1

Micred

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

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.

**A Framework for Developing Object-Oriented
Real-Time Software**

by

Donald Legh Bailey, B.Eng.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Engineering

Department of Systems and Computer Engineering,
Carleton University,
Ottawa, Ontario, Canada

February 15, 1989

© copyright
1989 Donald Legh Bailey

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.

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 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.

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 0-315-51190-7

The undersigned recommend to the Faculty of Graduate
Studies and Research acceptance of the thesis

*A Framework for Developing Object-Oriented
Real-Time Software*

submitted by Donald Legh Bailey, B.Eng.
in partial fulfilment of the requirements for
the degree of Master of Engineering.

Thesis Supervisor

Thesis Supervisor

Chairman,
Department of Systems and
Computer Engineering

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
March 22, 1989

Abstract

This thesis describes a framework for developing real-time, multitasking, multiprocessor software using an object-oriented programming language. To support system design, we extend an existing iconic notation for describing concurrent systems so that it may be used to depict object-oriented systems. We then develop a model for designing multitasking, object-oriented software. This model is illustrated using our iconic notation. Issues involved in supporting an object-oriented language under a real-time kernel are addressed, and we discuss the implementation of a prototype framework using the Objective C language and the Unison multitasking operating system. Finally, practical considerations in developing object-oriented, real-time software are discussed.

Acknowledgements

I would like to thank my co-supervisors, Dr. J.E. Neilson (School of Computer Science) and Dr. C.M. Woodside (Department of Systems and Computer Engineering), for their advice and continuing encouragement. They have allowed me the opportunity to explore avenues of interest that were, at times, peripheral to the thesis research, and known when to suggest that it was time that I get back on track.

The folks at Multiprocessor Toolsmiths answered numerous questions about the inner workings of the Unison operating system. My thanks, then, to Kim Rowe, Doug Graham, and Alan Donenfeld.

Many thanks to my family, for their continuing support and encouragement.

I gratefully acknowledge the financial support provided by Carleton University.

Contents

1	Introduction	1
1.1	Approaches to Engineering Real-Time Software Systems	1
1.2	Thesis Objective	3
1.3	Summary of Contributions	4
1.4	Thesis Outline	6
2	Object-Oriented and Concurrent Programming	7
2.1	The Object-Oriented Paradigm	7
2.2	Smalltalk-80 and Object-Oriented Programming	10
2.3	Object-Oriented Concurrent Programming	12
2.4	Object-Oriented Design	14
2.5	Graphical Notations for Design	15
3	Objective-C	19
3.1	The External View	19
3.2	A Simple Example	22
3.3	The Internal View	24
4	The Unison Operating System	26
4.1	Real-Time Programming	26
4.2	Tasks	27
4.3	Task Communication and Synchronization	28
4.4	Unison Application Design	31
4.5	Design Examples and Notation	32
5	Describing the Design of Object-Oriented Programs	38
5.1	Structure Graphs	38
5.2	A Notation for Object-Oriented Program Design	44
5.3	An Example: Cox's Dependency Graph Application	53

6	A Model for Object-Oriented Real-Time Programs	67
6.1	Primitive Data Types and Objects	67
6.2	Considering Tasks as Objects	69
6.3	Implementing Tasks as Objects	71
6.4	An Example: Notifier Tasks	77
6.5	Example: The Bounded Buffer Task	88
6.5.1	First Attempt (Conventional)	88
6.5.2	Second Attempt (Buffer as an Object)	91
6.5.3	Third Attempt (Messages as Objects)	95
6.5.4	Fourth Attempt (Buffering of Objects)	101
6.5.5	Summary	105
7	Objective C Support for Unison	106
7.1	Language Support for Embedded Systems	106
7.2	Issues with Objective C	107
7.3	Class Definitions and Message Expressions	108
7.4	Object Implementation	109
7.5	Porting the Class Library	113
7.5.1	I/O Support in Unix and Unison	113
7.6	Substrate Functions	116
7.6.1	Memory Management	117
7.6.2	Filing Support	118
7.6.3	Error Handling and Program Termination	118
7.6.4	Messaging Support	119
7.7	Deallocation of Resources	120
8	Prototype Implementations	122
8.1	Tradeoffs Using Object-Oriented Development	122
8.1.1	Memory Requirements	123
8.1.2	Execution Penalties	125
9	Conclusions	128
9.1	A Summary of Thesis Research	128
9.2	Further Research	131
	References	133
A	Unison Icon Set	137

List of Figures

4.1	Structure Graph: Unison Server Paradigm	33
4.2	Unison Producer-Consumer Bounded Buffer	37
5.1	Booch's Notation for Ada Objects	40
5.2	Notation for Uses Relationships (Booch)	41
5.3	Icons Describing Objects	45
5.4	Sending a Message to an Object Class	48
5.5	Creating an Object Instance	48
5.6	Shorthand Notation for Instance Creation	49
5.7	Uses Relationships Between Graph, Node, and Set Classes (Booch)	50
5.8	Instances of Graph and Node Classes	51
5.9	Implementation of the Node Class Uses the Set Class	52
5.10	A Dependency Graph	54
5.11	Classes Used in the Dependency Graph Application	62
5.12	Implementation of the Node Class	63
5.13	Implementation of the Graph Class	64
5.14	doGraph(): Use of the Graph Instance	65
5.15	yyparse(): Building the Dependency Graph	66
6.1	Structure Graph: Simple Control System	73
6.2	Creating Tasks Through Objects	76
6.3	Creating Process Instances Within Tasks	78
6.4	Structure Graph: Use of Notifier Tasks	80
6.5	Structure Graph: Bounded Buffer, without Objects	89
6.6	Bounded Buffer, using the Buffer Class	93
6.7	Bounded Buffer, Using Message Objects	99
6.8	Bounded Buffer, Buffering Message Instances	103
7.1	C Structures Implementing Objects	110
A.1	Unison Icon Set: Inter-task Level	139
A.2	Unison Icon Set: Intra-task Level	141

Chapter 1

Introduction

1.1 Approaches to Engineering Real-Time Software Systems

As the field of software engineering has matured, a wide variety of techniques for producing software have been developed. These methods address various issues surrounding the need for increased programmer productivity, and are intended to assist the practitioner in producing software that is correct, reliable, maintainable and reusable.

In the past, techniques for analysis and design have been developed and retrofitted to existing languages. Using these methodologies, software developers can iterate through the system design and evaluation phases, correcting problems before implementing the software in the selected language. The *structured techniques* [15, 47] found widespread acceptance as one approach to designing complex sequential systems in a variety of procedural languages. More recently, numerous programming languages have been developed to exploit one or more new computational paradigms.

In this thesis, we are concerned with the branch of software engineering known as real-time or embedded systems development. Advances in this field reflect developments in many areas of software engineering. The establishment of a solid body of theory in the area of operating systems has led to the development of a number of real-time executives [9, 10, 16, 31] for the management of multiple resources in

a real-time, concurrent environment. Research into language and compiler design, in conjunction with the rapid evolution of microprocessor architectures, has yielded several languages which may be efficiently executed within a relatively inexpensive hardware environment.

While the embedded systems designer can realize substantial increases in productivity through the use of real-time executives and high level languages during the implementation phase of a project, methods for the analysis and design of such systems are not commonplace. In part, this is a reflection of the fact that concurrent systems are more difficult to design than sequential systems. A typical real-time system is often implemented using a high level language originally intended for sequential programming; the mechanisms for dealing with concurrency are provided by the separate real-time executive. Unfortunately, the reference materials supplied with most commercially available executives are often limited in their scope, and only detail the particular set of operating system services provided by the kernel; descriptions of an overall design philosophy are usually absent. The end result is that applications are likely to be developed in an ad-hoc manner, and the resulting code may be difficult to reuse and maintain.

Recent research has led to the development of a number of methods for designing real-time systems [8, 7, 17, 21, 23, 40, 41]. Although these techniques are largely independent of any particular language or real-time kernel, practical applications of these methods have often used commercially available procedural languages and real-time kernels as the basis for implementation. Of interest to us, some of these methods rely heavily on the use of graphical, iconic notations to convey elements of the design [8, 7, 23].

In the past decade, several computing paradigms have gained favour as viable alternatives to the procedural paradigm supported by the more commonplace programming languages. In particular, those languages which encompass the object-oriented paradigm [11, 12, 18, 36, 37, 42] have been touted as leading to increased productivity by providing for increased reuse of code [11, 12, 29]. In this paradigm, software systems are modelled as autonomous objects which encapsulate both data and the algorithms which manipulate that data. Languages such as Smalltalk-80 [18] and

Objective C' [11] provide mechanisms by which objects may be designed to share the characteristics of other, similar objects.

In order to support this particular model of computation, the underlying implementation of many object-oriented languages on conventional microprocessor-based architectures has been less efficient than that of older, procedural languages [11]. Applications developed using an object-oriented language have typically required more memory, and executed more slowly, than equivalent programs implemented using a language like C or Pascal. Embedded systems are often constrained by having limited amounts of memory, and the ability to react to events in real-time is usually critical [16]. As such, the implementation of real-time systems using object-oriented languages has not been considered practical. In this thesis, we develop a framework for designing and implementing real-time object-oriented systems.

1.2 Thesis Objective

The objective of this thesis is to examine ways in which we can improve the design and implementation of real-time systems. Specifically, we develop an approach to implementing real-time systems using an object-oriented language, supported by a graphical notation for describing object-oriented system designs. Two factors have influenced the direction of this thesis:

1. Given the evidence of substantial productivity gains when using object-oriented languages for the development of sequential programs, we wish to exploit such gains while implementing real-time systems.
2. Given the existence of an accepted conceptual framework for the design of concurrent systems which is supported by an iconic notation for describing design [7], it is desirable that our approach to designing object-oriented systems be integrated with this technique.

