit.
Enter text for comment block ...
> 1 logical input port:
> tx ack from bus, message type: TRANSFER
```

(Listing 7.5 continued)

```
Port NAME and TYPE may be entered on the same line. Port name
can be any name which indicates the port's function (e.g..
message type and protocol used).  Port type must be the class
name of the message type for this port.
Port name > tx_ack
Port type > tx_msg
Port name >

Enter OUTPUT PORTS (max. 10) ...
Enter text for comment block ...
> 1 logical output port:
> tx_req to bus, message type: TRANSFER
>
Port name and type may be entered on the same line ...
Port name > tx_req tx_msg
Port name >

MODULE CONTROL
Enter text for comment block ...
> not implemented
>

MODULE MEASUREMENT
Enter text for comment block ...
> not implemented
>

MODULE ATTRIBUTES
Enter text for comment block ...
> 1 attribute:
> Cycle_time   CPU cycle time;
>                  default: 100ns, range: 25 ns-250 ns
>
Attribute NAME, TYPE, and DEFAULT value MAY be entered on the
same line.  Attribute values MUST be entered on the same
line, separated by blanks.  Valid attribute types are: F
(floating point), I (integer), T (string). Attribute value
may be min..max range of values, or a selection of values
preceded by a count (-1 for range).
Attribute name    > Cycle_time F 100.0
Attribute values > -1 25.0 250.0
Attribute name    >

MODULE INITIALIZATION
Enter text for comment block ...
>
Enter number of output ports to initialize > 1
Enter the name of each output port, in the same order as the
arguments for the module's init_port function.
Port name > tx_req
```

(Listing 7.5 continued)

```
Enter number of attributes to initialize > 1
Enter the name of each attribute, in the same order as the
arguments for the module's init_attr function.
Attribute name > Cycle_time
Keep module (y) or delete it (n) ? y
```

Listing 7.5: Adding a Module Description using **SimDesc**

In the same way, we also add descriptions for the bus and memory modules to the library file. To verify that the description has been accepted and that it gives the correct information, we may list simply the names of the modules in the module library (see Listing 7.6) or list all of the module description for a particular module. We may also edit the description for an existing module once it has been entered.

Once we are satisfied with the module descriptions, we save the library to a file and quit **SimDesc**. As we were creating a new library, **SimDesc** will prompt for the filename before writing the module library to disk (see Listing 7.6). Note that we only need to give the basename of the library file, the proper path and filename extension will automatically be generated by **SimDesc**; for verification, they will also be displayed on the screen. For documentation, **SimDesc** will also ask for the name of the library's author. The complete text of the generated module description file is listed in Appendix C.

```
==> l
Current library file: <none>
Mod#    Module Name
-----------------------
1       cpu
2       bus
3       mem
==> w
Enter name of module library file > Example1
Current module library file: .../MSS/mdf/Example1.mdf
Change filename (y/n) ? n
Enter author's name > Thomas Wieland
Writing module library .../MSS/mdf/Example1.mdf...
==> q
```

Listing 7.6: Saving the Module Library

7.2.2. Creation of the Module Object Library

The MDF file created in the previous step describes the modules in a library called **Example1**. To be able to use these modules in a simulator, we also have to compile the module definitions generated earlier and store them in an object library.

To compile the modules, a makefile is provided by MSS in the module source directory. To see the commands provided by the makefile or if we are not sure how to use it, we can invoke the make facility with the argument *help* to get a list of available commands (see Listing 7.7).

```
% make help
make MOD=<name>   - make simulation module <name>
make library LIB=<libname> "OBJS=<object names>.."
                  - make & install library <libname>
make mss.a        - make & install MSS library
make clean        - clean out garbage files
make backup       - backup simulation modules
```

Listing 7.7: Commands for the Module Makefile

The makefile provides a convenient way to compile simulation modules into objects as it automates the necessary steps as much as possible. For example, to compile the CPU module developed earlier, all we need to do is to type **make MOD=cpu1** and the UNIX *make* utility will issue the commands that generate the module object file **cpu1.o**. In the same way, we also generate the objects for the bus and memory modules.

Once the module object files have been generated, we then have to combine them into an object library for use by the linker. Issuing the appropriate make command will do this for us (Listing 7.8); note that we have to specify the names of the objects to include in the library as we cannot assume that all objects in this directory are to be included into the library. The *make* facility echoes to the screen the commands used to generate the library so the user can monitor its progress. The *ranlib* program invoked by the makefile creates a symbol table and prepends it to the library; this symbol table is used by the linker to satisfy external references in a simulation program.

```
% make library LIB=Example1 OBJS=*1.o
/bin/rm -f Example1.a
/bin/ar cvr Example1.a *1.o
a - bus1.o
a - cpu1.o
a - mem1.o
/usr/bin/ranlib Example1.a
/bin/mv Example1.a .../MSS/mdf
```

Listing 7.8: Generation of the Object Library

As its last step, the *make* script moves the generated object library into the correct directory so that other MSS utilities can find it. After the library has been created, we may then remove the separate module objects by invoking the command **make clean**.

7.3. Generation of the Simulator Description

We are now ready to generate a description of the simulator; as described in chapter five, we use the **SimBuild** program to do this. The operation of this program has been described in detail in that chapter, so we will not repeat it here.

We select **Example1** as the module library to use and create one instance each of the CPU, bus, and memory modules. We then link the CPU and bus modules by connecting the CPU's *transfer request* output port (**tx_req**) to the bus module's **tx_req** input port and the bus module's *transfer acknowledge* (**tx_ack**) output port to the CPU's **tx_ack** input port. Similarly, we connect the *access request* and *acknowledge* ports on the memory module to those of the bus module. If we want to select a different value for any of the modules' attributes, we can do so by clicking on that module's icon on the display.

Finally, we save the resulting simulator description file in **Example1.sdf** (see Appendix C for a listing of this file) and quit **SimBuild**. Figure 7.2 shows what a typical display of the simulator would look like on the final display in **SimBuild's** simulator window.

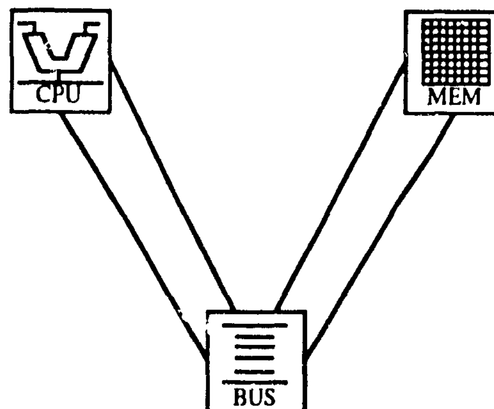

Figure 7.2: Display of the Generated Simulator

7.4. Generation of the Simulator

We now have created all the components needed to generate the simulator program. To create it, we invoke the **SimMake** utility program with **Example1** as the simulator description file to use as input. **SimMake** will read the SDF file and the module libraries

specified and will generate the appropriate simulator main program (**Example1.cxx**) and a makefile (**Example1.make**) to compile it (see Listing 7.9).

```
% simmake Example1
SimMake Version 1.1 (15-Mar-90)
Input file: .../MSS/sim/Example1.sdf
Reading SDF file...finished
Reading module library .../MSS/sdf/Example1.sdf...finished
Checking modules...all modules defined
Should the simulator P(rompt for the simulation length, read
it from the C(ommand line, N(one, or B(oth ?
Please enter your choice now (p/c/n/b) => b
Should the simulator allow the enabling of
SAMOC trace levels from the command line (y/n)?
Please enter your choice now (y/n) => y
Generating main program .../MSS/sim/Example1.cxx..finished
Generating make file .../MSS/sim/Example1.make...finished
Simulator program generated successfully, 0 errors
```

Listing 7.9: Generating the Simulator

As shown in the listing above, **SimMake** will prompt the user for the method to be used for specifying the length of the simulation run. There are four possible choices:

(i) *prompt* for the duration at simulator runtime
(ii) *read* the duration from the *command line*
(iii) generate code for *both* (i) and (ii)
(iv) do not set a simulation length

By entering the desired length of the simulator run on the command line (choice (ii)) or entering it when prompted by the simulator (choice (i)), we can run the simulator for varying lengths of time. Choice (iii) allows either interactive or command line driven operation of the simulator at the discretion of the user. In the case of choice (iv), **SimMake** will generate a simulator in which the simulation main program will be suspended indefinitely. The simulator designer must then edit the generated simulator main program manually to insert the desired termination conditions. If the simulator is set up to interpret command line arguments (case (ii) or (iii)), **SimMake** can generate a simulator which allows the enabling of specific SAMOC trace levels (see chapter four) from the command line.

As the final step, we compile the simulator using the makefile generated by **SimMake**. As the UNIX *make* facility by default uses the files **makefile** or **Makefile** as its input, we have to specify the name of the simulator's makefile on the command line like this: `make -f Example1.make`. After the compilation has finished, the file

Example1 holds the executable simulator. This file is automatically moved to the MSS *bin* directory.

7.5. Execution of the Simulator

To run the simulator, we just type the name of the executable file generated in the previous step. Note that for this to work, the MSS *bin* directory must be included in the UNIX shell's list of directories to search for commands.

The results obtained by running the simulator created above are given in Listing 7.10. This is the standard report generated by SAMOC; for each of the SAMOC objects used in the simulation, it lists the statistics that are collected by default by this kind of object. The generated simulator uses the default values for the attributes of all simulation modules, i.e. 100 nanoseconds for both the CPU and bus clock cycle times and 80 nanoseconds for the memory access time. Additional results for some simulator runs with varying module parameters are shown in Appendix C.

```
% Example1
Enter simulation length (microseconds) => 1000
Running simulation for 1000.000000 microseconds

                        clock time = 1000000.000

                                  BINS
                                  ****
title        / reset at/ users/init/ max/ now/ av. free/ av. wait/qmax
CPU 0             0.000    2632     0    1    0     0.000   180.000    1
BUS 0             0.000    5264     0    1    0     0.000   139.924    1
MEM 0             0.000    2632     0    1    0     0.000   299.810    1