1.3 Summary of Contributions

The major developments resulting from our research include:

- adaptation of the iconic notation developed by Buhr [8], with extensions to represent the structural design of systems developed using the object-oriented paradigm;¹
- a model for the development of real-time systems which integrates techniques for designing concurrent and object-oriented systems, and is expressed using our iconic notation;
- porting the run-time system of the Objective C language [33, 34] to run on top of the Unison real-time operating system [31, 32].

Each of these contributions is described below:

Iconic Notation for Object-Oriented Design

The use of an iconic notation as one method for describing the design of complex systems is meeting with increased acceptance in various software communities [8, 7]. This approach does not appear to have infiltrated the object-oriented programming community, in part, we suspect, because most object-oriented programming environments have been used for “programming in the small” [13]. The use of icons as a component of the design process is one approach that is well suited to large scale projects. The significance of this contribution is two-fold:

¹Our research has used the design notation presented in Buhr’s *System Design with Ada* (SDWA) [8] as a starting point. Recent research by Buhr in the area of visual prototyping [7] has lead to the development of the a revised notation (“Machine Charts”) which is not as closely associated with the Ada language. This revised notation is better suited than the older SDWA notation to describing systems implemented using object-oriented languages. The idioms used to support object-oriented systems within the Machine Charts notation were suggested, in part, by our thesis research. The reader should note that while many of the icons for describing object-oriented systems presented in this thesis are not the same as those used in Machine Charts, the diagrams we present could be mapped in a straightforward manner into pictures using the Machine Charts notation

- we have extended an existing notation that has been adopted by a number of software developers. Exploiting this new notation does not require the designer to discard the methods supported by earlier versions of the notation;
- our notation provides a set of visual idioms for describing object-oriented systems. We see potential in this approach for teaching concepts in object-oriented software engineering, as well as describing the design of large object-oriented systems.

Model for Real-Time Object-Oriented Designs

We have found that there are a number of similarities in the approaches used to design real-time, multitasking systems and object-oriented software. We suggest that multitasking systems should be designed as a set of concurrent objects, and that each of these objects should in turn be composed of a number of passive, non-concurrent objects. As a practical example, we suggest an approach in which a multitasking system implemented using the Unison kernel can exploit objects supported by the Objective C language. This holds the promise of increased reusability of software developed for real-time applications.

Porting Objective C to Unison

Not only does the syntax of an object-oriented language differ from a well known language like C or Pascal – the model of behaviour during execution is quite different. Characteristics of an object-oriented application which must be supported at run-time include:

- dynamic creation and deletion of object instances;
- messaging between objects;
- location and execution of code (methods) inherited by an object from another object (a superclass), and execution of methods defined in an object which override those defined in a superclass.

The software provided with Objective C to support the run-time behaviour of objects has been modified so that it may be used with applications written for the Unison operating system. This provides us with a functional testbed on which we may prototype practical real-time, object-oriented systems.

1.4 Thesis Outline

In the following chapter we describe the concepts underlying the design and implementation of object-oriented and concurrent systems, and describe recent research in this area.

The next two chapters discuss the tools used to construct the framework arising from this thesis. Chapter 3 provides an overview object-oriented programming in general, and describes the Objective C programming language. Chapter 4 discusses the Unison real-time operating system, and presents a design process used to develop Unison applications.

In Chapter 5, we present a graphical notation for describing the structural design of object-oriented programs. This notation is then used in Chapter 6, as we describe and evaluate alternative models for developing object-oriented, real-time systems. Chapter 7 describes the issues involved in supporting the Objective C language in an embedded system environment, and discusses implementation details associated with porting the language run-time routines to the Unison operating system. There are a number of practical considerations when implementing a prototype framework; these are presented in Chapter 8. Finally, Chapter 9 presents conclusions and suggests areas for further research.

Chapter 2

Object-Oriented and Concurrent Programming

In this chapter, we provide a definition of the term *object-oriented* as it is used in this thesis. Recent work in the area of object-oriented concurrent programming is reviewed, and compared with the goals of our research. Methods by which object-oriented programs are designed are presented, and compared with the traditional structured methods for software design. Finally, the use of graphic notations as part of design methodologies is discussed.

2.1 The Object-Oriented Paradigm

Use of the term *object-oriented* [11, 12, 18, 36, 37, 42] is spreading rapidly throughout the software engineering literature. It is used to describe, among other things, programming languages, design methodologies, databases, and operating systems. There is, however, no single accepted definition of the phrase.

The Smalltalk environment [22, 18, 14, 44] is often cited as being the prototypical example of the object-oriented metaphor. Yet, within the programming community, there are a numerous views as to what consitutes an object-oriented language. Rentsch has stated:

What is object-oriented programming? My guess is that object-oriented

programming will be in the 1980's what structured programming was in the 1970's. Everyone will be in favour of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no one will know exactly just what it is.

Wegner discusses the characteristics of the object-oriented paradigm in [42]; he coins the terms object-based and class-based to describe programming languages which contain a subset of the features found in object-oriented languages.

Underlying object-oriented development is the concept of an *object* as an entity (or module or component) which encapsulates both data and the operations which manipulate that data. In hardware terms, objects are treated as "black boxes" which are connected to build "systems". An object is viewed in terms of the operations it provides, and the internal details of its implementation are hidden from the user of the object. It is unnecessary for an object's implementation to be understood in order to use it. An object receives requests to perform operations, and may in turn request operations from other objects. Objects support abstraction, providing a means to manage the complexity of software designs.

Object oriented programming differs from programming using the more familiar procedural languages such as C or Pascal, not only in syntax, but in the way that programs are structured. To design programs for C or Pascal, a traditional approach has been to apply functional decomposition [47]. This involves partitioning a program into modules, with each responsible for a phase of the overall processing. Each of these modules may in turn be refined and decomposed into further modules. Functional decomposition methods concentrate upon algorithmic abstractions, and older languages provide structural blocks (procedures) to support these methods. Functional decomposition does not, however, adequately support data abstraction. Object oriented programming addresses this issue.

Before exploring the object-oriented metaphor further, we briefly describe how procedural languages handle data. Languages like C and Pascal support the declaration of primitive data items, through what historically have been referred to as "type definitions". Following the categories used by Booch[4], these primitive types

may be classified as follows:

- *atomic types*, which are efficiently implemented by the underlying processor architecture. For example, a “character type” is usually contained in an eight-bit byte. An “integer type” usually corresponds to the “natural” register size of the processor.
- *composite types*, including arrays, which contain a number of homogeneous items, and records, which support the grouping of heterogeneous elements.
- *pointers*, which support indirect access to data items.

The major problem with these elementary types is that they do not provide a sufficiently high level of abstraction to describe many of the entities that we wish to model in our programs. Using the elementary types, we can define data structures which implement these abstractions.

In programs designed by functional decomposition, the data structures are often global to several of the modules. A change to the definition of a data structure may necessitate changes to several of the program modules. Maintainability suffers, and resusability of code is limited, by the extent to which modules depend upon one another.

The question of how to organize large programs to improve their reliability and maintainability was examined extensively in the 1970's. This lead to the concept of *information hiding* [35], which decreed that no module in a program should “know” more design decisions than necessary. When applied to managing data structures, a natural evolution of principle was the *abstract data type* (ADT) [19, 27].

Newer languages support ADTs; perhaps the best known are Modula-2 [43] and Ada [38]. In fact, these languages can only be used effectively when the principles of information hiding and abstraction are used to guide the design.

Object oriented programming provides an even more rigorous approach to data abstraction. Abstraction and information hiding are central to the notion of object-oriented programming. These ideas were combined with the notion of *classes* (originating with the language Simula [6]) and extended in the Smalltalk language.

2.2 Smalltalk-80 and Object-Oriented Programming

To many, the Smalltalk-80 environment is synonymous with object-oriented programming. During the research for this thesis, we have applied the object metaphor as defined by the Smalltalk in our designs (to the extent that this is supported by the Objective C language.) In this section, we present a more thorough overview of the Smalltalk's object-oriented paradigm, to provide the background for the remainder of this thesis. The vocabulary used is commonly associated with Smalltalk, but as it has been adopted by other object-oriented languages, this discussion is applicable to those languages.

Smalltalk is designed around the notion of communicating objects. An *object* encapsulates data in its private memory and the operations that manipulate that data.

Objects communicate by sending *messages* to each other. A message indicates the operation that an object should perform, but does not specify how to perform that operation. The data contained in an object's private memory can be accessed only by the operations that an object provides, and these operations are invoked only when an object receives a message.

Upon receiving a message, an object executes the piece of code (called a *method*) associated with that message. A method is analogous to a function in a procedural language which manipulates an ADT's data structure. When the execution of a method terminates, an object is returned as the result. This is similar to the function return values in languages like C or Pascal. Note, however, that an object returned by a method may in turn be sent messages.

Every object has an internal state, as described by the values of the variables contained in the object. An object's state may change only when it receives a message and executes the corresponding method.

Smalltalk classifies objects according to their similarities and their differences. A *class* defines the behaviour of similar objects, by specifying the variables contained in the objects' private memories, and the code for the methods describing

the operations supported by the objects. Each object is an *instance* of some class. All instances of a class respond to the same set of messages, and have the same set of methods associated with these messages.

An object's private variables are referred to as *instance variables*, and the data contained in these variables are also objects. An object's methods may access only that object's instance variables; they cannot manipulate the instance variables of any other object.

The object metaphor is applied uniformly throughout the Smalltalk environment, including what we earlier termed atomic types: characters and numbers are objects, and respond to messages to perform operations. Smalltalk provides a variety of objects which contain collections of other objects. These implement the common data structures: arrays, sets, and dictionaries, to name a few.

Smalltalk's classes are arranged in a tree-structured hierarchy. Each class has exactly one superclass, and it may have one or more subclasses. Each of these subclasses may in turn have subclasses. Class Object is at the top of the hierarchy; every class is a subclass of Object. Although every class has one superclass, several classes may share the same superclass, hence the tree structured class hierarchy.

Smalltalk supports the reuse of software by providing *inheritance*. A new class is defined as a subclass of another class. New instance variables and methods are defined within the subclass. An instance of an object contains all of the instance variables defined in its class, and in addition, it inherits all of the instance variables defined in its superclasses. An object also inherits all of the methods defined in its superclasses.

When an object is sent a message, Smalltalk first searches for the associated method in the object's class. If the method is found, its code is executed. If, however, the method is not defined in the class, Smalltalk attempts to locate the method in the class' superclass. This step is repeated for all the of the object's superclasses, until it reaches class Object. If the method is not found in the chain of superclasses, then the object has received a message it does not understand, and an error has occurred. As a result, an error message is usually displayed. This approach to determining, at run-time, the code associated with a message is referred to as

dynamic binding.

As noted earlier, a class describes the similarities of a related group of objects. Subclasses support specialization, by providing a means to describe how an object differs from the objects defined by its superclasses. Inheritance permits the construction of class hierarchies.

Another characteristic of object-oriented programming is *polymorphism*. Different objects may respond to the same message in different ways. Polymorphism allows us to add new classes of objects to an existing application, as long as these classes provide the methods required by the application. The application need not change to support the new classes; instead, it sends the existing message to the new object, which responds to the message in the manner appropriate for its class.

2.3 Object-Oriented Concurrent Programming

The previous description of object-oriented programming considered only the sequential behaviour of objects. In Smalltalk-80, an object that sends a message is blocked while the object that received the message computes. The sender resumes execution when the invoked method returns. After a method terminates, the receiving object remains inactive until another message is sent to it. In this way, the semantics of messaging between objects is equivalent to a procedure call.

Smalltalk-80 provides limited support for concurrency, by adding the notion of a process to an object. The syntax of the Smalltalk language provides a way to group a number of statements into a form called a *block*. A concurrent thread of execution is created by sending a “fork” message to the block. Communication between processes is through shared variables. Mutual exclusion and synchronization are provided by using semaphore objects. This approach to supporting concurrency within a programming language is quite primitive when compared with Concurrent Pascal’s monitor [20] or Ada’s rendezvous [38].

A relatively new and active area of research is the field of object-oriented concurrent programming. Research efforts in this area seek to expand the realm of object-oriented programming so as to exploit computational parallelism, drawing

upon the techniques for controlling concurrency developed in operating system and distributed database research.

Research into object-oriented concurrent programming has lead to the design of several programming languages. A number of these efforts are briefly described below.

ConcurrentSmalltalk [46] introduces concurrent constructs and a mechanism for execution synchronization to Smalltalk-80. Changes to the syntax and semantics of Smalltalk-80 have been mimimized to ensure compatibility with the language. ABCL/1 [45] allows for the modelling and programming of a variety of distributed, concurrent systems. POOL-T [2] is intended to support the design of systems which execute on highly parallel architectures. Presto [3] is a system for writing object-oriented parallel programs in a multiprocessor environment. The system provides objects to support this. Thread objects provide fine grained control of execution, while synchronization objects allow threads to coordinate with one another.

Research in the area of artificial intelligence has made a number of contributions to the object metaphor. The actor model of computation exploits message passing between concurrent actors (objects) as a basis for large-scale concurrent computation. Languages which implement this model include Act 1 [26] and Act3 [1].

While the research reported in this thesis deals with object-oriented programming, there are a number of differences with the research projects listed above. These papers address the syntactic and semantic issues of object-oriented concurrent languages. Most of these projects are not concerned with issue of concurrency in a real-time environment. In contrast, we are concerned with an existing object-oriented language which does not support concurrency (Objective C), and investigating its suitability for creating applications for an existing multiprocessor, real-time operating system (Unison).

2.4 Object-Oriented Design

Previous sections have examined the characteristics of the object-oriented paradigm, without considering the process by which object-oriented systems are designed.

Papers dealing with object-oriented programming tend to focus on the syntax and semantics of object-oriented languages, but do not provide detailed methodologies for designing object-oriented systems.

Object oriented programming is based upon the objects that exist in our model of reality. Booch states:

Each module in the [object-oriented] system denotes an object or class of objects from the problem space.[4, 5]

Typical steps involved in designing an object-oriented system are:

1. Identify the objects that appear in the description of the problem and its solution.
2. Classify the objects according to their similarities and differences.
3. Specify the messages to which each object will respond.
4. Implement the objects by defining its instance variables and methods.

Booch suggests a similar approach [4]:

1. Identify the objects and their attributes. Objects are often identified from the nouns used in a statement of the problem space. Objects with similar characteristics should be grouped in a class.
2. Identify the operations requested of and required by each object. The verbs used in the statement of the problem often correspond to operations.
3. Establish the visibility of each object in relation to other objects. This identifies the objects with which each object directly communicates.

4. Design the interface for each object. This defines the specification for every object's external view (that is, it explicitly specifies the messages to which each object class will respond).
5. Implement the internal representation of each object, including the algorithms for the interfaces defined in the previous step.

We have a number of concerns with this approach. First, there is the question of how to choose the “right” objects for an application. Since the objects in a program correspond to the objects we identify in our model of reality, this would appear to be a straightforward task. The “noun-verb” approach mentioned above assumes that a complete description of the problem space exists. Creating this description is part of the activity known as “systems analysis”; the object metaphor does not deal with this [4]. For complex systems, building this model of reality during the analysis phase is a major undertaking.

There are difficulties with the classification of objects. The Smalltalk-80 environment provides a rich collection of classes. Considerable experience is needed to learn the class hierarchy, and the operations provided by each class.

A Smalltalk program consists of numerous communicating objects; often there is no readily visible single thread of control as found in a program developed through functional decomposition. How, then, does one obtain an overview of a Smalltalk program, short of examining the code for all the objects? As part of our thesis research, we have suggested that graphical notations are well suited to describing object-oriented systems.

2.5 Graphical Notations for Design

Structured programming techniques have been widely applied since the early 1970's and, as an alternative to ad-hoc approaches to coding, have improved the quality of the software implementations, but these techniques alone are not sufficient to guarantee the production of reliable, maintainable systems. As a software project proceeds from the design phase to the coding and debugging stages, the cost of

altering the software increases rapidly. As such, it is crucial that a correct design exist prior to the start of coding.

Design methodologies have been developed in order to cope with the complexities involved in designing large software systems. The *structured analysis* and *structured design* methods developed by Yourdon and others have found wide acceptance [17, 15]. Part of the success of these methods stems from their use of diagrams to express aspects of a system's requirements and design. Following the adage that "a picture is worth a thousand words", graphic notations provide a means to quickly create and convey design information. Structured analysis methods make use of data flow diagrams to represent a system's functional requirements. Structured design methods are used to describe the design of software systems; the diagrams used here are called structure charts, and show the organizational hierarchy of the modules in the system.

These methods have been widely applied to the creation of large, sequential applications, but were found not to be entirely suitable for the design of real-time systems. In particular, the structured methods do not provide a means of expressing time-critical operations and the concurrency present in such systems. To meet the needs of real-time design, the structured methods have been extended to address these issues [21, 40, 41], and additional graphical notations have been developed to support the extensions. These include control flow diagrams and state transition diagrams which are used to illustrate control and timing requirements.

Yourdon's structured design methods include a process by which the data flow diagrams created in the analysis phase are transformed into the system's structure charts. This transformation involves functional decomposition, which, as we described earlier, is less desirable than object-oriented decomposition.

A graphic notation for the description of object-oriented programs would aid in the understanding of these systems. A traditional structure chart is not suitable for this purpose. First, a structure chart's module is not sufficient for describing an object. We need a means of showing the grouping of an object's data and its operations. Second, a structure chart does not provide the means to illustrate the class hierarchies within object-oriented systems. Third, a notation for object-oriented

programs must provide a means for showing classes and instances of those classes. Fourth, the structure chart presents a static view of a program's organization. An object-oriented program is a dynamic entity. Objects are created and freed throughout the execution of the program.

Booch has described a method for the object-oriented design of Ada programs [4, 5]. His approach has been strongly influenced by Smalltalk's view of the object metaphor. Booch presents a graphical notation to describe objects as implemented using the constructs available in Ada. We believe his approach is useful for identifying the classes of objects in a system and the visibility between them; however, a significant amount of information is not presented. In particular, the dynamic nature of object creation and deletion not shown. Inheritance is not adequately depicted, in part because Ada's support for inheritance is limited to generics [38]. Booch's approach is sufficient to identify the packages and tasks used in an Ada program and the dependencies between them; however, they are not intended to describe the structure of the program in any detail.

Buhr has developed a notation for the object-oriented development of concurrent Ada programs [8]. This notation depicts the tasks and packages provided by the Ada language and the interactions between those entities. He adopts the view of objects as "black boxes" which may be "plugged together" to form systems. Using this iconic notation, the interface that an object presents to other objects is clearly defined and separated from the implementation details of that object. This notation supports rapid and convenient evaluation and refinement of alternative designs; the mapping of the final design into the Ada language is straightforward. The notation in [8] is tied to Ada, and is not sufficient to represent many of the concepts present in the object-oriented paradigm we have described in this section. Notations for depicting inheritance and data abstraction are not supported. (Buhr's Machine Charts [7], a revised and extended version of his earlier notation which is independent of the Ada language, provides a number of icons which may be used to describe systems implemented using an object-oriented language. The relationship between object-oriented programming systems and this iconic notation was, in part, suggested by the research described in this thesis).

We have found that the original “Buhrgrams” provide a cleaner representation of the concurrency and structural partitioning of real-time systems, when compared with the diagrams proposed by Hatley or Ward [21, 40, 41]. As such, we have used Buhr’s notation as a starting point for exploring the graphical representation of object-oriented systems. This notation is discussed in Chapter 5. In Chapter 6, we refine this notation to illustrate issues in object-oriented concurrent systems.