                             DISTRIBUTIONS
                             *************
title        /   obs/type      /parameters
I_time          1316 i_const    constant =          4
I_size          1316 i_const    constant =          1
O_size          1316 i_const    constant =          1
```

Listing 7.10: Output of the Simulator

As we can see, each of the distribution objects has been accessed 1316 times, each one once for each instruction executed. Our simple computer model therefore executes instructions at a rate of 1.316 million instructions per second (MIPS).

The part of the report which gives us more information about what happened during the simulation is that for the *bin* objects labeled **CPU 0**, **BUS 0**, and **MEM 0**. These

are the bins (member variable ***inqueue*** of class ***simmod***, see chapter six) used to synchronize each module's execution with the incoming messages. The *users* column lists how many times each ***bin***'s *take* function was called; in other words, it shows how many messages were received by the module. As each instruction processed by the CPU causes two memory accesses, the CPU and memory modules list twice as many users as instructions were executed. Each data transfer over the bus results in two messages being exchanged; thus the bus module lists twice again as many uses of the *take* function as it has to process both the request and acknowledgement for each type of message.

The *init*, *max*, and *now* columns list the initial, maximum, and current token (i.e. message) counts for the bin; the *average free* column shows the average time a token spent in the bin. As only one message is circulating in this simple model, there is a maximum of one message in the bin at all times; each message is therefore processed immediately when it arrives, so its wait time in the input queue is always zero. The *qmax* column lists the maximum number of entities waiting in each ***bin***'s queue, which is always one as only the simulation module itself can wait for messages in its input queue. The *average wait* column gives the time each module spent on the average waiting for messages. For the CPU module, this time can be interpreted as the average duration of a data transfer from memory, i.e. the overall system memory access time. For the bus and memory modules, this time represents the average idle time between periods of activity.

By comparing the reported results with the simulator's module parameters, we can verify that our simulator operates correctly. For example, each data transfer should be delayed by one bus cycle time and one memory access time, or 180 nanoseconds. As this matches the CPU module's *average wait* time, the simulator functions correctly in this respect. By diagramming the activities of each of the three models during several data transfers and averaging their respective wait times, we can verify that the other results are correct, too.

As can be seen from this example, the output generated by the simulator needs quite an interpretation effort to arrive at some meaningful results. Other interesting characteristics of the system, such as the amount of time the bus is busy, cannot be seen directly from the given report and may have to be deduced indirectly, where possible. In the following section, we will refine our modules further to provide such details.

7.6. Refinement of the Simulator

As could be seen in the previous sections, there are quite a few limitations in the simulation modules we have developed so far. This level of sophistication (or lack thereof) is typical for a first attempt at constructing a simulator, in which the goal is to develop a set of base modules and verify their correct operation, rather than to get results that are immediately meaningful for a real system. The next steps in the simulator development cycle would be to return to the specification phase and incrementally incorporate more of the necessary details into each module. As an example of how the existing modules can be expanded, we will perform one more iteration of the development process.

In this iteration, we want to achieve two goals: first, we want to increase the functionality of the modules generated in the initial development cycle, and, second, we want to experiment with modeling a more sophisticated computer system. The changes to the simulation modules will be described in the following paragraphs. The system we want to model this time is shown in Figure 7.3; this system incorporates two processor modules, so in this model we will have contention for the bus. Also note the new *router* module which is now necessary; its purpose will be described in section 7.6.2. below.

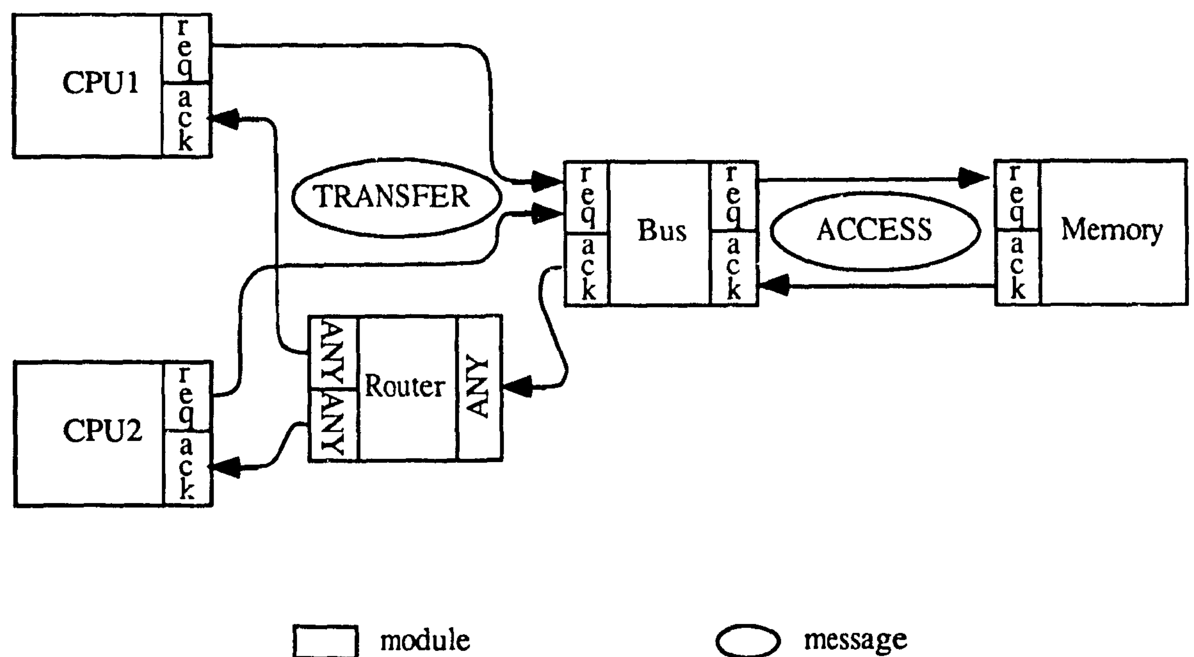

Figure 7.3: Expanded Example Simulator

7.6.1. Enhancing the Simulation Modules

The changes to the CPU module introduce variable length instructions and operands as well as variable instruction execution times to the existing module. As we have already used the SAMOC distribution objects in the existing code, most of the changes required to accomplish this are to modify the declaration of those objects and adapt the calls to their constructors. The parameters used for the distributions are patterned loosely after the Motorola 68020 CPU [MC68020 1985]. Another addition involves the possible generation of a write transfer as the result of executing an instruction for a certain percentage of instructions; this models writing a result operand back to memory. To allow easier interpretation of the simulation results, the CPU module also includes a custom report which prints the number of instructions executed, their average duration, and a MIPS rating of the CPU.

In the existing bus module's operation we assumed that each data transfer request from the CPU would generate exactly one memory access request; we also assumed a bus size which was the same as the size of the data to transfer. We now add a new attribute to the bus model which allows the simulator designer to specify the width of the data path as either 8, 16, or 32 bits. Transfer requests from the CPU thus do not necessarily have the same size as the data that can be transferred in one bus cycle. We must therefore update the logic handling a *transfer request* message to generate the required number of memory access requests for each transfer request. We simulate the transfer of data as before, but generate a *transfer acknowledge* only after the last memory access.

The bus module now also keeps track of the amount of time it is busy (i.e. a data transfer is in progress) and idle; by reporting this statistic using the module's optional custom report facility, we can measure the impact that different bus widths or the number of CPU modules have on bus utilization.

As in the previous example, the memory module remains the most simple of the three modules. The only addition is the modeling of the memory's cycle time, i.e. of the time a memory array needs after a previous memory access to be ready for another access. As we now have two processors contending for memory access, which may lead to simultaneous memory access requests, we must include the cycle time in the memory model to properly reflect the characteristics of existing memory components. We model the cycle time by calculating the earliest time that the next access is possible whenever the module

receives an *access request*. If another *access request* is received before that time has elapsed, it will be delayed until its cycle time requirement has been fulfilled.

For a listing of the enhanced simulation modules, see Appendix D.

7.6.2. Router Module

As shown in Figure 7.3, a router module is inserted into the message path between the bus and the two CPU modules. Looking back at the code used to implement our example simulation modules, we notice that each output port has been implemented as a single pointer to the destination module. We have therefore assumed that for each output port there is only one possible destination.

Naturally, if the outgoing messages can go to two or more destination modules, we need that number of destination pointers in each module. This requirement, however, runs contrary to the modularity that we want to achieve for each simulation module. We cannot *a priori* determine the number of destinations that each module may have to send the output message to at the time that we design it. We could try to determine a "reasonable" number of destination modules for each output port and implement that number of possible connections in the module. This has the disadvantage of introducing additional complexity into each and every module for managing multiple destination pointers and handling message routing. Furthermore, these added features would likely go unused for a majority of simulation modules, while still not providing all the necessary destination pointers for some of them.

Our solution to this problem is to provide a specialized router module which processes an incoming message stream and routes each message to the module given in the message's destination field. Thus, all modules still have only one destination pointer associated with each output port and all messages of that type are sent through this port. If a situation arises in which messages need to be routed to two or more destinations, a router is inserted into the message stream. By providing an explicit router module, MSS also imposes no limit on the number of possible destinations for a message as any desired "fan-out" can be achieved by cascading several routers.

The router modules provided in the MSS library have fan-outs of two, three, and five (classes ***router2***, ***router3***, and ***router5***, respectively), i.e. they can route messages to that number of different destinations. The destination module pointers and ID numbers that each router needs to accomplish its task are provided using its ***init_port*** and ***init_attr***

functions when the module is initialized. The only action of a router is to wait for an incoming message, determine its destination, and forward it to the specified module. The router does not test the type of the message sent and can therefore be used to route any type of message; it can even be used to route different message types to two or more different sets of destination modules. The routing process does not consume any simulation time as the router does not model any physical component of the simulated system, but is an artifact introduced by MSS. The header and source code for the router modules are listed in Appendix A.

Figure 7.4 shows a possible use of a five-way router in a simulator with four modules (only the relevant input and output ports are shown). The router module is used to forward messages of type "1" from the modules A and B to modules C and D. Similarly, it routes messages of type "2" from module A to any of modules B, C, and D.

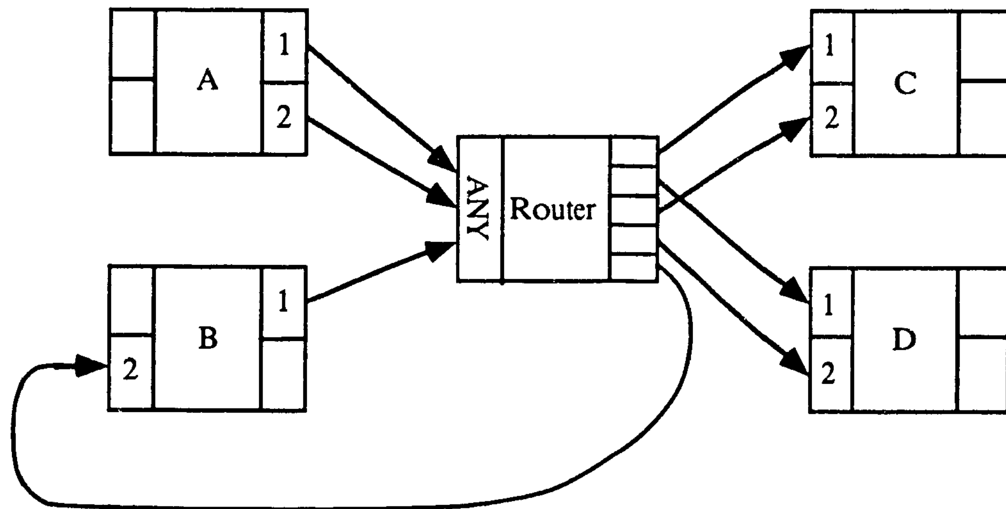

Figure 7.4: Possible Use of Router Module

7.6.3. Generation of the Expanded Simulator

After having created the enhanced simulation modules described above, we follow the same steps as before to generate the simulator. We use **SimDesc** to update the module descriptions of the three previously generated simulation modules as needed to reflect their new functionality. Note that we do not need to create a module description for the router module as it is given in the **mss.mdf** file that describes the characteristics of those modules supplied with MSS.

We then update the module object library and generate the simulator shown in Figure 7.3 using **SimMake**. The output generated by this simulator, using default attribute values, is shown in Listing 7.11 below. We can again experiment with varying some of the user-definable module attributes and with different simulator configurations. The summary of the results obtained for some of these simulators are given in Appendix D.

```
% Example2 1000
Running simulation for 1000.000000 microseconds
CPU 0 - Instructions executed: 382
CPU 0 - Performance: 0.382 MIPS
CPU 1 - Instructions executed: 372
CPU 1 - Performance: 0.372 MIPS

                         clock time = 1000000.000

                                   BINS
                                   ****
title        / reset at/ users/init/ max/ now/ av. free/ av. wait/qmax
CPU 0            0.000     852     0    1    0     0.000   548.239    1
CPU 1            0.000     847     0    1    0     0.000   552.255    1
BUS 0            0.000    5296     0    2    0 1.098e-02   120.687    1
MEM 0            0.000    3597     0    1    0     0.000   193.617    1
ROUTER2 0        0.000    1699     0    1    0     0.000   587.911    1

                               DISTRIBUTIONS
                               *************
title        /   obs/type        /parameters
I_time          383 randint     lower bound = 0, upper bound = 5
I_size          383 randint     lower bound = 0, upper bound = 5
O_size          383 randint     lower bound = 0, upper bound = 5
I_time          373 randint     lower bound = 0, upper bound = 5
I_size          373 randint     lower bound = 0, upper bound = 5
O_size          373 randint     lower bound = 0, upper bound = 5
W_prob          382 draw        probability = 0.250
W_prob          382 draw        probability = 0.250

                                ACCUMULATES
                                ***********
title        /   obs/  average/  est.st.dv/ minimum/ maximum
Bus_Usage       3398    0.663        0.473    0.000    1.000

                                  COUNTS
                                  ******
                           title        /  obs
                           Instructions   382
                           Instructions   372
```

(Listing 7.11 continued)