Chapter 3

Objective-C

Objective-C [11, 33, 34] is a hybrid programming language based on the C language [24]. It extends the syntax of C to provide object-oriented capabilities. The syntax and semantics of Objective-C's style of object-oriented programming are strongly influenced by Smalltalk-80. The language provides the object-oriented paradigm discussed in Chapter 2: it supports objects, classes, inheritance, dynamic binding and polymorphism.

In this chapter, we explain the syntax of Objective-C language in sufficient detail to allow the reader to follow the examples presented in later chapters. We do not attempt to cover all aspects of the language; in particular we do not describe the class library supplied with the language.

3.1 The External View

First, we examine the external view of Objective-C objects, that is, we describe the syntax used to create object instances and send messages to these objects.

Objective-C adds one data type to C, the *id*, which identifies objects. This type is used in C expressions to declare variables which contain object identifiers. An object identifier does not distinguish between the different classes of object; a variable of type *id* identify any kind of object.

The C code fragment:

```
int counter;
id anObject;
```

declares that `counter` is a variable of type `int`, and `anObject` is a variable of type `id`. Any integer value can be stored in the variable `counter`, and similarly `anObject` may contain the identifier for any object.

(As an aside, we note that Objective-C uses the physical address of the data structures that implement an object as its identifier. The compiler defines type `id` as a pointer to these structures; however, the details need not be understood in order to use objects.)

Operations are requested of objects by writing message expressions. This provides the syntax for sending a message to an object, and obtaining the reply returned by the object. In Objective-C, message expressions are written in square brackets. Message expressions contain the name of the object that receives the message, the name of the message (also known as the selector), and any optional arguments of the message. For example, the message expression

```
aSet = [Set new];
```

sends the message `new` to the object `Set`¹. Here, `Set` is the receiver, `new` is the selector, and the reply from object `Set` is stored in the variable `aSet`. Message expressions may be written anywhere function calls may be used in C code.

Following the terminology used by Smalltalk, Objective-C message expressions can be classified as unary expressions or keyword expressions². Unary expressions have no arguments; for example:

```
[aReport displaySummary];
```

Keyword expressions have arguments. The message selector is made up of one or more keywords, each one followed by an argument. Each keyword ends with a colon. As an example:

¹The message expression syntax should not be confused with Smalltalk-80's block declaration syntax, which also uses square brackets. Objective-C appears to have adopted the syntax of Smalltalk-76 [22], which used the same syntax for messaging.

²Smalltalk-80's binary message expressions are not supported.

```
[aMemo sendTo:address];
```

is a keyword expression with selector `sendTo:` and argument `address`.

```
[aMemo sendTo:address withPriority:high];
```

has the keyword selector `sendTo:withPriority:`. This message expression has two arguments, `address` and `high`; one argument is written after each keyword.

Message expressions may return values, in the same manner as C functions. As with C functions, the return values have types.

```
id aSet
aSet = [Set new];
```

returns a value of type `id`, which is stored in variable `aSet`, also of type `id`, while

```
int n
n = [aSet size];
```

returns a value of type `int`.

In an Objective-C program, all objects are instances of the classes used by the program. Objective-C uses the term *factory object* to refer to a class. As in Smalltalk-80, a class is an object, and is sent messages to create initialized instances of the class. There is one factory object for every class used in a program; these are created by the Objective-C compiler and installed in the program's executable image by the linker.

In the examples above, the new message is sent to the factory object `Set`. `Set` returns an initialized instance of the `Set` class, and the object identifier for that instance is stored in the variable `aSet`.

Names of factory objects are global to the program, and C's `extern` statement is used to make the names visible within a file, in this way:

```
extern id Set;
```

The following conventions are used when naming factory objects and object instances:

1. Factory objects names always start with an upper case letter; for example: Set.
2. Factory object names describe the class of objects they produce.
3. Instance names always begin with a lower case letter. If an instance name is composed of multiple words, then the first letter of the second and subsequent words are capitalized. In a previous example, we named the instance aSet, not ASet or aset.

3.2 A Simple Example

In this section, we use the syntax presented so far to present a simple example. Consider a factory object (class) named Pen, which produces instances of pen objects.

The code fragment to create a pen is:

```
extern id Pen;
id aPen;
aPen = [Pen new];
```

The instance aPen is an object which responds to messages. Note, however, that an instance understands different messages than the factory object which created the instance. For example, the object aPen does not understand the new message and cannot create new pen instances. Instances of pens are created only by sending messages to the factory object Pen.

When a factory object creates an instance we often request it to initialize the instance in a particular way. The following code creates three pens, each a different colour:

```
id redPen, bluePen, blackPen;
redPen = [Pen red];
bluePen = [Pen blue];
blackPen = [Pen black];
```


We can send messages to pens, requesting them to report information about themselves. That is, we ask about the state of the pen objects:

```
penColour = [aPen colour];
currentLocation = [bluePen location];
```

Pens respond to messages to draw images; these messages change the state of the object:

```
[redPen up];
[redPen goTo:newPosition]; //move the pen, without drawing
[redPen down];
[redPen turn:90];           //turn the pen 90 degrees
[redPen go:100];           //draw a line 100 bits long
```

Extending this example, we can envision wanting to collect pens and pencils in a pencil cup. A cup is an instance of a Container, which is an object that can contain other objects.

```
extern id Container, Pen, Pencil;

...

id pencilCup = [Container new];
[pencilCup add:bluePen];
[pencilCup add:aPen];
[pencilCup add:[Pencil new]];
```

The last line contains a nested message expression. The innermost expression sends a message to Pencil to return a new pencil instance; the new object is then used as the argument of the add: message.

In this example, we use both Pen and Pencil factory objects. Pen may be a subclass of Pencil, Pencil may be a subclass of Pen, or both classes may be a subclass of some DrawingImplement class. Notice that this information is not required in order to use Pen and Pencil objects. All we need to know is what messages each of these objects recognize. Also, notice that the object pencilCup is not restricted to holding a particular type of object. We have shown it holding pen and pencil object, but it could be used without alteration to hold other kinds of objects.

3.3 The Internal View

We now deal with how classes are coded in Objective-C. The code for Objective-C classes reside in class description files; a separate file is used for each class. A class description file defines:

- the name of the class;
- the class' superclass;
- the declarations of the instance variables;
- the code for the methods.

Each class description file begins by identifying the name of the class, its superclass, and the instance variables. The syntax is:

```
= Pen:BitBlt { COLOUR penColour; id position; int heading;}
```

Pen is defined as a subclass of class BitBlt. Pen defines three instance variables: penColour, position, and heading. In addition, Pen inherits all of the instance variables and methods of its superclass. (A C typedef, not shown, would be used to declare the type COLOUR in an appropriate manner.)

A class declaration file describes two different kinds of object: the factory object for the class, and the instance of the class. The instance variables introduced above are, of course, associated with the instances. Two kinds of methods are implemented in the class description file: factory methods and instance methods.

A method definition describes:

- whether it is an instance method or a factory method;
- the method's selector, the names and types of its arguments, and the return type of the method; and
- the implementation of the method. These resemble C functions, but can include message expressions.

An instance of class `Pen` responds to requests to report its colour and change its colour. The instance methods are written as:

```
- (COLOUR)colour { return penColour; }

- colour:(COLOUR)newColour {
    penColour = newColour;
    return self; }
```

Note that the message selectors for these methods are not the same. Selector “colour” has no arguments, while “colour:” is a selector requiring exactly one argument.

By convention, if a method has anything useful to return, it returns that. Otherwise, the method returns the id of the object that received the message. In the first example, the color method returns the value of the `penColour` instance variable. In the second example, the `colour:` method returns the id of the pen object that receives the message.

The syntax used to define factory methods is identical to that used those for instance methods, except that the method names are prefaced with a `+` instead of a `-`. Often, no factory methods are defined in a class definition, because the methods provided by the superclass suffice. Again, we note that only factory objects execute factory methods, and instances of classes execute instance methods.

Chapter 4

The Unison Operating System

Our research was prototyped using Unison, a real-time, multitasking, multiprocessor operating system [31, 32]. The operating system is intended for use in a target environment containing several single board computers linked by a common bus. Unison applications are written in the C language. For each processor in the target system, the object modules for the code that is to run on that processor are linked with a copy of the Unison kernel. These executable processor images are then downloaded to the target system.

The multiprocessing support in Unison is transparent, in that processes executing on different processors communicate with each other in the same manner as processes executing on a single processor.

In this chapter, we describe the concept of *tasks* in Unison, and outline the mechanisms by which tasks communicate and synchronize. A pictorial notation for describing Unison designs is presented, and guidelines for the structuring of Unison programs are discussed.

4.1 Real-Time Programming

In this section, we describe what we mean by the phrase *real-time*. To start, real-time programs differ from conventional application programs. Real-time programs must respond to the occurrence of events in the real world, and these events may occur

asynchronously with the current state of processing of the program. For example, a hardware device, upon obtaining data from the external world, must inform the program of this event in order for the application to capture and process this data. Similarly, a hardware device may signal to the program that it is expecting data from the program to transfer to the outside world, or it may indicate that it has just completed such a transfer. As such, a real time program must synchronize with the external world, so that its computation may track the occurrence of these events. In Unison, this is accomplished by partitioning real-time applications into tasks and interrupt service routines.

A second way in which real-time programming is different is the notion that time constrains the program's behaviour. For example, it is usually necessary for a real-time program to respond to an event within a specified amount of time. Periodicity may be important, in that some computation must be performed at regular intervals, with minimal jitter. Finally, there may be a constraint on total throughput; that is, the number of events, and the associated computation, handled in each unit of time must meet some specified minimum.