```
                              HISTOGRAMS
                              **********
                            S U M M A R Y
title         / obs/       sum/    average/  est.st.dv/ minimum/ maximum
REQ Length    1699   3597.000       2.117        1.234    1.000    5.000

 cell/        lower lim/   n/     freq/  cum
                                               I---------------------------
    0         -infinity    0      0.00   0.00  I
    1             1.000    653    0.38  38.43  I****************************
    2             2.000    607    0.36  74.16  I*************************
    3             3.000    166    0.10  83.93  I*******
    4             4.000    133    0.08  91.76  I******
    5             5.000    140    0.08 100.00  I******
                                          *** rest of table empty ***
                                               I-----------------------
```

Listing 7.11: Output of Expanded Simulator

The reports produced by the expanded simulator are more extensive than the ones from the first simple simulator. The reports for the ***bin*** objects can be interpreted as before; we notice that the average wait times have increased as both CPU modules now contend for the bus and each fetches longer instructions. The CPU also produces a two-line report each showing its "MIPS" rating; the count objects reported on were used to keep track of the number of instructions. The report for the ***accum*** object **Bus Usage** shows the average bus utilization in the *average* column; the remaining fields bear no useful information. The histogram **REQ Length** shows the distribution of the length of the transfer requests received by the bus module.

Again in the report for this simulator we observe how, even though we collected more specific information about the operation of our modules, the mass of data produced by SAMOC's default report tends to obscure the really useful data. It would help, therefore, if we could disable some of the less useful items in the report, such as those that duplicate information reported elsewhere. It should be possible to accomplish this task using a module's control inputs (to disable certain reports), and measurement outputs (to filter out irrelevant or duplicated data).

7.7. Future Refinement Cycles

Once we have generated a basic set of modules, we can continue by generating additional modules for other typical components of a computer system, such as disk drives or network interfaces. Of course, adding these comparatively slow peripheral devices into

the system will also introduce additional message types, such as I/O requests, as their operations cannot be adequately modeled with the existing message types.

To model a specific system, we will have to modify the existing components to more accurately reflect their real-life equivalents. In the bus module, for example, we will need to support bus access and arbitration protocols between different bus masters and additional characteristics such as burst transfers and interrupt signals. We can also incorporate additional routing capability based on the destination field of a message; this would make it possible, for example, to access two or more memory modules.

The CPU module would also need to be similarly enhanced, especially with regard to instruction execution. To model a particular software environment more precisely, we can then no longer assume a random distribution of instructions. Instead, we would have to create a simulation module which models the software environment and which provides the CPU module with an instruction stream to execute.

The possibilities for further refinement of the existing modules and for the addition of new modules are almost endless. But this is precisely the beauty of an open system such as MSS, that the user is not confined to a certain limited set of operations (or modules) deemed useful by the system's creator.

8. Conclusion

As noted in chapter three, a hierarchical, modular discrete-event modelling environment based on the DEVS formalism has previously been proposed in [Zeigler 1987]. Zeigler and others [Ören 1984, Aytaç 1986] introduced the view of a discrete-event simulation program as a composition of self-contained entities wherein the entities communicate using event messages. The entities in their simulation were selected from a model database and they shared a common port structure which allowed their coupling to form a hierarchy of models.

Using these ideas as a starting point, we have further refined the concept of a simulation model to include control ports, which allow the configuration of a model's operation; and measurement ports, which describe the statistical output to be generated as a result of the model's operation. We have shown how the coupling of models creates a model hierarchy and have addressed the related issue of how the varying levels of details can be incorporated into different models. We have also briefly discussed some of the advantages and disadvantages of a modular simulation system.

We then used this abstract view of a simulation model to guide our design of a simulation module that incorporates the model's characteristics. Also, such modules form the basic building blocks of our Modular Simulation System (MSS). By implementing this design in an object-oriented language, we are able to guarantee compatibility between all modules in the system, while still remaining open to user modification and extension of existing modules.

The concept of a model base, used by Zeigler et al., is realized in our design in the form of a module library. A module library incorporates the source and object code of a number of simulation modules, as well as a description of each module's characteristics. By providing a module description which can be interpreted by suitably equipped programs, a module library can be used as more than just a repository for simulation modules.

To allow the greatest possible number of users access to the simulation system's capabilities, we have to accommodate different types of users with widely varying backgrounds and degrees of knowledge. Some users, such as the developers of simulation modules or those modeling a real-world system, possess detailed knowledge of programming and modeling techniques and they are experts in their fields. Other users may bring with them only some high-level operational or managerial experience, but none

of the specialist knowledge of the two previous types of users. For these users, we have to encapsulate as much as possible of the system's complexity and provide an intuitive, concrete [Christie 1985], and [illegible] friendly interface.

To accomplish this task, we have created the **SimBuild** program, which acts as a graphical user interface for the simulation system. **SimBuild** represents simulation modules as icons and allows users to interactively generate a simulation program by choosing icons and connecting them on the screen. It guides the user in the selection of appropriate simulation modules using the module descriptions stored in each module library. **SimBuild** also verifies the values entered for module attributes and the module connections selected by the user to ensure that only a syntactically correct simulator can be constructed. By providing an easy-to-use interface, **SimBuild** encourages experimentation with different simulator configurations by all types of users and can be used as a rapid prototyping tool [Taylor 1982] aiding simulator design. We use the information supplied by **SimBuild** to generate an executable simulator from simulation modules stored in the module libraries.

The Modular Simulation System we have constructed forms a user-friendly environment for the generation of simulation programs. Its graphical user interface allows the user to rapidly construct and configure a simulator, while hiding unnecessary details. The reuse of existing simulation modules in the construction of simulators is encouraged by archiving them in module libraries for public use. The flexibility of each module is increased by the provision of user-selectable module configuration parameters. The Modular Simulation System illustrates the advantages that can be gained from object-oriented, modular software construction.

The system as presented in this thesis is meant to demonstrate the feasibility of the basic concept of using simulation modules and module libraries to generate a simulation program. To turn the system into a product useful to the general public, additional effort would be required. Certain features that are present in the module design have currently not been implemented or would need further work. For example, control and measurement ports have not yet been incorporated into the framework supported by the system. Other issues, such as a mechanism for dealing with module time bases or a way to specify simulator runtime parameters, remain unresolved and would need to be addressed. Furthermore, we would need to greatly expand the existing module libraries as users would demand many more simulation building blocks for realistic applications.

Other, more challenging, extensions might include adding an artificial intelligence component to the system or distributing the system over several machines. The first kind of extension has been proposed in [Radhakrishnan 1989] and would consist of a simulation *design assistant* [Rich 1988] incorporated into **SimBuild**. As the simulator generated by MSS already consists of self-contained entities which communicate by messages, it should not be too difficult to create a distributed version of the simulator.

References

Aytaç, Z. K. and Ören, T. I. **1986**. "Magest: A Model-Based Advisor and Certifier for GEST Programs", *Modelling and Simulation Methodology in the Artifical Intelligence Era* (M. S. Elzas, T. I. Ören and B. P. Zeigler, eds.), Elsevier Science Publishers B. V. (North-Holland), pp. 299-307.

Barth, P. S. **1986**. "An Object-Oriented Approach to Graphical Interfaces", *ACM Transactions on Graphics*, vol. 5, no. 2, pp. 142-172.

Bezivin, J. **1987**. "Some Experiments in Object-Oriented Simulation", *Proceedings of OOPSLA 87 Conference*, pp. 394-405.

Bharath-Kumar, K. and Kermani, P. **1984**. "Performance Evaluation Tool (PET): An Analysis Tool for Computer Communication Networks", *IEEE Journal on Selected Areas in Communications*, vol. 2, no. 1, pp. 220-225.

Birtwistle, G. M. , Dahl, O.-J., Myrhaug, B. and Nygaard, K. **1973**. *Simula Begin*, Auerbach Publications, New York.

Birtwistle, G. M. **1979**. *DEMOS - A System for Discrete Event Modelling on Simula*, MacMillan.

Booch, G. **1986**. "Object-Oriented Development", *IEEE Transactions on Software Engineering*, vol. 12, no. 2, pp. 211-221.

Christie, B. (ed.) **1985**. *Human Factors of the User-System Interface - A Report on an ESPRIT Preparatory Study*, North-Holland, New York City.

Eldredge, D. L., McGregor, J. D. and Summers, M. K. **1990**. "Applying the Object-Oriented Paradigm to Discrete-Event Simulations Using the C++ Language", *Simulation*, vol. 54, no. 2, pp. 83-91.

Fairly, R. **1985**. *Software Engineering Concepts*, McGraw-Hill Book Company.

Fishman, G. S. **1973**. *Concepts and Methods in Discrete Event Digital Simulation*, J. Wiley & Sons.

Guariso, G. Hitz, M. and Warthner, H. **1989**. "An Intelligent Simulation Model Generator", *Simulation*, vol.53, no. 2, pp. 57-66.

Hekmatpour, S. and Ince, D. **1988**. *Software Prototyping, Formal Methods and VDM*, Addison-Wesley Publishing Company, Inc.

Kernighan, B. W. and Ritchie, D. M. **1988**. *The C Programming Language, Second Edition*, Prentice-Hall, Englewood Cliffs, NJ.

Kleijnen, J. P. C. **1986**. "Selecting Random Number Seeds in Practice", *Simulation*, vol. 47, no. 1, pp. 15-17.

Kramer, J., Magee, J. and Ng, K. **1989**. "Graphical Configuration Programming", *IEEE ' ɔmputer*, pp. 53-65.

Larkin, T. S., Carruthers, R. I. and Soper, R. S. **1988**. "Simulation and Object-Oriented Programming: The Develop.nent of SERB", *Simulation*, vol. 52, no. 1, pp. 93-100.

Law, A. M. **1983**. "Statistical Analysis of Simulation Output Data", *Operations Research*, vol. 31, no. 3, pp. 983-1029.

Luqi 1989. "Software Evolution Through Rapid Prototyping", *IEEE Computer*, pp. 13-25.

Machiavelli, N. **1950**. *The Prince and The Discourses*, Modern College Editions, Random House, Inc.

Mathewson, S. C. **1975**. "Simulation Program Generators", *Simulation*, vol. 23, no. 6, pp. 181-189.

MC68020 1985. *32-Bit Microprocessor User' Manual, Second Edition*, Prentice-Hall, Englewood Cliffs, NJ.

McLeod, J. **1986**. "Computer Modeling and Simulation: The Changing Challenge", *Simulation*, vol. 46, no. 3, pp. 114-118.

McLeod, J. **1988**. "Simulation: The Formative Years", *Announcement of the Eastern Simulation Multiconference*, Orlando, FL.

Messerschmitt, D. G. **1984**. "A Tool for Structured Functional Simulation", *IEEE Journal on Selected Areas in Communications*, vol. 2, no. 1, pp. 137-147.

Meyer, B. **1988**. *Object-Oriented Software Construction*, Prentice-Hall, Englewood Cliffs, NJ.

Nygaard, K. **1986**. "Basic Concepts in Object-Oriented Programming", *ACM SIGPLAN Notices*, vol. 10, no. 10, pp. 128-132.

Nygaard, K. and Dahl, O.-J. **1981**. "Simula 67", *History of Programming Languages* (ed. R. L. Wexelblat), Academic Press, New York.

Ören, T. I. **1984**. "GEST - A Modelling and Simulation Language Based On System Theoretic Concepts", *Simulation and Model-Based Methodologies: An Integrative View* (T. I. Ören, B. P. Zeigler and M. S. Elzas, Eds.), Springer Verlag, Berlin, pp. 281-336.

Park, S. K. and Miller, K. W. **1988**. "Random Number Generators: Good Ones Are Hard to Find", *Communications of the ACM*, vol. 3, no. 10, pp. 1192-1201.

Parnas, D. L. **1972**. "On the Criteria to be Used in Decomposing Systems Into Modules", *Communications of the ACM*, vol. 15, no. 12, pp.1053-1058.

Parnas, D. L. and Clements, P. C. **1986**: "A Rational Design Process: How and Why to Fake It", *IEEE Transactions on Software Engineering*, vol. 12, no. 2, pp. 251-257.

Parnas, D. L., Clements, P. C. and Weiss, D. M. **1985**. "The Modular Structure of Complex Systems", *IEEE Transactions on Software Engineering*, vol. 11, no. 3, pp. 259-272.

Radhakrishnan, T., Grossner, C. and Wieland, T. **1989**. "A Design Apprentice Approach for the Simulation of Value Added Services", *Proceedings of the 1989 Singapore International Conference On Networks*, pp. 271-277.

Rentsch, T. **1982**. "Object-Oriented Programming", *SIGPLAN Notices*, vol. 17, no. 9, p. 51.

Rich, C. and Waters, R. C. **1988**. "The Programmer's Apprentice: A Research Overview", *IEEE Computer*, pp. 10-25.

Ruiz-Mier, S. and Talavage, J. **1987**. "A Hybrid Paradigm for Modeling of Complex Systems", *Simulation*, vol. 48, no. 4, pp. 135-141.

SAMOC 1988. *Reference Manual*, volumes. 1 and 2, Jade Simulations International, Calgary, Alberta.

Sauer, C. H., MacNair, E. A. and Kurose, J. F. **1984**. "Queueing Network Simulations of Computer Communication", *IEEE Journal on Selected Areas in Communications*, vol. 2, no. 1, pp. 203-219.

Schneider, G. M. **1978**. "A Modeling Package for Simulation of Computer Networks", *Simulation*, vol. 31, no. 6, pp. 181-192.

Sinclair, J. B., Dos. ! K. A. and Madala, S **1985**. "Computer Performance Evaluation With GIST: A Tool for Specifying Extended Queueing Network Models", *Proceedings of the 1985 Winter Simulation Conference*, pp. 290-299

Sommerville, I. **1982**. *Software Engineering*, Addison-Wesley Publishing Company, Inc.

Stroustrup, B. **1986**. *The C++ Programming Language*, Addison-Wesley Publishing Company, Inc. (reprinted with corrections 1987).

Taylor, T. and Standish, T. A. **1982**. "Initial Thoughts on Rapid Prototyping Techniques", *ACM SIGSOFT Software Engineering Notes*, vol. 7, no. 5, pp. 160-166 (reprinted in: Agresti, W. W. 1986: *New Paradigms For Software Development, Computer Society Press*, Washington, DC).

Treu, S. **1988**. "Designing a 'Cognizant Interface' Between the User and the Simulation Software", *Simulation*, vol. 51, no. 6, pp. 227-234.

Wade, W. D., Mortara, M. E., Leong, P. K. and Frost, V. S. **1984**. "Interactive Communication Systems Simulation Model - ICSSM", *IEEE Journal on Selected Areas in Communications*, vol. 2, no. 1, pp. 102-128.

Wilson, J. R. and Pritsker, A. B. **1978**. "A Survey of Research on the Simulation Startup Problem", *Simulation*, vol. 31, no. 4, pp. 55-58.

Yamamoto, Y 1986. "Graphic Interfaces for Modelling Systems", *Modelling and Simulation Methodology in the Artifical Intelligence Era* (M. S. Elzas, T. I. Ören and B. P. Zeigler, eds.), Elsevier Science Publishers B. V. (North-Holland), pp. 399-403.

Yeh, J. W. **1979**. "Simulation of Local Computer Networks - A Case Study", *Computer Networks*, vol. 3, pp. 401-417.

Zeigler, B. P. **1984a**. *Multifacetted Modelling and Discrete Event Simulation*, Academic Press, London and Orlando.

Zeigler, B. P. **1984b**. "System Theoretic Foundations of Modelling and Simulation", *Simulation and Model-Based Methodologies: An Integrative View* (T. I. Ören, B. P. Zeigler and M. S. Elzas, Eds.), Springer Verlag, Berlin, pp. 91-118.

Zeigler, B. P. **1987**. "Hierarchical, Modular Discrete-Event Modelling in an Object-Oriented Environment", *Simulation*, vol. 49, no. 5, pp. 219-230.

Zelkowitz, M. V. **1984**. "A Taxonomy of Prototype Designs", *ACM SIGSOFT Software Engineering Notes*, vol. 9, no. 5, pp. 11-12.

Appendix A. Source Code for MSS Base Classes

This appendix lists the class declaration and implementation files for classes ***msg***, ***msg_queue***, and ***simmod*** (see chapter six), as well as for the ***router5*** class discussed in chapter seven. Additionally, it also lists some of the other files provided by MSS, such as the makefiles supplied with the system and the include files which define the MSS directory structure.

A.1. MSS Base Classes

A.1.1. Inter-Module Messaging: Classes *msg* and *msg_queue*