To deal with these time constraints, a real-time application must be able to schedule the computations that it performs. If several computations are possible, there is the matter of deciding which computation to perform first. Another issue that must be addressed is whether the computation that is currently executing should be allowed to run until it completes, before starting a new computation, or whether the current computation should be preempted. The Unison kernel handles these scheduling issues through a well defined set of rules associated with the priorities of tasks, and the behaviour of the intertask communication and synchronization mechanisms. These are discussed below.

4.2 Tasks

A Unison application is composed of a set of tasks. These tasks may all be executed on a single processor, or they may be split up among several processors. Each task consists of a group of related procedures and performs the processing for some

well-defined portion of the real-time program.

A task is comprised of one or more C functions. Each instance of a task must be explicitly created by calling the kernel's `_create` primitive, passing it the address of the task's *root function*. When a task is instantiated the root function of the task is, in effect, called by the kernel. Each task instance is allocated its own stack when it is created. When a task is instantiated, it executes independently of, and concurrently with, other tasks in the system. The context of a task includes its stack, its root function, and any other functions that are called.

Each task executes sequentially and synchronously, in the same manner as a conventional program. However, at various times, tasks may synchronize and communicate with other tasks that are executing concurrently. True concurrency is only available on a multiprocessor system. Unison supports multiprocessing, and allows a task executing on one processor to request the creation of a task which will execute on another processor. (However, Unison tasks are bound to a particular processor when an application is built; it is not possible to select, at runtime, the processor on which a particular task will execute.)

Since several tasks may run on a single processor, Unison employs prioritized, preemptive multitasking to schedule tasks, so as to share the processor between them. In a multiprocessor system, tasks compete for scheduling only with other tasks running on the same processor. Each task has a fixed priority level. For each priority level, a separate first-in, first-out (FIFO) ready queue is maintained by the Unison kernel. All tasks at a given priority level which are ready to execute are in the corresponding ready queue. The task which is currently executing is referred to as the active task; it is the first task on the highest priority queue which is not empty. The active task continues to execute until it blocks to wait for an event, or until it is preempted by a higher priority task becoming ready.

4.3 Task Communication and Synchronization

As stated earlier, tasks execute synchronously and sequentially; all of the asynchronism and concurrency in an application is between the tasks.

In practical systems, tasks must synchronize and communicate with each other. The Unison kernel provides for task synchronization and communication through message passing. Specifically, *rendezvous-based* messaging is supported through *blocking send*, *blocking receive*, and *non-blocking reply* primitives.

In Unison, a message is defined to be a variable length, contiguous block of bytes. The first two bytes are treated as an unsigned 16 bit integer, and specify the length of the block.

The messaging primitives are provided by three functions:

```
id = _send(request, response, taskid, port_number, timeout);
id = _receive(request, portset, sigset, &rcv_port, timeout);
id = _reply(response, taskid);
```

A task prepares to send a message to another task by storing the message in a buffer pointed to by the request argument. The space pointed to by the response argument is initialized (that is, the length of the response buffer is stored in the first two bytes of that buffer); this is the space into which a reply message will be stored. The task then calls the *_send* primitive. The *task_id* argument specifies the task to which the request message is to be sent. The *port_number* indicates the port through which the correspondent task will receive the message.

When the call to *_send* returns, the value returned by the function is the id of the correspondent task (that is, the receiver). The response buffer contains any message replied by the receiving task.

Tasks receive messages using the *_receive* primitive. A task prepares to receive a message by initializing the message buffer pointed to by the request argument, into which the received message will be copied. A task which receives messages has up to 32 ports, numbered 0 through 31. These provide for a selective accept mechanism, similar to that provided by Ada [38]. Specifically, the *portset* argument is a 32 bit mask; each bit set in the mask indicates that the task is willing to receive a message through the corresponding port. Messages are enqueued on each port, with new messages placed at the rear of the queue. If there are messages waiting on more than one port, and the receiving task has set the port mask to receive messages

from multiple ports, then there is no mechanism to indicate that the message on one port should be received in preference to a message on another port.

When the call to `_receive` returns, the value returned by the function indicates the correspondent task (that is, task id of the sender is returned). The sender's message will have been copied into the receiver's request buffer, and `rcv_port` will contain the number of the port through which the message was received.

After the receiving task has processed the sender's message, it must reply to the sending task. The receiver builds a reply in the buffer pointed to by `response` and calls `_reply`. It is not necessary to reply to sending tasks in any fixed order; as such, the id of the task which is being replied to is passed as an argument of `_reply`. The id of the correspondent task is returned by the function.

Earlier, we stated that Unison provided rendezvous-based message passing. The behaviour of the rendezvous is provided by the blocking nature of the `_send` and `_receive` primitives. The sending task is blocked when the `_send` primitive is called, and remains blocked until the receiving task replies to the sender. The receiving task blocks when it calls `_receive`, if no task has as yet sent a message to the receiver through one of the ports specified in the `portset` argument. The receiving task is unblocked and returns from the `_receive` primitive when a message becomes available.

The `_reply` primitive does not block, because the sender to which it is replying is known to be blocked. Once the reply message is copied to the sender's response buffer, the sender is unblocked.

Notice, then, that the copying of the message from the sender's address space to the receiver's address space does not occur until both tasks have blocked, indicating that they are willing to rendezvous, by calling the `_send` and `_receive` primitives.

Also note that the message passing primitives provide the mechanism for both intertask communication and synchronization. Communication occurs via the messages that are received and replied. Synchronization is provided by the blocking behaviour of the primitives.

In this discussion of the message passing primitives, a number of arguments were not discussed. The `timeout` parameter provides the means through which conditional or timed rendezvous may be implemented. In the discussions so far, we have assumed

that the timeout argument was set to `FOREVER`, a constant indicating that the receiving task should remain blocked until a message is received. For the `_receive` primitive, a timeout value of 0 indicates that a polled receive should occur. If no messages are waiting when the `_receive` call is made, the call returns immediately, and the id returned by the function indicates that no message was received. A timeout value between 0 and `FOREVER` specifies how long that the receiving task is willing to wait for a message; the receiver unblocks either when a message has been received or when the timeout has expired.

Currently, timeouts on message sends are not supported. For the `_send` primitive, the timeout parameter must be set to `FOREVER`, so the task will block until its message has been received and it has been replied to.

Real-time application programs are often informed of the occurrence of outside events through hardware interrupts. Unison allows *interrupt service routines* (ISRs) to be installed to handle these interrupts; the details will, of course, differ between various hardware implementations. ISRs may schedule tasks to run by *signalling* them. The `_receive` primitive allows a task to wait for signals through the *sigset* argument. *Sigset* is a 32 bit mask indicating which signals that a task is willing to receive. Using the *portset* and *sigset* masks we can specify in a single `_receive` call that a task is willing to wait for both messages (through ports) and signals.

4.4 Unison Application Design

When designing an operating system or language that supports multitasking, the semantics of the mechanisms for task interaction often support a particular paradigm for the design and structuring of the applications. In order to effectively use an operating system such as Unison, it is not sufficient to merely understand the syntax of the message passing primitives.

The structuring of systems in which concurrent tasks interact via rendezvous-based messaging has been previously examined; these include Thoth [9, 17] and Harmony [16]. These use a design paradigm that may be effectively exploited by Unison applications, called *client-server* structuring. In this paradigm, tasks do not

access and manipulate a resource (for example, a data structure or hardware device) directly. Instead, one or more resources are considered to be owned by a single task. Other tasks send messages to the task which owns the resource, requesting operations upon the resource on their behalf. A task which manipulates a resource is considered to be a *server*, and the tasks which request operations from this task are referred to as *clients* of the server. The server enforces mutual exclusion upon the resource, because there is no concurrency within the server (the task executes sequentially), and messages sent to the server are received serially.

It is also possible for Unison tasks to share global data structures in memory accessible to all tasks. This approach is typically used with applications built upon an operating system that supports asynchronous message passing. When supporting this method with Unison, the inter-task Unison messages are used to pass pointers to the shared data, rather than the data themselves. While this approach avoids the overhead of copying large amounts of data between tasks, the protocol between the tasks must be carefully designed to ensure that mutual exclusion on the data structure is enforced. As such, it is usually preferable if this approach is avoided.

4.5 Design Examples and Notation

As an example of Unison applications in general, and the server paradigm in particular, consider a system containing a number of tasks. From time to time, each of these tasks must interact with a human operator via a console. The structure of this application is shown in Figure 4.1. The graphical notation is based on Buhr's *System Design with Ada* [8], as adapted for Unison.

The symbols used in this notation are described in Appendix A; however, for this example, the symbols in Figure 4.1 are briefly described:

- parallelograms represent Unison tasks;
- small parallelograms positioned at the edges of tasks correspond to a task's ports;

Figure 4.1: Structure Graph: Unison Server Paradigm

- lines connecting tasks and task ports indicate rendezvous between tasks. Each line implies a complete send/receive/reply cycle.
- labelled arrows with circular tails indicate data flow between tasks, and the direction of flow. These items of data are stored as components of the byte-string messages that are copied between tasks during the send/receive and reply operations.

For this discussion, the activities of the tasks Parent, Child1, and Child2 are not important; we assume that each performs some activity concurrently with the other tasks. We also assume that each of these tasks executes in an endless loop, cycling through its activities forever. Each of the tasks requires the services of the operator console from time to time. The Parent task sends status reports to the console, and the Child1 and Child2 tasks must occasionally obtain input from the operator.

Following the server paradigm, we use a `console_server` task to handle the console screen and keyboard.¹

The `console_server` task receives messages through a single port. In order to indicate the type of service they require, each of the calling tasks must pass a “message-type” indicator as part of the message sent to the server. The Parent task’s message is of type “display characters”, and requests the server to display the character string passed in the message. The messages sent by the child tasks contain an “obtain input” indicator and a prompt string to be displayed on the console screen. After prompting the user, the `console_server` waits for characters to be entered at the keyboard, and then returns the buffer of keystrokes as the reply message.

This example, although simple, illustrates the use of servers to manipulate shared resources within a concurrent system. All of the code and data structures required to support the console are encapsulated in the server task, ensuring mutual exclusion on this resource. The client tasks view the console as an abstract entity from which they request services, without being concerned with the details of how those services are provided.

¹The details of the interaction with the console hardware, including any interrupt service routines, have been omitted in order to simplify the example.

The notion of messages containing a type indicator is borrowed from concurrent system designs based upon Thoth or Harmony. With these two operating systems, the semantics of the rendezvous are slightly different than that described for Unison. In particular, Harmony and Thoth lack messaging ports. Instead, when a task chooses to receive a message, it must specify whether it is willing to wait for a message from a particular task or from any task. Once the message has been received, the message indicator is used to determine the request that has been made.

We have found that the Unison rendezvous, supporting selective acceptance of messages through ports, provides greater flexibility in structuring concurrent systems, while allowing us to retain the server/client paradigm for design. (As is often the case, flexibility is traded off against simplicity. Syntactically, Unison's message passing primitives are not as "clean" as those provided by Thoth and Harmony: this additional complexity must be handled in even the simplest programs.)

If we choose to build applications in the style supported by Harmony, then we may associate one port with each of calling tasks. A receive-specific operation is then performed by waiting for a message on a single port, corresponding to the single task with which we wish to communicate. A receive-any operation is performed by specifying that a message may be received through any of the task's ports. In this case, it is still necessary for the callers to place a message type indicator within the message (unless it is known beforehand that each of the calling tasks will only send a single type of message). Additionally, we have additional flexibility in that the receiving task may specify that it wishes to wait for messages from, say, three of a possible seven tasks, by calling the `_receive` primitive with the port mask set appropriately.

We may also choose to associate each port with a specific message type. This use of ports corresponds most closely to the named entries of Ada tasks. In this model, all tasks which wish to send a particular message to a task specify the corresponding port when they send the message.

This approach is now illustrated with another example, the well-known producer-consumer, bounded buffer problem. This design is illustrated in Figure 4.2 and introduces further graphic notation, as described below:

- A group of ports (such as Get, Put, and Count in task Buffer) connected by an intersecting line define a selective accept.
- A dot beside a port denotes a guard, indicating that depending on its internal state, a receiving task may choose not to wait on that port for a message.

The function of the three tasks is now described. The Producer task cycles endlessly, producing data items and sending them to the Buffer task's Put port to be stored. The Consumer task cycles endlessly, requesting data from the Get port of the Buffer task and using the data in some manner. From time to time, both the Producer and the Consumer tasks request the Buffer task to report the number of data items stored within it.

The Buffer task manages the storage pool of data items; it provides the server role for this design. It selectively accepts messages through the Put, Get and Count ports. Here, we have associated each port with a particular message type. The Put and Get ports are shown as being guarded. Obviously, the Buffer task should not accept messages through the Put port when the storage buffer is full, and messages through the Get port are disallowed when the buffer is empty. Messages requesting the current count of items stored in the buffer are allowed at any time.

Note that with this structure, there is no need for a message type to be included within the message itself.

Figure 4.2: Unison Producer-Consumer Bounded Buffer

Chapter 5

Describing the Design of Object-Oriented Programs

Programs developed using an object-oriented language differ from applications coded in a procedural language. Coding in an object-oriented language allows the programmer to take advantage of the existing library of classes through subclassing and inheritance. Individual methods coded in Smalltalk or Objective C tend to be quite short, and for typical application, the amount of new code produced by a programmer is much less than if the programmer were to produce the equivalent program in Ada or C. However, a common complaint is that it is difficult to obtain an overall understanding of the application's organization, without reading the code and having a thorough understanding of the language's class library.

In this chapter, we present a method for pictorially describing the design of object-oriented programs, which is based upon and extends the notation used by Buhr to represent Ada programs [8, 7].

5.1 Structure Graphs

The well known *structured methods* [47, 15], as applied in the data processing industry, use a number of graphical conventions for illustrating aspects of system analysis and design. One of these diagrams, the structure graph, is used to describe the

relationship of the modules (procedures, subroutines) in a program. In the classic “Yourdon and Constantine” structure graph, rectangles represent modules. Arrows connecting the rectangles indicate the caller/callee relationship of the modules. Data flow between the modules (often passed as parameters of the inter-module calls) is shown using arrows with terminated with circles.

The structure graph shows the overall organization of a program. It also assists in outlining the levels of functional abstraction in the program.

If a program is developed using the traditional approach of functional decomposition, the resulting structure graph often has what is termed a “mosque shape”. A single module is at the top of the graph, usually corresponding to the “main program”. As we work down through the graph, successive layers of modules provide increasingly elementary operations. When viewing the overall design, we may choose to ignore the lower levels of graph: each layer provides a level of abstraction which is implemented by the layers below it. The concept of the structure graph has been extended by Buhr [8] and Booch [4] for the Ada language.

Booch’s work focusses on abstract data types, and he suggests that object-oriented design and implementation is possible using Ada. He maps the concepts of classes and instances, as defined by Smalltalk, to Ada packages, generics, and private types.

Booch’s notation for an Ada object is shown in Figure 5.1. Packages that export private types represent object classes. Object instances are provided by instances of these exported types; variables serve as the names of these objects. Procedures exported from the package provide the class operations (methods).

When providing a graphical description of an object-oriented design, Booch does not use all the icons defined in Figure 5.1. Instead, he shows the classes (packages) and the *uses relationships* between them, as shown in Figure 5.2.

This style of diagram serves as an aid when defining the dependencies between code modules. Note, however, that these diagrams do not indicate the names of all of the procedures, variables, and types provided by one module that are accessible by other modules. Details of module interaction, including calling relationships and data flow, are not described.

Figure 5.1: Booch's Notation for Ada Objects

Figure 5.2: Notation for Uses Relationships (Booch)

Buhr's diagrams describe the structural design of concurrent systems based upon Ada's model of process organization and communication. Buhr describes icons for representing Ada's procedures, packages and tasks; [8] provides several examples of iconic representations of concurrent systems. Packages, tasks, and procedures may be nested within one another. Icons can be nested within one another, corresponding to the textual nesting of packages, procedures, and tasks allowed by Ada. Layers of functional abstraction can be shown in this way.

Buhr's diagrams provide a greater level of detail than those suggested by Booch. In particular, the interactions between structural components are indicated. Although the diagrams may be annotated to provide some indication of temporal behaviour, they are intended to provide a static "snapshot" of a system's structure. Data abstraction, and the concepts of inheritance as supported by object-oriented languages, are not explicitly supported, although recent work has addressed these issues [7].

We have found the idea of describing object-oriented programs through pictures to be attractive. Buhr's work illustrates the benefits of an iconic approach to design, but the Ada-based model which his notation represents is overly restrictive, lacking support for many of the notions found in the object-oriented paradigm. Structure graphs as described above are not sufficient for describing programs developed in Objective C or Smalltalk. To extend this "Ada-style" notation to object-oriented languages, a number of issues must be addressed:

- While Ada may support object-oriented design, it is not an object-oriented language [42]. However, the language's package syntax offer a means to encapsulate data and provide abstraction. If we consider a package to be equivalent to a class, then sending a message to an object instance is roughly equivalent to calling a procedure provided by a package. A package's procedures are analogous to instance methods; and instances may be represented by pointers to data structures of a private type. A structure chart for object-oriented systems must provide a means to describe objects and their methods, as well depicting messaging between object. There are sufficient similarities to the icons used to describe Ada constructs to use this notation as a starting point.

- Ada does not support the notion of inheritance, except through the somewhat more limited mechanism of generics. This concept is poorly depicted in Buhr's and Booch's diagrams. For object-oriented languages, a means of depicting the class hierarchies appears desirable.
- The dynamic behaviour of programs are not captured by Booch's diagrams, and only partly by Buhr's. In object-oriented languages, objects are created and destroyed throughout the lifetime of the program. It isn't sufficient to show only the object classes; we need a way to show the dynamic nature of the program as objects are instantiated from the classes, used, and then discarded.
- In Objective C, objects may be sent arguments of any type when a method is invoked, and methods may return items of these types. Any of C's primitive types may be used: ints, chars, or pointers are all acceptable. However, objects themselves may be sent as arguments of another object's method-call, and methods may return objects. For example, if object A returns an instance of class B as a result of a method-call, we can then send messages to that returned object. Objects must be viewed as both data and procedure. In the traditional structure graph, data (variables) are passed between modules. To graphically depict object-oriented systems, we need a way to show an object instance as both "data" (that is, as a method argument or return value) and "code" (that is, an entity which can send and be sent messages).
- Objective C or Smalltalk classes may not be hidden by nesting the program text for one class within another. All classes (including all subclasses of a class) are always visible from all points within an application. This is in contrast to Ada, where structural components may be textually nested and thus hidden within layers in the corresponding design diagram. If we attempt to describe an application as Booch does in Figure 5.2, showing the relationships between all classes on a single diagram, the diagram is likely to become cluttered.

5.2 A Notation for Object-Oriented Program Design

To address the issues raised in the previous section, we propose a number of icons for describing the design of object-oriented programs. This notation is an extension of that described by Buhr. The icons are shown in Figures 5.3 through 5.6; each is briefly described below:

- **Classes** are represented by the dashed rectangle. Small rectangles represent class methods, and are labelled with the message selectors. Superclasses are optionally shown underlying the classes. Inherited methods may be shown in the class in which they are defined, or in the class which inherits them.
- An instance is drawn as a solid rectangle. The symbol is labelled with the instance's name, and its class. Small rectangles adjacent to the edge of the instance icon represent instance methods. Methods that are defined within the instance's class, as well as any inherited methods, may be shown.
- Through inheritance, an object often supports many more methods than are used in a particular application. It is inconvenient to draw all of these methods in the object icon. A method icon containing ellipses may be used to indicate that an object provides other, unspecified methods.
- An arrow is used to indicate the sending of a message. The head of the arrow touches the invoked method in the receiving object.
- In order to indicate an object instance being passed into or out of an object, we use the data flow arrow as described by Buhr, with a superimposed square (suggesting the object icon). The instance name and/or the class name identify the object.