```
// msg_queue.hxx - declarations for inter-module messages

#ifndef     _MSG_QUEUE_HXX_
#define     _MSG_QUEUE_HXX_

#ifndef     LIST_DECLS
#include    "list.hxx"
#endif

//
//  Message types
//

typedef int      msg_type;

const msg_type  ANY_MSG   = -1; // any/all messages, regardless of type
const msg_type  UNDEFINED = 0;  // catch undefined fields


//
//  Module Identification Number
//   (Note: declared here, otherwise circular reference)
//

typedef int      mod_ID;
const mod_ID     UNKNOWN   = -1; // module's ID is unknown

//
//  Instances of class msg() are used for representing messages
//  between simulation modules; individual messages are held on
//  priority FIFO-ordered msg_queue()s by each simulation module.
//  Class msg() supports a simple request/acknowledge protocol,
//  where a message of the same type can be exchanged between two
//  modules.  Sending in one direction is defined as a request,
//  sending the other way as an acknowledgement of the previous
//  request message; only one message instance is needed.
//
```

```
class msg_queue;

class msg : public pr_list_el {
  int        Priority;   // message priority
protected:
  // all messages provide the following fields
  msg_type   Mtype;      // type of this message
  mod_ID     SendID;     // ID of sending module
  mod_ID     DestID;     // ID of destination module
  boolean    ReqAck;     // T if request, F if acknowledge
public:
  //
  // adapt pr_list_el{} member functions for msg{}
  //
  msg        *prev()     { return (msg*) pr_list_el::prev(); }
  msg        *next()     { return (msg*) pr_list_el::next(); }
  msg_queue *current_list() { return (msg_queue*)
pr_list_el::current_list(); }
  // insert this message into the msg queue based on priority
  void       insert(msg_queue* mq, boolean before = FALSE)
                 { pr_list_el::insert((pr_list*)mq, before); }
  // remove this message from any message queue
  void       remove()    { pr_list_el::remove(); }

  // rank() is used by pr_list{} for ordering pr_list_el's;
  // for msg_queue{} we want higher priority objects to go to the front,
  // but for pr_list{} (which is used to implement msg_queue{}) lower
  // ranked items are placed closer to the head of the list. Therefore,
  // the rank() of a msg_queue_el{} its Priority subtracted from 0.
  double     rank()      { return (double) -Priority; }

  //
  // member functions specific to messages
  //
  int        prio()              { return Priority; }
  msg_type   type()              { return Mtype; }
  int        get_send()          { return SendID; }
  void       set_send(mod_ID id) { SendID = id; }
  int        get_dest()          { return DestID; }
  void       set_dest(mod_ID id) { DestID = id; }
  void       set_ID2(mod_ID sid, mod_ID did)  { SendID=sid; DestID=did; }
  void       swap_ID()     { mod_ID id=SendID; SendID=DestID; DestID=id; }
  // support for request/acknowledge message protocol
  void       set_req()           { ReqAck = TRUE; }
  void       set_ack()           { ReqAck = FALSE; }
  boolean    is_req()            { return ReqAck; }
  boolean    is_ack()            { return not ReqAck; }
  void       print();            // print type & priority

  // we must tell a msg{} object what type of message it represents,
  // but priority is optional and defaults to 0 for all messages
  msg(msg_type mt, int pr = 0)  { Mtype = mt; Priority = pr; }
};
```

```
//
// Instances of msg_queue{} hold messages derived from msg{};
// most functions are already provided by class pr_list{}
//
class msg_queue : public pr_list {
public:
  // available from pr_list{}:
  // void       print();     // print elements on list
  // boolean    empty();     // return TRUE if queue empty

  // adapt member functions of pr_list{}
  int    length()      { return pr_list::cardinal(); }
  msg*   first()       { return (msg*) pr_list::first(); }
  msg*   last()        { return (msg*) pr_list::last(); }

  // get the first message from the queue (or NULL)
  msg*   getfirst();
};

#endif  _MSG_QUEUE_HXX_
```

A.1.2. Simulation Modules: Class *simmod*

```
// simmod.hxx - declarations for baseclass of simulation mods

#ifndef     _SIMMOD_HXX_
#define     _SIMMOD_HXX_

#ifndef     SYNCH_DECLS
#include    "synch.hxx"      // includes samoc.hxx
#include    "random.hxx"
#include    "collect.hxx"
#endif

#ifndef     _MSG_QUEUE_HXX_
#include    "msg_queue.hxx"
#endif

//
//  Instances of class simmod{} supply the
//  basic functionality for simulation modules
//
class simmod : public entity {
  bin       *inqueue;   // count incoming messages & keep statistics
  msg_queue *mqueue;    // holds incoming messages
protected:
  mod_ID    ID;         // module ID
public:
  // member functions for manipulating messages
  int       msg_length()        { return mqueue->length(); }
  void      send_msg(simmod*, msg*);            // send msg to a module
```

```
  msg*      wait_msg(msg_type mt = ANY_MSG);    // await next message
  msg*      get_msg(msg_type mt = ANY_MSG);     // get next message
  msg*      only_msg(msg_type);            // wait for msg of this type ONLY
  boolean   msg_avail(msg_type mt = ANY_MSG);   // TRUE if msg available
  // support for request/acknowledge protocol
  msg*      request(simmod*, msg*);         // send request, get ack
  msg*      reqack(simmod*, msg*);          // send request & wait ack
  void      send_ack(simmod*, msg*);        // return acknowledgement
  // Fatal error handler: prints routine, error message, message
  void      mod_error(char* rt, char* em, msg* mess=NULL); // ABORTS

  simmod();
  ~simmod();
  void init_port(mod_ID id) { ID = id; } // output port initialization
  void init_attr()          {}          // module attribute init'ion
  void body()               {}          // simulation time behavior
  void report()             {}          // custom final reports
};
NEW_(simmod, 1024);

#endif  _SIMMOD_HXX_
```

A.1.3. Implementation of Base Classes

```
// mss.cxx - member functions for classes msg{}, msg_queue{}, simmod{}


#ifndef     simmod
#include    "simmod.hxx"    // includes msg_queue.hxx
#endif


//
// Class msg{} - inter-module messages
//

// Print message attributes
void msg::print()
{
    printf("Message: Type=%3d, Prio=%3d, Send=%3d, Dest=%3d -- ",
            Mtype, Priority, SendID, DestID);
    if ( ReqAck )
        puts("Request");
    else
        puts("Acknowledge");
}



//
// Class msg_queue{} - queues for inter-module messages
//
```

```
// get the first message out of the queue (NULL if none exists)
msg* msg_queue::getfirst()
{
    msg*    mess;

    if ( mess = first() ) {
        mess->remove();
        return(mess);
    }
    else
        return(NULL);
}
```

```
//
// Class simmod{} - simulation modules
//

simmod::simmod()
{
    inqueue = new bin(title(), 0);
    mqueue  = new msg_queue();
}


simmod::~simmod()
{
    delete inqueue;
    delete mqueue;
}


// Send a message to a module
void simmod::send_msg(simmod* mod, msg* mess)
{
    mess->insert(mod->mqueue);
    mod->inqueue->give(1);
}


// Wait for (if necessary) and get the first available message;
// if a type is specificed, make sure the message is of that type.
// Leaves unexpected message in the input queue and returns NULL.
msg* simmod::wait_msg(msg_type mt = ANY_MSG)
{
    msg *mess;

    inqueue->take(1);        // no message in the queue, wait
    mess = mqueue->first(); // get first message from queue

    // make sure it is the message type we want
    if ( mt == ANY_MSG || mess->type() == mt ) {
        mess->remove();      // take message out of the queue
        return(mess);
    }
    else {
        ::warning("simmod::wait_msg", "Unexpected message of type %d
                  in %s\n", mess->type(), title());
        inqueue->give(1);
        return(NULL);
    }
}
```

```
// Wait for (if necessary) and get the first available message;
// make sure the message is of that type, otherwise abort simulation.
// Use ONLY if NO OTHER message type can be received.
msg* simmod::only_msg(msg_type mt)
{
    msg *mess;

    inqueue->take(1);     // if no message is in the queue, wait for one
    mess = mqueue->getfirst();     // get the first msg out of the queue

    // make sure it is the message type we want
    if ( mt != ANY_MSG && mess->type() != mt ) {
        mod_error("simmod::only_msg", "Unexpected message type", mess);
    }
    return(mess);     // return message of type mt
}


// Get a message of type mt from the input queue.
// ASSUMES there is a MESSAGE AVAILABLE (ie. no waiting)
msg* simmod::get_msg(msg_type mt = ANY_MSG)
{
    msg *mess;

    if ( not mqueue->length() ) {        // no message waiting
    ::warning("simmod::get_msg","No message available (NULL returned)");
    return(NULL);
    }

    mess = mqueue->first(); // get first message from queue

    if ( mt == ANY_MSG ) {
        inqueue->take(1);     // take one message
        mess->remove();       // remove message from queue
    }
    else {
        // scan for message of type 'mt'
        while ( mess /* != NULL */  && (mess->type() != mt) )
            mess = mess->next();

        if ( mess ) {    // found a message
            inqueue->take(1);
            mess->remove();
        }
        else
    ::warning("simmod::get_msg","No message available (NULL returned)");
    }
    return(mess);    // NULL or message of type mt
}
```

```
// Search input queue for a message of type mt;
// return TRUE if a message is available, FALSE otherwise
boolean simmod::msg_avail(msg_type mt = ANY_MSG)
{
    if ( mt == ANY_MSG )
        return(not mqueue->empty());

    msg *mess = mqueue->first();     // get first message from queue

    // scan for a message of type 'mt'
    while ( mess /* != NULL */  && (mess->type() != mt) )
        mess = mess->next();

    if ( mess )
        return(TRUE);   // found a message
    else
        return(FALSE);  // end of queue
}