Our experience with these diagrams has lead us to develop a few additional guidelines for their use.

Figure 5.3: Icons Describing Objects

- Attempting to show both the class hierarchies and the instances on a single diagram usually results in a cluttered diagram. It is preferable if one diagram or set of diagrams is used to describe the superclass/subclass hierarchy for the classes used in the application. Other diagrams are used to show the instances, and only those classes to which messages are directly sent.
- Although the source code for a class contains the code for both the class methods (factory methods in Objective C) and instance methods, these methods cannot be invoked arbitrarily. A class method is executed when a message is sent to a class. An instance method is executed when a message is sent to an *instance* of that class. Instances may not execute class methods, and similarly, classes may not execute instance methods. Given this, the class icon should show only class methods, and an instance icon should depict instance methods.
- Objects typically inherit several methods from their superclasses, and it usually undesirable to show all of the methods provided by each object. As such, for each class or instance, we usually show only those methods that are actually used (that is, a method is shown only if another object sends the corresponding message). Methods that are defined in the class are not distinguished from those inherited from the superclasses.
- The most common reason for sending a message to a class is to create a new instance of the class. However, class methods are provided for other purposes. Objective-C's `Object` class provides methods to support interrogation of the inheritance tree, including:
 - `+class` - replies the name of the class;
 - `+superclass` - replies the name of the receiving classes superclass;
 - `+instancesRespondTo:` - replies whether or not instances of this class respond to the message passed as an argument.

Graphically, we can show any of these messages being sent to a class icon, as shown in Figure 5.4. Requesting a new instance of a class is depicted in

Figure 5.5. The new instance is shown as being returned by **new**, and is labelled to correspond with the instance icon. This behaviour is sufficiently common that we use a visual shorthand, as shown in Figure 5.6. A thick arrow pointing to an instance implies that the instance was created by sending a message to its class. In this way, we avoid explicitly showing the class icon.

- In an object-oriented structure graph, we should show the internals of a class separately from the usage of instances of that class. Diagrams showing the application of an object need not show the implementation details of that object. For example, in the program that is described in the next section, we define a class **Node**, which is a subclass of class **ByteArray**. Within a **Node** instance, we use an instance of class **Set**, to store the ids of other objects referenced by that node. Additionally, the node instances are stored in an instance of class **Graph**. Using Booch's notation, we would show the "uses" relationships between the **Graph**, **Node**, and **Set** classes as shown in Figure 5.7. However, the set instance is used within the **Node** instance, therefore, we believe it is misleading to show class **Set** at the "global level".

An alternative is shown in Figure 5.8. In the top level diagram, instances of **Node** and **Graph** classes are shown. The **Set** class does not appear here.

Figure 5.9 shows a portion of the implementation of class **Node**. Since this diagram deals with the internals of **Node**, it is the appropriate place to show how the **Set** class is used.

To summarize, "top level" diagrams show the creation of objects that are considered to be global to an application, and the messaging between them. Separate diagrams are used to depict the internal design of objects; any objects which are used in the implementation of an object (and can thus be considered "private" objects) are shown in these figures.

Figure 5.4: Sending a Message to an Object Class

Figure 5.5: Creating an Object Instance

Figure 5.6: Shorthand Notation for Instance Creation

Figure 5.7: Uses Relationships Between Graph, Node, and Set Classes (Booch)

Figure 5.8: Instances of Graph and Node Classes

Figure 5.9: Implementation of the Node Class Uses the Set Class

5.3 An Example: Cox's Dependency Graph Application

In his book, Cox [11] illustrates his approach to object-oriented design by presenting a “dependency graph” application. Figure 5.10 shows a dependency graph, which, in this case, represents the relationship of the modules in a program. Node “main” is the root of the graph and is dependent on nodes “proc1”, “proc2”, and “proc3”. Node “proc2” is in turn dependent on node “proc4”. “proc5” is dependent on “proc3”.

This application reads in a file describing the graph. The first entry in each line defines a node; the remaining entries are the nodes it references:

```
main proc1 proc2 proc3
proc2 proc4
proc3
proc5 proc3
proc4
```

When the program is run, it builds the graph, and marks all nodes that are reachable from the root (main). Here, nodes proc2, proc3, and proc4 will be found to be reachable. proc5 is unreachable from main. proc1 is listed as being undefined, since no line in the input file begins with that node.

The steps Cox follows to design this application are now briefly described. First, the required objects are identified from the description of the problem space. Two new classes are identified in this way: the graph, and the nodes contained within the graph.

Next, the application is developed by iterating between what Cox terms the “consumer’s view” of the objects and the “supplier’s view”. The consumers’ view is essentially the specification for a class: defining the methods that are required and showing how they will be used.

The supplier’s view corresponds to the implementation details of the objects. First, the supplier (that is, the software engineer) must decide where each object fits

Figure 5.10: A Dependency Graph

in the class hierarchy. This, of course, determines what methods an object inherits from its superclasses. This step requires that the developer have a good understanding of the existing class library. Extensive experience in developing object-oriented applications (or at least, a knowledgeable colleague) is of great importance here. In the dependency graph application, class `Graph` is defined as a subclass of `Set`, while class `Node`'s superclass is `ByteArray`.

Then, the code for the new methods is developed. The developer may choose to create instances of other classes in order to provide the required functionality. For example, each node instance contains an instance of class `Set`, to contain the list of nodes which it references.

Cox's book presents the code for the `Graph` and `Node` classes, as well as the "driver" routine which reads the input file, builds the graph, and determines which nodes are reachable from the root node.

Before presenting the code, a few comments are in order. First, the application is much shorter than the equivalent program written in C. A substantial amount of the program's functionality is obtained through inheritance, and from exploiting the existing class library.

However, our initial reaction when reviewing this code is that it is quite fragmented. We have a number of objects, sending messages and returning new object instances between themselves. The overall control flow is not as apparent as with a program developed in the traditional top-down, procedural paradigm. In order to understand the code, we must be aware of what functionality the `Node` and `Graph` classes obtain from other objects in the class library.

Additionally, the way that Cox describes the design process, the application seems to have "evolved", until it reached a point where he felt it was complete. The design of the application finished when the coding was completed.

Our concern is, how can you present an overview of the design of this application? That is, how can we understand this application without working our way through the code? As with any software project, we may have to present the design for review prior to starting the implementation. Also, we need some way of documenting the design for maintenance purposes. If the code can't be understood, there will be a

great temptation to throw it out and start from scratch. This is a major problem with current software projects, which the object-oriented paradigm is intended to correct!

Cox discusses the notion of a “software blueprint” to present a design, but the format of his blueprints do not provide sufficient detail to describe the implementation. He provides a unique icon for each class, the shape of which suggests the nature of the class. However, these icons don’t indicate the methods that the classes provide or the messaging between objects.

Here is the code for the dependency graph, reprinted (with corrections) from Cox’s book:

Graph.m

```
// Dependency Graph
#include "objc.h"
extern id Node;

= Graph:Set { }

// Add a new node
- addNode:(STR)aCharPointer
  { return [self filter:[Node str:aCharPointer]]; }

// Add a new node and define it
- defineNode:(STR)aCharPointer
  { return [[self addNode:aCharPointer] define]; }

//Mark the given node (and all nodes referenced by it)
- mark:(STR)aNodeName
  { return [[self addNode:aNodeName] mark]; }
=:
```

Node.m

```
// Node in graph of named nodes
#include "objc.h"
```

```

= Node:ByteArray
  { BOOL isDefined, isMarked; id references; }

//This node references anotherNode.
- addReference:anotherNode { extern id Set;
  if (references == nil) references = [Set new];
  [references add:anotherNode];
  return self;
}

// Mark myself and my dependents as reachable.
- mark { if (isMarked == YES) return nil;
  isMarked = YES;
  [references eachElementPerform:@selector(mark)];
  return self;
}

//Mark myself as defined
- define { isDefined = YES; return self; }

//Reply whether I'm reachable or defined
- (BOOL)isMarked { return isMarked }
- (BOOL)isDefined { return isDefined }
=:

mGraph.m ( Driver routine - called by main() )

#include "stdio.h"
#include "objc.h"
  id graph, lastNode;
  extern id Graph, Node;
  extern STR root;
  char yytext[BUFSIZ]; //lexical buffer

doGraph() { id sequence, node;

  // Build the graph by reading the input file
  graph = [Graph new];
  yyparse();

```

```

// Mark all nodes reachable fromn the root
[graph mark:root];

// Print reachable nodes
sequence = [graph eachElement];
printf("Reachable from %s: ", root);
while (node = [sequence next])
    if ([node isMarked]) printf("%s", [node str]);
printf("\n");

// Print unreachable nodes
[sequence toFirst];
printf("Not reachable from %s: ", root);
while (node = [sequence next])
    if (![node isMarked]) printf("%s", [node str]);
printf("\n");

// Print undefined nodes
[sequence toFirst];
printf("Undefined nodes: ");
while (node = [sequence next])
    if (![node isDefined]) printf("%s", [node str]);
printf("\n");

[sequence free];
}

#define DefAction(yytext)\
    lastNode = [graph defineNode:yytext]
#define RefAction(yytext)\
    [lastNode addReference:[graph addNode:yytext]]
#include "Syntax.c"

```

Syntax.c contains code that parses tokens from the input file and invokes one of the macros, DefAction or RefAction, according to whether the input token was the first token on a line. The logic that builds the graph is hidden inside the parsing function, yyparse().