// Send a request message to a module and wait for the corresponding
// acknowledgement message; if the next message received is not the
// expected acknowledgement, return NULL as error indication.
// Leaves unexpected message in the input queue and returns NULL.
msg* simmod::request(simmod* mod, msg* mess)
{
    msg_type    mt = mess->type();  // remember type of message

    // send the message
    mess->set_req();
    mess->insert(mod->mqueue);
    mod->inqueue->give(1);

    // wait for acknowledgement
    inqueue->take(1);         // if no msg is in the queue, wait for one
    mess = mqueue->first(); // get first message from queue

    // make sure it is an acknowledgement of the correct message
    if ( (mess->type() == mt) && (mess->is_ack()) ) {
        mess->remove();       // take acknowledgement out of the queue
        return(mess);         // and return it
    }
    else {
        ::warning("simmod::request", "Unexpected message of type %d
                in %s\n", mess->type(), title());
        inqueue->give(1);    // message remains in queue
        return(NULL);
    }
}
```

```
// Send a request message to a module and wait for the corresponding
// acknowledgement message; if the next message received is not the
// expected acknowledgement, abort the simulation.
// Use ONLY if NOTHING BUT the ACKNOWLEDGEMENT message can be received.
msg* simmod::reqack(simmod* mod, msg* mess)
{
    msg_type    mt = mess->type();  // remember type of message

    // send the message
    mess->set_req();
    mess->insert(mod->mqueue);
    mod->inqueue->give(1);

    // wait for acknowledgement
    inqueue->take(1);          // if no msg is in the queue, wait for one
    mess = mqueue->getfirst();  // get the first msg out of the queue

    // make sure it is an acknowledgement of the correct message
    if ( mess->type() != mt )
        mod_error("simmod::reqack","Unexpected message received", mess);
    if ( mess->is_req() )
        mod_error("simmod::reqack", "Acknowledgement expected", mess);

    return(mess);    // return acknowledgement
}


// Send the acknowledgement message for a received request message
void simmod::send_ack(simmod* mod, msg* mess)
{
    mess->set_ack();
    mess->insert(mod->mqueue);
    mod->inqueue->give(1);
}


// Fatal error handler: if error was caused by message traffic (ie.
message
// pointer non-NULL), print contents of message; in all cases, print
// routine name and error message; abort simulation run
void simmod::mod_error(char* rt, char* em, msg* mess=NULL)
{
    if ( mess ) {
        puts("\nThe following message caused a fatal error:");
        mess->print();
    }
    ::error(rt, "%s in %s\n", em, title());
}
```

A.2. Include and Make Files

A *C* include file in the MSS root directory describes the MSS directory hierarchy in terms of *C* preprocessor macros. This file is used by the MSS utility programs to supply default pathnames and to select the directory where generated files are stored. Several makefiles supplied with MSS allow automatic compilation of simulation modules, utilities and simulators. All makefiles include two other makefile fragments (kept in the MSS root directory), which supply macros defining the MSS directory hierarchy and compilation rules.

A.2.1. C Include File

```
// MSS_Path.h - Definition of MSS directory structure

#ifndef _MSS_PATH_H__
#define _MSS_PATH_H__

/* MSS root directory */
#define MSSDIR  "/home/grad/thomasw/MSS"

/* MSS backup directory */
#define MBAK    "/home/grad/thomasw/bak"

/* MSS binaries directory */
#define MBIN    "/home/grad/thomasw/bin"

/* MSS include file directory */
#define MINC    "/home/grad/thomasw/inc"

/* MSS libraries directory */
#define MLIB    "/home/grad/thomasw/lib"

/* MSS module libraries directory */
#define MMDF    "/home/grad/thomasw/mdf"

/* MSS simulation program source directory */
#define MSIM    "/home/grad/thomasw/sim"

/* SIMBUILD directory */
#define SBDIR   "/home/grad/thomasw/simbld"

/* SIMBUILD icons file directory */
#define SBICS   "/home/grad/thomasw/simbld/icons"

/* SIMBUILD include file directory */
#define SBINC   "/home/grad/thomasw/simbld/inc"
```

```
/* SIMBUILD source file directory */
#define SBSRC    "/home/grad/thomasw/simbld/src"

/* MSS simulation module source directory */
#define MSRC     "/home/grad/thomasw/src"

/* MSS utilities include file directory */
#define UINC     "/home/grad/thomasw/util/inc"

/* MSS utilities source file directory */
#define USRC     "/home/grad/thomasw/util/src"


#endif  _MSS_PATH_H__
```

A.2.2. Makefiles

The following two makefiles are included by all MSS makefiles; they define the standard directory paths and the default compiler options to use when creating MSS programs.

A.2.2.1. Directory Hierarchy

```
# Make.Path - description of MSS directory tree for Makefiles


# MSS root directory
MSSDIR = $(HOME)/MSS

# MSS backup directory
MBAK  = $(MSSDIR)/bak

# MSS binaries directory
MBIN  = $(MSSDIR)/bin

# MSS include file directory
MINC  = $(MSSDIR)/inc

# MSS libraries directory
MLIB  = $(MSSDIR)/lib

# MSS module libraries directory
MMDF  = $(MSSDIR)/mdf

# MSS simulation program source directory
MSIM  = $(MSSDIR)/sim

# SIMBUILD directory
SBDIR = $(MSSDIR)/simbld

# SIMBUILD icons file directory
SBICS = $(SBDIR)/icons
```

```
# SIMBUILD include file directory
SBINC = $(SBDIR)/inc

# SIMBUILD source file directory
SBSRC = $(SBDIR)/src

# MSS simulation module source directory
MSRC  = $(MSSDIR)/src

# MSS utilities include file directory
UINC  = $(MSSDIR)/util/inc

# MSS utilities source file directory
USRC  = $(MSSDIR)/util/src


#
# SAMOC directories
#

# SAMOC root directory
SAMOCDIR = /jupiter3/site/source/samoc

# SAMOC include file directory
SINC = $(SAMOCDIR)/include

# SAMOC libraries directory
SLIB = $(SAMOCDIR)/lib

# SAMOC source code directory
SSRC = $(SAMOCDIR)/Src
```

A.2.2.2. Compilation Rules

```
# Make.Rules - Compiler options & rules for Makefiles


# C++ compiler name (Designer C++ for Sun) & quiet compile option
CPP = ccxx @Q

# compiler flags: debug all (including SAMOC!)
CFLAGS0 = -g -DDEBUG
CPPFLAGS0 = -g -DDEBUG

# compiler flags: include debugger info
CFLAGS1 = -g
CPPFLAGS1 = -g

# compiler flags: optimize
CFLAGS2 = -O
CPPFLAGS2 = -O
```

```
# set compiler flags to use (select debug (*0/*1) or optimize (*2))
CFLAGS = $(CFLAGS1) $(INCLUDES)
CPPFLAGS = $(CPPFLAGS1) $(INCLUDES)

# command used to compile source to an object file (only)
CCMD = $(CC) -c $(CFLAGS)
CPPCMD = $(CPP) -c $(CPPFLAGS)

# transformation rules for C (supersedes built-in rule) & C++ source
.SUFFIXES: .c .cxx
.c.o:
	$(CCMD) $<
.cxx.o:
	$(CPPCMD) $<
```

A.2.3. Module Generation

This is a makefile which is provided in the module source directory. It can be used to compile simulation module source files and to generate the object libraries for a module library.

```
#
#   Generate MSS simulation modules.  Call with: make MOD=<name>
#

# get MSS directory tree
include ../Make.Path

# get compiler options & rules
include ../Make.Rules

#
# Directories & paths for compiler
#

# directories to search for include files
INCLUDES = -I$(MINC) -I$(SINC)

# libraries to bind with final object
LIBES =

#
# Macros for related files
#

# dummy declaration, in case MOD is undefined
MOD = TYPE_make_help_FOR_HELP

# all files to include in backup
BAK_FILES= $(MSSDIR)/Make.Path $(MSSDIR)/Make.Rules $(MSSDIR)/MSS_Path.h
           $(MINC)/*.hxx $(MSRC)/Makefile $(MSRC)/*.cxx $(MMDF)/*.mdf
```

```
#
# Targets
#

# Make simulation module (object only)
$(MOD).o:    $(MOD).cxx $(MINC)/$(MOD).hxx

# Install module library
# To create module library: ar cvr <library name> <obj module names>..
library:     always
             /bin/rm -f $(LIB).a
             /bin/ar cvr $(LIB).a $(OBJS)
             /usr/bin/ranlib $(LIB).a
             /bin/mv $(LIB).a $(MLIB)

# Make & install library for MSS base class
mss.a:       mss.o
             /bin/rm -f mss.a
             /bin/ar cvr mss.a mss.o
             /usr/bin/ranlib mss.a
             /bin/mv mss.a $(MLIB)

mss.o:       mss.cxx $(MINC)/msg_queue.hxx $(MINC)/simmod.hxx

#
# Utility targets
#

# Usage
help:        always
     @echo "Usage: make MOD=<name>   - make simulation module <name>"
     @echo "       make library LIB=<libname> \"OBJS=<object names>..\""
     @echo "                       - make & install module library <libname>"
     @echo "       make mss.a        - make & install MSS library"
     @echo "       make clean        - clean out garbage files"
     @echo "       make backup       - backup simulation modules"

# clean out object files & assorted garbage
clean:       always
             /bin/rm -f a.out core *.o *%

# create backup archive of simulations & modules
backup:      always
             ar cvr MSS.Modules $(BAK_FILES)
             compress MSS.Modules
             chmod 400 MSS.Modules.Z
             mv MSS.Modules.Z $(MBAK)

# dummy target to force execution
always:
```

A.2.4. Simulator Generation

This is a sample makefile which provides all the include directories and library files needed to compile a simulator. It can be used when a simulator generated by **SimMake**.

```
#
#   Manually generate a MSS simulation program.
#   Call with: make SIM=<name> LIB=<mdflib>
#

# get MSS directory tree
include ../Make.Path

# get compiler options & rules
include ../Make.Rules

#
# Directories & paths for compiler
#

# directories to search for include files
INCLUDES = -I$(MINC)  -I$(SINC)

# libraries to bind with final object
LIBES = $(MLIB)/mss.a $(SLIB)/samoc.a -lm

#
# Macros for related files
#

# dummy declaration, in case SIM is undefined
SIM = TYPE_make_help_FOR_HELP

# all files to include in backup
BAK_FILES = $(MSIM)/Makefile $(MSIM)/*.sdf $(MSIM)/*.cxx $(MSIM)/*.make


#
# Targets
#

# Make simulation program
# This is for compilation by hand, without a SIMMAKE-generated makefile
# LIB is the name of the module object library (hope there is only one)
$(SIM):     $(SIM).o
            $(CPP) $(CPPFLAGS) $(SIM).o $(LIBES) $(MLIB)/$(LIB).a
            /bin/mv $(SIM) $(MBIN)

$(SIM).o:   $(SIM).cxx
```

```
#
# Utility targets
#

# Usage
help:       always
        @echo "Usage: make SIM=<name> LIB=<mdflib>"
        @echo "- make simulation <name> using MDF library <mdfname>"
        @echo "make clean       - clean out garbage files"
        @echo "make backup      - backup simulators (includes MDF, SDF)"


# clean out object files & assorted garbage
clean:      always
            /bin/rm -f a.out core *.o *%


# create backup archive of simulations & modules
backup:     always
            ar cvr MSS.Simulators $(BAK_FILES)
            compress MSS.Simulators
            chmod 400 MSS.Simulators.Z
            mv MSS.Simulators.Z $(MBAK)


# dummy target to force execution
always:
```

A.3. Router Modules

We list only the header and source files for the one-to-five router module class ***router5***. The other two classes, ***router2*** and ***router3***, are identical except for a reduced number of arguments to their *init_port* and *init_attr* functions and some minor optimizations in their implementation.

```
// router.hxx - message router classes router2, router3, router5

#include    "simmod.hxx"

// Note: class declarations for router2 and router3 omitted

//************************
//  1->5 router module  *
//************************
//
// Time Base: unaffected
//
// Model Attributes: none
//
```

```
// Model Characteristics:
//  - routes incoming messages (regardless of type)
//    to one of several (up to 5) connected modules
//  - no simulation time delay is encountered for
//    messages passing through this module
//
// Additional Statistics: none
//
const int   MAX_PORTS5 = 5;   // max. number of output ports in module

class router5 : public simmod {
// PORTS
  simmod    *Port[MAX_PORTS5];     // ptrs to I-port of connected modules
// INTERNALS
  int       Num_ports;             // number of connected modules
  mod_ID    Mod_ID[MAX_PORTS5];    // module IDs of connected modules
  friend int router_check_portmap(int maxp, int np, simmod* prt[],
                                  mod_ID mid[], char* t);
public:
  router5();
  ~router5();
  void      init_port(mod_ID id, simmod *p1=NULL, simmod *p2=NULL,
                  simmod *p3=NULL, simmod *p4=NULL, simmod *p5=NULL);
  void      init_attr(int np, mod_ID id1=-1, mod_ID id2=-1,
                  mod_ID id3=-1, mod_ID id4=-1, mod_ID id5=-1);
  void      body();
};
NEW_(router5, 1024);


// router.cxx - message router classes router2, router3, router5

#include    "router.hxx"

//*** Note: Implementation of classes router2 and router3 omitted

//*** Check port mapping for consistency for all routers (friend
function);
//    returns number of port entries actually used
int router_check_portmap(int maxp, int np, simmod* port[], mod_ID mid[],
char* t)
{
      int         slot, slot2;

      // Check for errors in port mapping (ie. port* <-> module ID)
      for (slot=0; slot<maxp; slot++) {
            if ( mid[slot] > 0 ) {  // assume slot in use
                  if ( port[slot] == NULL )
                        ::error("router_check_portmap",
    "Pointer to module #%d (ID #%d) is NULL in %s", slot, mid[slot], t);
            }
```

```
            else  // assume slot not in use
                  if ( port[slot] != NULL )
                        ::error("router_check_portmap",
             "Pointer to unused module #%d is non-NULL in %s", slot, t);
      }

  // We have consistent <port, mid> entries; there may still be holes in
  // the port and mid arrays. We now move all used entries to the start
      for (slot=0; slot<maxp && mid[slot]>0; slot++) ; // find free slot

      // If not all slots are used, move all used entries forward
      if ( slot < maxp )       {      // 'slot'  points to unused entry
            slot2 = slot+1;           // 'slot2' points to next used entry
            while ( slot2 < maxp ) {
                  if ( mid[slot2] > 0 ) { // copy used slot to free slot
                        mid[slot] = mid[slot2];
                        port[slot] = port[slot2];
                        slot++;             // next slot is free or stale
                  }
                  slot2++;
            }
      }
      // one final sanity check
      if ( slot != np )
            ::error("router_check_portmap",
       "Number of connections does not match number of ports in %s", t);
      return(slot);     // return number of used slots
}


//*** router5: 1->5 message router

router5::router5()
{ }

router5::~router5()
{ }

void router5::init_port(mod_ID id, simmod *p1=NULL, simmod *p2=NULL,
                  simmod *p3=NULL, simmod *p4=NULL, simmod *p5=NULL)
{
      ID = id;

      // just assign pointers blindly, leave error checking for later
      Port[0] = p1;  Port[1] = p2;  Port[2] = p3;
      Port[3] = p4;  Port[4] = p5;
}
```

```
void router5::init_attr(int np, mod_ID id1=-1, mod_ID id2=-1,
                   mod_ID id3=-1, mod_ID id4=-1, mod_ID id5=-1)
{
      if ( np <= 0 )
            ::error("router5::init_attr",
      "No ports specified for %s", title());
      else if ( np > MAX_PORTS5 )
            ::error("router5::init_attr",
      "Too many ports specified for %s (max. %d)", MAX_PORTS5, title());

      // assign module IDs to slots
      Mod_ID[0] = id1; Mod_ID[1] = id2; Mod_ID[2] = id3;
      Mod_ID[3] = id4; Mod_ID[4] = id5;

      // check for errors
      Num_ports=router_check_portmap(MAX_PORTS5,np,Port,Mod_ID,title());
}

void router5::body()
{
      mod_ID      dest_ID;
      int         slot;
      msg         *in_msg;    // pointer to received message

      while ( TRUE ) {
            in_msg = wait_msg();          // wait for next message
            dest_ID = in_msg->get_dest(); // get destination module ID

            // scan for destination module ID
            for (slot=0; (Mod_ID[slot] != dest_ID) && (slot<MAX_PORTS5);
                  slot++) ;

            if ( slot >= MAX_PORTS5 ) {
                  fprintf(stderr,
              "router5::body: Message: Type=%d, SendID=%d, DestID=%d\n",
               in_msg->type(), in_msg->get_send(), in_msg->get_dest());

                  ::error("router5::body",
                        "Destination module #%d not found", dest_ID);
            }

            send_msg(Port[slot], in_msg); // transfer message
      }
}
```

Appendix B. Format of MDF, SDF, and SGF Files

B.1. Module Description File Structure

This section gives the format of the module description file (MDF) produced by **SimDesc**. String literals, except for MDF keywords, which are given in their literal form (see chapter six, section 6.3.1), are quoted; all other text strings describe one field of the MDF. A double underscore ("__") indicates a required sequence of whitespace characters (tabs and/or spaces). A single hash ("#") character stands for an optional comment block (one or more lines) which may be inserted at that point in the description using SIMDESC; additional comments may be inserted later by manually editing the module description file. A single colon (":") in a line indicates that the line preceding it may be repeated, depending on the number of items described in that line.

```
"# Module library:" filename of this MDF
%AUTHOR __ name of author
%CDATE  __ creation date
%MDATE  __ modification date
%SYMTAB number of module descriptions in symbol table
module name __ line offset __ byte offset
:
%BOF number of module descriptions in file

# %MODBEG __ module class name
filename for module's header and object files
module icon type1 or filename

# %INPORT number of module's input ports
input port name __ input message type
:

# %OUTPORT number of module's output ports
output port name __ output message type
:

# %CONTROL (No format defined yet)

# %MEASURE (No format defined yet)

# %ATTRIB number of module's attributes
attribute name __ attr. type2 __ attr. default value
number of values in selection __ first ... last value
      or
-1 __ low limit __ high limit
:
```

```
# %INIT
number of arguments to module's init_port function
output port name
:

number of arguments to module's init_attr function
attribute name __ attribute value

%MODEND
:
: more module descriptions
:

%EOF
```

Notes:

(1) Icon type can be one of: B (bus), C (CPU), D (disk), G (gateway between networks), K (sink), M (memory), N (network interface), R (router), S (software)

(2) Attribute type can be one of: T (text string), I (signed long integer), or F (double floating point)

B.2. Simulator Description File Structure

This section describes the format of the simulator description file (SDF) produced by **SimBuild**. The description follows the same format as used for the module description file. For a list of the keywords used, see chapter six (section 6.3.2). Comment fields may be inserted for documentation manually after the SDF has been created.

```
"# Simulator description:" filename of this SDF
"#"
"# Module libraries used:"
"#" __ name of module library
:

%AUTHOR __ name of author
%CDATE  __ creation date

%BOF number of module instances in the simulator
%RUNTIME (No format defined yet)
%LIBRARY filename of MDF for following modules
%MODBEG __ module name __ module ID number
class name of this module instance

%OUTPORT output port name __ destination module ID
:

%ATTRIB attribute name __ attribute value
:

%MODEND
:
: more module instances from same library
:

:
: more module libraries
:

%EOF
```

B.3. Simulator Graphics File Structure

This section describes the format of the simulator graphics file (SGF) produced by **SimBuild** The description follows the same format as used for the module description file. This file is used internally by **SimBuild** to keep track of icon positions on the screen.

```
%ICON number of icons (= number of modules)
icon type (or filename) x coordinate __ y coordinate
:
: repeat for each icon
:

%CONNECT number of ports for module
number of connections of this port port type
src x coord __ src y coord __ dest x coord __ dest y coord
:
: repeat for each port
:

:
: repeat for each module
:
```

Appendix C. Listings For Example Simulator

This appendix shows the source code listings, MDF, and SDF for the first example simulator generated during the session described in chapter seven, sections 7.1 through 7.5.

C.1. Inter-Module Messages

```
// msgs.hxx - inter-module message types: ACCESS, TRANSFER

#ifndef     _MSGS_HXX_
#define     _MSGS_HXX_

#ifndef     _MSG_QUEUE_HXX_
#include    "msg_queue.hxx"
#endif

//****************************************
//  ACCESS message: Bus <-> Memory
//   - access request:      Bus -> Memory
//   - access acknowledge: Memory -> Bus
//
//  An ACCESS msg is used to transfer data to and from a memory module
//  or other passive module (bus slave) connected to a bus. The data
//  size is implied to be the width of the memory interface and is
//  determined by the (bus) module sending the ACCESS message.  An
//  access can be either a read or a write access, as determined by the
//  sending module using the Read variable.
//

const msg_type  ACCESS = 1;

class acc_msg : public msg {
public:
  boolean   Read;    // Read (TRUE) / Write (FALSE) access

  // initialize an access request
  void  init_acc(mod_ID send, mod_ID dest, boolean rd=TRUE)
    { ReqAck = TRUE; SendID = send; DestID = dest; Read = rd; }

  acc_msg(msg_type mt=ACCESS, int pr=0)  : (mt,pr) {}
};
```

```
//*************************************
// TRANSFER message: CPU <-> Bus
//  - transfer request:      CPU -> Bus
//  - transfer acknowledge: Bus -> CPU
//
// A TRANSFER message is used to transfer data to and from a CPU module
// or other active module (bus master) connected to a bus. The data
// size is variable and is specified by the module sending the TRANSFER
// message in the Length field; the receiving (bus) module must deter-
// mine the number of ACCESS requests(bus cycles) necessary to transfer
// the requested data. The start address of the data transfer may be
// specified by the sender in the Adr field; it may be used by the
// receiver of the message to select a particular module as the
// destination of the resulting ACCESS messages. The direction of the
// data transfer is determined using the Read field.
//

const msg_type  TRANSFER = 2;

class tx_msg : public acc_msg {
public:
  long      Adr;    // start address of memory access
  long      Length; // number of bits to transfer

  // initialize a transfer request
  void  init_trans(mod_ID send, mod_ID dest, long len,
                   boolean rd=TRUE, long adr=0)
    { ReqAck = TRUE; SendID = send; DestID = dest;
      Read = rd; Adr = adr; Length = len; }

  tx_msg(msg_type mt=TRANSFER, int pr=0) : (mt,pr) {}
};

#endif  _MSGS_HXX_
```

C.2. CPU Module

```
// cpu1.hxx - simple CPU module (non-random)

#ifndef     _SIMMOD_HXX_
#include    "simmod.hxx"
#endif

#ifndef     _MSGS_HXX_
#include    "msgs.hxx"
#endif

//**********************
//  Simple CPU module  *
//**********************
//
// Time Base: 1 time unit = 1 nanosecond
//
// Module Attributes:
//   - CPU clock cycle time (default: 100 ns, i.e. 10 MHz CPU clock)
//
// Module Characteristics:
//   - fixed instruction execution time (4 cycles)
//   - fixed instruction size (1 word)
//   - fixed operand size (1 word)
//   - all read, no write transfers
//
// Additional Statistics: none
//
// Simulation model:
//      1. Instruction fetch
//      2. Instruction decode
//      3. Operand fetch
//      4. Instruction execute
//
class cpu : public simmod {
// PORTS
  simmod    *Bus_port;       // pointer to input port of bus module
// MESSAGES
  tx_msg    *Tx_msg;         // pointer to transfer message (to/from bus)
// ATTRIBUTES
  sim_time  Cycle_time;      // CPU clock cycle time
// DISTRIBUTIONS
  i_const   *I_time;         // instruction execution time (clk cycles)
  i_const   *I_size;         // instruction size (in words)
  i_const   *O_size;         // operand size (in words)
public:
  // REQUIRED FUNCTIONS
  cpu();
  ~cpu();
  void  init_port(mod_ID id, simmod *bp)
        { ID = id; Bus_port = bp; }
```

```
  void  init_attr(sim_time ct=100.0)
        { Cycle_time = ct; }
  void  body();
};
NEW_(cpu, 1024);



// cpu1.cxx - simple CPU module (non-random)

#ifndef     cpu
#include    "cpu1.hxx"
#endif

cpu::cpu()
{
    Tx_msg = new tx_msg();
    I_time = new i_const("I_time", 4);  // 4 cycles/instruction
    I_size = new i_const("I_size", 1);  // 1 word/instruction
    O_size = new i_const("O_size", 1);  // 1 word/operand
}

cpu::~cpu()
{
    delete  Tx_msg;
    delete  I_time;
    delete  I_size;
    delete  O_size;
}

void cpu::body()
{
    while ( TRUE ) {
        //*** I-fetch
        Tx_msg->init_trans(ID, UNKNOWN, 16*I_size->sample());
        Tx_msg = (tx_msg*) reqack(Bus_port,Tx_msg); // send req, get ack
        if ( Tx_msg->get_dest() != ID )
            goto cpu_err;

        //*** I-decode -- no time used

        //*** O-fetch
        Tx_msg->init_trans(ID, UNKNOWN, 16*O_size->sample());
        Tx_msg = (tx_msg*) reqack(Bus_port, Tx_msg);
        if ( Tx_msg->get_dest() != ID ) {
cpu_err:
            mod_error("cpu::body",
             "Unexpected TRANSFER ack: not destination module", Tx_msg);
        }

        //*** I-execute
        hold(Cycle_time*I_time->sample());
    }
}
```

C.3. Bus Module

```
// bus1.hxx - simple bus module (fixed length transfers)

#ifndef     _SIMMOD_HXX_
#include    "simmod.hxx"
#endif

#ifndef     _MSGS_HXX_
#include    "msgs.hxx"
#endif

//**********************
//  Simple bus module  *
//**********************
//
// Time Base: 1 time unit = 1 nanosecond
//
// Module Attributes:
//  - bus cycle time: time to transfer one data unit (default: 100 ns)
//
// Module Characteristics:
//  - fixed bus width (1 word = 16 bits)
//  - fixed length transfers (1 word/request), each CPU transfer
//    request leads to exactly one memory access request
//  - only one tx request active at any time, other requests are queued
//
// Additional Statistics: none
//
// Simulation model:
//     1. wait for incoming message
//     2. process message:
//                   RECEIVED        ==>        SENT
//        a. CPU --transfer req--> Bus ---access req---> Mem
//        b. Mem ---access ack---> Bus --transfer ack--> CPU
//     3. transfer data (before or after access, depending on R/W)
//
class bus : public simmod {
// PORTS
  simmod*   Cpu_port;        // pointer to input port of CPU module
  simmod*   Mem_port;        // pointer to input port of memory module
// MESSAGES
  tx_msg    *Tx_msg;         // pointer to transfer message (to/from CPU)
  acc_msg   *Acc_msg;        // pointer to access message   (to/from mem)
// ATTRIBUTES
  sim_time  Cycle_time;      // bus clock cycle time (for one transfer)
// INTERNALS
  msg_queue *Pendingq;       // queue of pending bus requests
  boolean   Bus_avail;       // T if bus idle, F if transfer in progress
public:
  bus();
  ~bus();
```

```
  void  init_port(mod_ID id, simmod *cp, simmod *mp)
        { ID = id; Cpu_port = cp; Mem_port = mp; }
  void  init_attr(sim_time ct=100.0)
        { Cycle_time = ct; }
  void  body();
};
NEW_(bus, 1024);


// bus1.cxx - simple bus module (fixed length transfers)

#ifndef     _BUS1_HXX_
#include    "bus1.hxx"
#endif

bus::bus()
{
      Tx_msg = NULL;            // to be received from CPU
      Acc_msg = new acc_msg();
      Pendingq = new msg_queue;
      Bus_avail = TRUE;         // bus starts out free
}


bus::~bus()
{
      delete      Pendingq;
      delete      Acc_msg;
}


void bus::body()
{
      msg*  in_msg;             // holds incoming message

      while ( TRUE ) {

            //*** get/wait for incoming message
            if ( Bus_avail && Pendingq->length() ) {
                  // get first pending request
                  in_msg = Pendingq->first();
                  in_msg->remove();
            }
            else
                  in_msg = wait_msg();          // wait for next request

            switch ( in_msg->type() ) {
              case TRANSFER:
                  if ( in_msg->is_ack() )
                        mod_error("bus::body",
                  "Illegal message: TRANSFER acknowledge", in_msg);

                  if ( Bus_avail ) {      // accept transfer request
```

```
                    Bus_avail = FALSE;       // transfer in progress
                    Tx_msg = (tx_msg*) in_msg; // save transfer msg

                    //*** transfer data for write request
                    if ( not Tx_msg->Read )
                         hold(Cycle_time);

                    //*** send ACCESS request
                    Acc_msg->init_acc(ID, Tx_msg->get_send());
                    send_msg(Mem_port, Acc_msg);  // send access req
                    Acc_msg = NULL;               // message N/A
               }
               else {       // bus is busy
                    in_msg->insert(Pendingq);
               }
               break;

          case ACCESS:
               if ( in_msg->is_req() )
                    mod_error("bus::body",
                    "Illegal message: ACCESS request", in_msg);

               // make sure access acknowledge is valid
               if ( Acc_msg )
                    mod_error("bus::body",
        "Unexpected ACCESS ack: no ACCESS request issued", in_msg);
               if ( not Tx_msg )
                    mod_error("bus::body",
"Unexpected ACCESS ack: no corresponding TRANSFER request", in_msg);
               if ( in_msg->get_dest() != Tx_msg->get_send() )
                    mod_error("bus::body",
"ACCESS destination ID does not match TRANSFER sender ID", in_msg);

               Acc_msg = (acc_msg*) in_msg;  // save access message

               //*** transfer data for read request
               if ( Tx_msg->Read )
                    hold(Cycle_time);

               //*** send TRANSFER acknowledge
               Tx_msg->set_ID2(ID, Tx_msg->get_send());
               send_ack(Cpu_port, Tx_msg);
               Tx_msg = NULL;           // no current transfer message
               Bus_avail = TRUE;        // bus now idle
               break;

          default:
               mod_error("bus::body", "Unknown message received",
in_msg);
          }
     }
}
```

C.4. Memory Module

```
// mem1.hxx - simple memory module (no cycle time)


#ifndef     _SIMMOD_HXX_
#include    "simmod.hxx"
#endif

#ifndef     _MSGS_HXX_
#include    "msgs.hxx"
#endif


//*************************
//  Simple memory module  *
//*************************
//
// Time Base: 1 time unit = 1 nanosecond
//
// Module Attributes:
//   - memory access time: time to access one data unit (default: 80 ns)
//
// Module Characteristics:
//   - fixed length transfers (1 word/request)
//   - word length determined by bus interface
//   - only one memory access active at any time
//   - no cycle time
//
// Additional Statistics: none
//
// Simulation model:
//      1. wait for access request
//      2. access data
//      3. send access acknowledge
//
class mem : public simmod {
// PORTS
  simmod*   Bus_port;    // pointer to input port of bus module
// MESSAGES
  acc_msg*  Acc_msg;     // pointer to access message (to/from bus)
// ATTRIBUTES
  sim_time  Acc_time;    // memory access time
public:
  mem() { Acc_msg = NULL; Acc_time = 0.0; }
  void  init_port(mod_ID id, simmod *bp)
        { ID = id; Bus_port = bp; }
  void  init_attr(sim_time at=80.0)
        { Acc_time = at; }
  void  body();
};
NEW_(mem, 1024);
```

```
// mem1.cxx - simple memory module (no cycle time)


#ifndef     mem
#include    "mem1.hxx"
#endif


void mem::body()
{
    while ( TRUE ) {

        //*** wait for ACCESS request (exclusively)
        Acc_msg = (acc_msg*) only_msg(ACCESS);
        if ( Acc_msg->is_ack() )
            mod_error("mem::body", "Illegal message: ACCESS
acknowledge", Acc_msg);

        //*** simulate memory access
        hold(Acc_time);

        //*** return ACCESS acknowledge to bus module
        Acc_msg->set_send(ID);
        send_ack(Bus_port, Acc_msg);
    }
}
```

C.5. Module Description File

```
# Module library: /home/grad/thomasw/MSS/mdf/Example1.mdf

%AUTHOR T. Wieland
%CDATE  Sun Mar 18 00:10:39 1990
%MDATE  Sun Mar 18 00:10:39 1990

%BOF
3

#  Simple CPU module with non-random events
#  and fixed instruction & operand sizes.
#
#  Time unit is 1 nanosecond
%MODBEG cpu
cpu1
C

#  1 logical input port:
#  tx_ack from bus, message type: TRANSFER
%INPORT
1
tx_ack  tx_msg
```

```
#  1 logical output port:
#  tx_req to bus, message type: TRANSFER
%OUTPORT
1
tx_req  tx_msg

#  not implemented
%CONTROL

#  not implemented
%MEASURE

#  1 attribute:
#  Cycle_time   CPU cycle time; default: 100ns, range: 25 ns - 250 ns
%ATTRIB
1
Cycle_time  F   100
-1  25  250

#  1 port to initialize:
#  tx_req        1st argument
#
#  1 attribute to initialize:
#  Cycle_time    1st argument
%INIT
1
tx_req
1
Cycle_time

%MODEND

#  Simple bus module with fixed length transfers,
#  1 transfer request from CPU per memory access
#
#  Time unit is 1 nanosecond
%MODBEG bus
bus1
B

#  2 logical input ports:
#  tx_req from CPU, message type: TRANSFER
#  acc_ack from memory, message type: ACCESS
%INPORT
2
tx_req  tx_msg
acc_ack acc_msg

#  2 logical output ports:
#  tx_ack to CPU, message type: TRANSFER
#  acc_req to memory, message type: ACCESS
%OUTPORT
2
tx_ack  tx_msg
```

```
acc_req acc_msg

%CONTROL

%MEASURE

#  1 attribute:
#  Cycle_time   Bus cycle time; default: 100 ns, range: 50 ns - 1 us
%ATTRIB
2
Cycle_time  F   100
-1  50  1000

#  2 ports to initialize:
#  tx_ack    1st argument
#  acc_req  2nd argument
#  1 attribute to initialize:
#  Cycle_time   1st argument
%INIT
2
tx_ack
acc_req
1
Cycle_time


%MODEND

#  Simple memory module; only access time, no cycle time
#  Time unit is 1 nanosecond
%MODBEG mem
mem1
M

#  1 logical input port:
#  acc_req from bus, message type: ACCESS
%INPORT
1
acc_req acc_msg

#  1 logical output port:
#  acc_ack to bus, message type: ACCESS
%OUTPORT
1
acc_ack acc_msg

%CONTROL

%MEASURE

#  1 attribute:
#  Acc_time memory access time; default: 80.0 ns, range: 50ns - 500ns
%ATTRIB
1
```

```
Acc_time    F   80
-1  50  500

#  1 port to initialize:
#  acc_ack  1st argument
#
#  1 attribute to initialize:
#  Acc_time 1st argument
%INIT
1
acc_ack
1
Acc_time

%MODEND

%EOF
```

C.6. Simulator Description File

```
# Simulator description:
# /home/grad/thomasw/MSS/sim/Example1.sim
#
# Module libraries used:
# /home/grad/thomasw/MSS/mdf/Example1.mdf

%AUTHOR   Thomas Wieland
%CDATE    Sun Mar 18 00:11:03 1990

# three module instances
%BOF
3

%RUNTIME

%LIBRARY
Example1.mdf

%MODBEG cpu 1
Cpu

%OUTPORT
tx_req  2

%ATTRIB
Cycle_time   100.0

%MODEND


%MODBEG bus 2
Bus
```

```
%OUTPORT
tx_ack  1
acc_req 3

%ATTRIB
Bus_width   16
Cycle_time  100.0

%MODEND


%MODBEG mem 3
Memory

%OUTPORT
acc_ack 2

%ATTRIB
Acc_time    80.0

%MODEND

%EOF
```

C.7. Simulator Source File

This simulator source file was generated with prompting,command line input, and tracing enabled.

```
// Example1.cxx - Simulation main program
//
// Creation Date: Fri Apr  6 11:37:43 1990
//
// Simulator description file: /home/grad/thomasw/MSS/sim/Example1.sdf
//


// Include simulation module declarations
#include    "simmod.hxx"
#include    "cpu1.hxx"
#include    "mem1.hxx"
#include    "bus1.hxx"


main(int argc, char* argv[])
{
      // Simulation length
      sim_time     rt = -1.0;

      // Variables for command line interpretation
      int          a;
      char*        ap;
```

```
// Variables for setting of trace levels
int         t;
boolean     tflag = FALSE;
static char tt[10] = { -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 };

// Get command line arguments
for ( a=1; a<argc; a++ ) {

      if (*(ap=argv[a]) == '-' ) {  // get options
            switch ( *++ap ) {
              case 'h': // show help
                  printf("Usage: %s [ options.. ] [ runtime ]\n",
                        argv[0]);
                  printf("Command line arguments:\n");
                  printf("\t-h\tdisplay usage information\n");
                  printf("\t-tn\tset trace level (n=0..9)\n");
                  printf("\truntime\tsimulator runtime\n\n");
                  terminate_simulation(FALSE);
                  exit(0);
                  break;

              case 't': // set SAMOC trace level
                  if ( sscanf( ++ap, "%d", &t) != 1) {
                        fprintf(stderr,
"warning: could not convert trace level for option '%s'\n", argv[a]);
                  }
                  else if ( t<0 || t>9 ) {
                        fprintf(stderr,
"warning: option '%s' ignored, trace level must be 0..9\n", argv[a]);
                  }
                  else {
                        tflag++;
                        tt[t] = t;
                        printf("*** Now tracing ");
                        switch ( t ) {
                          case 0:
                              printf("coroutines\n");
                              break;
                          case 1:
                              printf("lists\n");
                              break;
                          case 2:
                              printf("queues\n");
                              break;
                          case 3:
                              printf("entities\n");
                              break;
                          case 4:
                              printf("entity synchronization\n");
                              break;
                          case 5:
                              printf("events\n");
                              break;
```

```
                                default:
                                    printf("user trace level %d\n", t);
                                    break;
                            }
                        }
                        break;
                    default:
                        fprintf(stderr,
                  "warning: unknown option '%s' ignored\n", argv[a]);
                        break;
                }
            }
            else {      // read simulation length
                if ( sscanf(argv[a], "%lf", &rt) != 1 ) {
                    fprintf(stderr,
"warning: numeric conversion failed for argument '%s'\n", argv[a]);
                }
            }
    }

    // Prompt for simulation length
    while ( rt < 0 ) {
        printf("Enter simulation length (microseconds) => ");
            if ( scanf("%lf", &rt) != 1 )
                scanf("%*s");
    }

    if ( tflag )
        trace_on(tt[0], tt[1], tt[2], tt[3], tt[4],
                 tt[5], tt[6], tt[7], tt[8], tt[9]);

    printf("Running simulation for %lf microseconds\n", rt);

    // Create simulation modules
    cpu*  cpu12 = new_cpu("cpu");
    mem*  mem13 = new_mem("mem");
    bus*  bus14 = new_bus("bus");

    // Initialize module output ports
    cpu12->init_port(12, bus14);
    mem13->init_port(13, bus14);
    bus14->init_port(14, cpu12, mem13);

    // Initialize module attributes
    cpu12->init_attr(100);
    mem13->init_attr(80);
    bus14->init_attr(100);

    // Activate simulation modules
    cpu12->schedule(0.0);
    mem13->schedule(0.0);
    bus14->schedule(0.0);
```

```
    // Start simulation run
    hold(1000.0*rt);

    // Print custom reports for simulation modules
    cpu12->report();
    mem13->report();
    bus14->report();

    // Print SAMOC standard reports & terminate
    terminate_simulation();
};
```

C.8. Output Generated By Simulator

Partial reports are shown for a few simulator generated with different attribute values. All simulations are run for 1000 microseconds simulation time.

C.8.1. Default Simulator

Attributes: CPU cycle=100ns, bus cycle=100ns, mem access=80ns

```
                                   BINS
                                   ****

title       / reset at/ users/init/ max/ now/ av. free/ av. wait/qmax
CPU 0           0.000    2632     0    1    0     0.000   180.000    1
BUS 0           0.000    5264     0    1    0     0.000   139.924    1
MEM 0           0.000    2632     0    1    0     0.000   299.810    1

                                DISTRIBUTIONS
                                *************

title       /   obs/type      /parameters
I_time         1316 i_const    constant =          4
I_size         1316 i_const    constant =          1
O_size         1316 i_const    constant =          1
```

C.8.2. Simulator with Slower Memory

Attributes:CPU cycle=80ns, bus cycle=100ns, mem access=100ns

```
                                   BINS
                                   ****

title       / reset at/ users/init/ max/ now/ av. free/ av. wait/qmax
CPU 0           0.000    2778     0    1    0     0.000   200.000    1
BUS 0           0.000    5556     0    1    0     0.000   129.942    1
MEM 0           0.000    2778     0    1    0     0.000   259.849    1
```

DISTRIBUTIONS
****

title	/ obs/type		/parameters	
I_time	1389	i_const	constant =	4
I_size	1389	i_const	constant =	1
O_size	1389	i_const	constant =	1

Appendix D. Listings For Expanded Example Simulator

This appendix shows only the header files for the modules of the extended example simulator; implementation is straight-forward and follows the discussion in chapter seven. Other files, whose contents can be deduced from the source files and the discussion in chapters six and seven, have also been omitted. The files shown were generated during the session described in chapter seven, section 7.6.

D.1. CPU Module

```
// cpu1.hxx - simple CPU module (random events)

#ifndef     _SIMMOD_HXX_
#include    "simmod.hxx"
#endif

#ifndef     _MSGS_HXX_
#include    "msgs.hxx"
#endif

//*************** *******************
//  Simple CPU module (68020-like)   *
//**********************************
//
// Time Base: 1 time unit = 1 nanosecond
//
// Module Attributes:
//   - CPU clock cycle time (default: 100 ns, i.e. 10 MHz CPU clock)
//   - Probability of write operand (default: 25%, range: 0% - 100%)
//
// Module Characteristics:
//   - variable instruction execution time (4-25 cycles; 4*I_size+0..5)
//   - variable instruction size (1-5 words, uniform)
//   - variable operand size (1-2 words, uniform)
//   - user-selectable probability of a write transfer
//     occuring as part of the instruction execution
//
// Additional Statistics:
//   - number of instructions completed
//   - MIPS performance
//
// Simulation model:
//      1. Instruction fetch
//      2. Instruction decode
//      3. Operand fetch
//      4. Instruction execute
//      5. Operand write
//
```

```
class cpu : public simmod {
// PORTS
  simmod     *Bus_port;          // pointer to input port of bus module
// MESSAGES
  tx_msg     *Tx_msg;            // pointer to tx message (to/from bus)
// ATTRIBUTES
  sim_time   Cycle_time;         // CPU clock cycle time
  double     Write_pcnt;         // Percentage of write transfers
// DISTRIBUTIONS
  randint    *I_time;            // instr. execution time modifier
  randint    *I_size;            // instruction size (in words)
  randint    *O_size;            // operand size (in words)
  draw       *W_prob;            // probability of write operand
// STATISTICS
  count      *num_instr;         // records number of instructions executed
// INTERNALS
  int        curr_I;             // holds length of current instruction
  int        curr_O;             // holds length of current operand
public:
  // REQUIRED FUNCTIONS
  cpu();
  ~cpu();
  void       init_port(mod_ID id, simmod *bp)
             { ID = id; Bus_port = bp; }
  void       init_attr(sim_time ct=100.0, double wp=25.0)
             { Cycle_time = ct; Write_pcnt = wp;
               W_prob = new draw("W_prob", Write_pcnt/(double)100.0); }
  void       body();
  // OPTIONAL FUNCTIONS
  void       report();
};
NEW_(cpu, 1024);
```

D.2. Bus Module

```
// bus2.hxx - simple bus module (variable length transfers)


#ifndef    _SIMMOD_HXX_
#include   "simmod.hxx"
#endif

#ifndef    _MSGS_HXX_
#include   "msgs.hxx"
#endif


//**********************
//  Simple bus module  *
//**********************
//
// Time Base: 1 time unit = 1 nanosecond
//
```

```
// Module Attributes:
//  - bus cycle time: time to transfer one data unit (default: 100 ns)
//  - bus width: number of bits transferred in parallel (default: 16)
//
// Module Characteristics:
//  - bus width determines width of interface to passive module(s)
//  - variable length transfers (>=1 word/request), a transfer
//    request may lead to more than one memory access
//  - only one tx request active at any time, other requests are queued
//
// Additional Statistics:
//  - Bus busy time (in % of total simulation time)
//  - Histogram of length of transfer requests
//
// Simulation model:
//      1. wait for incoming message
//      2. process message:
//         a. CPU --transfer req--> Bus
//         b. repeat as necessary:  Bus ---access req---> Mem
//                                  Bus <--access ack---- Mem
//         c. CPU <-transfer ack--- Bus
//      3. transfer data (before or after access)
//
class bus : public simmod {
// PORTS
  simmod*    Cpu_port;        // pointer to CPU input port
  simmod*    Mem_port;        // pointer to memory input port
// MESSAGES
  tx_msg     *Tx_msg;         // pointer to transfer message
  acc_msg    *Acc_msg;        // pointer to access message
// ATTRIBUTES
  sim_time   Cycle_time;      // bus clock cycle time
  int        Bus_width;       // bus width in bits
// STATISTICS
  accum      *Bus_busy;       // records bus busy/idle times
  histo      *Req_length;     // records hist. of request len.s
// INTERNALS
  msg_queue  *Pendingq;       // queue of pending bus requests
  boolean    Bus_avail;       // T if bus idle, F if busy
  sim_time   Time_last;       // time of last change in state
  long       Bits_left;       // bits left to transfer
public:
  bus();
  ~bus();
  void  init_port(mod_ID id, simmod *cp, simmod *mp)
        { ID = id; Cpu_port = cp; Mem_port = mp; }
  void  init_attr(sim_time ct=100.0, int bw=16)
        { Cycle_time = ct; Bus_width = bw; }
  void  body();
};
NEW_(bus, 1024);
```

D.3. Memory Module

```
// mem2.hxx - simple memory module (access & cycle time)


#ifndef     _SIMMOD_HXX_
#include    "simmod.hxx"
#endif

#ifndef     _MSGS_HXX_
#include    "msgs.hxx"
#endif


//*************************
//  Simple memory module  *
//*************************
//
// Time Base: 1 time unit = 1 nanosecond
//
// Module Attributes:
//  - memory access time: time to access one data unit (default: 80 ns)
//  - memory cycle time:  min. time between accesses (default: 160 ns)
//
// Module Characteristics:
//  - fixed length transfers  (1 word/request)
//  - word length determined by bus interface
//  - only one memory access active at any time
//  - next memory access blocked until cycle time has
//    elapsed since the start of the previous access
//
// Additional Statistics: none
//
// Simulation model:
//      1. wait for access request
//      2. if necessary, delay access to account for cycle time
//      3. access data
//      4. send access acknowledge
//
class mem : public simmod {
// PORTS
  simmod*   Bus_port;    // pointer to input port of bus module
// MESSAGES
  acc_msg*  Acc_msg;     // pointer to access message (to/from bus)
// ATTRIBUTES
  sim_time  Acc_time;    // memory access time
  sim_time  Cyc_time;    // memory cycle  time
// INTERNALS
  sim_time  Next_acc;    // earliest time of next access
public:
  mem()  { Acc_msg = NULL; Acc_time = 0.0;
           Cyc_time = 0.0; Next_acc = ::time(); }
```

```
  void  init_port(mod_ID id, simmod *bp)
        { ID = id; Bus_port = bp; }
  void  init_attr(sim_time at=80.0, sim_time ct=160.0)
        { Acc_time = at; Cyc_time = ct; }
  void  body();
};
NEW_(mem, 1024);
```

D.4. Output Generated By Simulator

Instead of repeating the long-winded report produced by the expanded simulator, we will present the results in a table. To verify the correct operation of our models, we have run several tests with different parameters.

The test systems in table D.1 below have been generated; each system varies one module attribute, so we can determine its effect on the system. The first system is the default configuration, which we use as a base line to compare against; the last system (H) varies all parameters simultaneously. In all simulators, the CPU *write probability* attribute was kept at 25%.

System	CPU cycle time (ns)	bus cycle time (ns)	bus width (bits)	mem. access time (ns)	mem. cycle time (ns)
A	100	100	16	80	160
B	100	100	16	120	200
C	60/100	100	16	120	200
D	60/60	100	16	80	160
E	100	60	16	80	160
F	100	100	32	80	160
G	100	60	32	80	160
H	60	60	32	80	120

Table D.1: Generated Test Simulators

While we will not discuss the results for these systems in any detail, we would like to note that the obtained results are intuitive. For example, increasing the amount of data the bus can transfer improves system throughput; conversely, slowing down the memory will lower it etc.

System	CPU1 MIPS	CPU2 MIPS	Avg CPU1 wait (ns)	Avg CPU2 wait (ns)	Bus Usage (%)	Avg Bus wait (ns)	Avg Memory wait (ns)
A	0.382	0.372	548	552	66.3	121	194
B	.339	.330	692	699	71.3	145	190
C	.495	.366	527	578	75.7	97	159
D	.460	.447	599	601	80.0	88	145
E	.416	.410	449	446	63.0	131	154
F	.476	.461	317	322	52.8	144	266
G	.498	.487	269	273	47.7	158	235
H	.666	.652	292	293	64.2	109	149

Table D.2: Results of the Simulator Test Runs