Syntax.c

```

#define WRD 1
#define EOL 2

/* Handcoded parser */
yyvsparse() { register int c;
  for(;;){
    /* for each line in the file.... */
    switch(c = yylex()) {
      case WRD: DefAction(yytext); /* define a node */
        for(;;){ /* for each reference in the line... */
          switch(c = yylex()) {
            case WRD: RefAction(yytext); /* ref a node */
              continue;
            case EOL: break;
            default: yyerror("syntax error 2"); continue;
          } break;
        } break;
      case 0: return 0;
      case EOL: continue;
      default: yyerror("syntax error 1");
    }
  }
}

#include "ctype.h"
/* Lexer */
yylex() { register int c;
  while ((c = getc(stdin)) != EOF) {
    if (c == '\n') return EOL;
    else if (isspace(c)) {
      do { c = getc(stdin); } while (isspace(c));
      ungetc(c, stdin); continue;
    } else { register char *p = yytext;
      do { *p++ = c; c = getc(stdin); }
      while (!isspace(c));
      ungetc(c, stdin); *p = 0; return WRD;
    }
  }
  return 0;
}

```

This is a small program; however, although only two small classes have been

defined, the overall design is not readily apparent. Consider the line:

```
lastNode = [graph defineNode:yytext]
```

This sends the `defineNode:` message to the `graph` instance, causing the node whose name is in the string “yytext” to be defined. The new node is returned and stored in `lastNode`.

Now, look at the code for the `defineNode` method:

```
{ return [[self addNode:aCharPointer] define]; }
```

Here, the nested method-call `[self addNode:aCharPointer]` returns an object which in turn is sent the `define` message; the result of that method-call is then returned to whatever object sent the `defineNode` message. But exactly what is returned by the two methods? To understand this, we have to know that `addNode:` is another method defined in `Graph`, which creates the new node instance, adds it to the set of nodes maintained by `Graph`, and returns the new node. That new node is sent the `define` message. The `define` method is implemented in `Node`, and it returns “self” (that is, the node to which the message was sent).

To understand a method, we have to comprehend both how the method is used as well as its implementation, which requires us, in turn, to understand any other methods that are referenced.

The `addNode:` method in `Graph` provides another example:

```
{ return [self filter:[Node str:aCharPointer]]; }
```

Method `str:` is inherited by the `Node` class from `ByteArray`; it returns a new instance of `Node`. The `filter:` method is inherited by `Graph` from its superclass `Set`, and its argument is the object (the new node instance) that is to be added to the set. `filter:` is sent to `self` (an instance of class `Graph`) but it returns its argument (the new instance of `Node`). We are expected to understand the `ByteArray` and `Set` classes when reading this line of code.

Given the amount of information implied by a single line of code, we believe a graphic representation is especially important. The following diagrams are an attempt to capture the design of the dependency graph application.

Figure 5.11 shows all of the classes used in the application, along with their superclasses. Figure 5.12 shows the internals of the **Node** class. Figure 5.13 describes the **Graph** class. Numbers annotate the message send arrows, indicating the order in which messages are sent. Given these, Figure 5.14 and Figure 5.15 describe the code for portions of **doGraph()**, **yyparse()**, and **yylex()**, inside file **mGraph.m**. Figure 5.14 shows that code within **doGraph()** creates an instance of the **Graph** class, calls **yyparse()**, and sends messages to the **graph** instance. Figure 5.15 illustrates the behaviour of function **yyparse**, as it builds the dependency graph. It sends messages both to the **graph** instance, and to the node instances returned by the **graph**.

As discussed earlier, an object-oriented application is not easily represented by a single “snapshot”. Objects are being created and destroyed dynamically while the program runs. It seems best that a series of structure graphs be used, each representing a portion of the temporal activity of the program.

Figure 5.11: Classes Used in the Dependency Graph Application

Figure 5.12: Implementation of the Node Class

Figure 5.13: Implementation of the Graph Class

Figure 5.14: `doGraph()`: Use of the Graph Instance

Figure 5.15: `yyparse()`: Building the Dependency Graph

Chapter 6

A Model for Object-Oriented Real-Time Programs

This chapter develops a model for the design and implementation of real-time systems using the object-oriented programming paradigm. Specifically, it explores how Unison applications may be developed using the Objective C language.

We examine how objects may be used in the multitasking environment supported by Unison, and we explore several approaches to defining multitasking objects. Several versions of a bounded buffer task are developed to illustrate features of our model.

6.1 Primitive Data Types and Objects

Unison tasks are coded as C functions, and may use the primitive data types supplied by that language: chars, ints, floats, pointers, and so on. Complex data structures, such as priority queues or trees, are formed out of the elementary types supported by the language. Data abstraction and encapsulation are used as the programmer sees fit, and, to a limited extent, provide a means to implement “generic” abstract data types.

In Smalltalk, essentially all entities are objects: integers, stacks, queues, pens, windows, and so on. As a result, an instance of an array can hold any other object.

In fact, the array could hold a heterogeneous collection of objects; for example, one array instance might contain both integers and real numbers.

In some cases, a specialized subclass of an array is required. For example, a `CharacterArray` class provides us with character strings. There are several operations for character strings that would not be appropriate for arrays of objects other than characters, so a specialized character class may be appropriate for a particular application. Despite this, the ability of an “array of objects” class to handle all entities supported by the language is a powerful concept.

Objective C is a hybrid language. The Objective C compiler translates the object syntax into structures comprised of C’s primitive types. The base types provided by C are not treated as objects. Hence, in Objective C, we have two “groups” of entities. On the one hand we have instances and classes, all of which are descendants of the `Object` class. On the other hand, the programmer still has to deal with all of C’s primitive types.

This affects the number of classes supplied with the language and required by a typical application. For example, consider the `Array` abstract class supplied with the language. The `IdArray` subclass derived from `Array` holds objects. (Actually, it holds the pointers to the objects’ instance variables, but the abstraction of it holding the objects themselves is useful.) We also need an array subclass for each primitive type. `ByteArray` holds characters, and provides “string objects”. `IntArray` (an array of integers) and `FloatArray` (an array of floats) are both required, even though the two classes provide the same methods.

Presumably, we could convert integers into “integer objects”, and then store them in an `IdArray`, rather than store integers in `IntArrays`; however, there are inefficiencies associated with this approach.

As we shall see later, there are advantages to trying to develop code where all entities are objects. However, this must be weighed against the overhead of attempting to convert primitive types into objects.

6.2 Considering Tasks as Objects

For this thesis, we understand “object-oriented” to mean an environment which allows us to exploit the concepts of data abstraction, encapsulation/information hiding, dynamic binding, and inheritance.

In a system consisting of a number of concurrent tasks, it is reasonable to expect that object-oriented techniques may be used to encapsulate the data manipulated within each task. The same advantages and disadvantages of object-oriented techniques apply as when developing a program with no concurrent elements. This is not surprising, considering that there is a single thread of control within a task (although this flow may be blocked for a period while the task communicates with other tasks).

Objects supported by Objective C are not concurrent. They execute only upon receiving a message. An object which has sent a message remains blocked until the method invoked in the receiving object finishes execution.

In this chapter, we consider the development of concurrent objects, that is, we examine approaches for aligning the concepts of tasks and objects. The notion of communicating sequential processes appears to have a number of similarities to the object-oriented world, so it is worth asking: “are tasks the same as objects?” In other words, should we attempt to merge Unison tasks and Objective C objects? At first glance, the syntactic uniformity suggested by this is appealing. In a multitasking, multiprocessor system, concurrent objects would be addressed in the same manner as passive, nonconcurrent objects.

Objects represent autonomous units which perform work in response to messages from other objects, and may in turn send messages to other objects to request work. The similarity to tasks communicating by message passing is obvious¹. Consider the characteristics of object-oriented systems, and the extent to which tasks demonstrate these characteristics, as listed below:

- *Encapsulation -*

¹We note that another approach to comparing multitasking and object-oriented systems is presented in [28]. This evaluates the Harmony operating system [16] as an object-oriented system, using the classification scheme suggested by Wegner [42]

Objects provide for the encapsulation of data. As such, objects are similar to tasks which share no common data with other tasks. Messages are used to pass data into and out of a task, and inform the task of the operations to be performed on the data. This is similar to the way in which objects communicate. Objects and tasks appear to support the concept of encapsulation equally well.

- *Abstraction* –

From the structural viewpoint, models for common tasks have been defined, with names suggestive of their human counterparts: proprietor, transporter, agent, middleman, and so on [8, 17]. These are similar to the passive abstract data types supported by Objective C (for example, queues, stacks, and, sets) except that the task abstractions are concurrent. While object-oriented programming provides for data abstraction by encapsulating the data and operations in an object, it is possible to have “abstract tasks” without resorting to implementing the tasks using an object-oriented language.

- *Polymorphism* –

With polymorphism, each object is responsible for handling all messages sent to it in a manner appropriate for that object. Tasks may be designed to be polymorphic, in that each may be coded to handle all possible messages exchanged between tasks in a concurrent system. Different tasks would respond to the same message in different ways, depending upon the role of the task in a system. Polymorphic behaviour by tasks can implemented without using an object-oriented language. Also, we note that for practical purposes, it is unlikely that a system design would require different tasks to handle all types of inter-task communication.

- *Inheritance* –

Currently, if we want a task that is similar (but not identical) to another task, we have to obtain and edit the source code for the task. When a section of code that is common to several tasks is changed in one task, we have the problem

of locating all the other tasks where that code was used, and duplicating the changes as required.

As supported by object-oriented languages, inheritance allows for the creation of objects that are specializations of other objects. Code that is common to several classes resides in the methods of the superclass for those objects.

The Unison tasking model allows for the creation of generic server tasks. When a task is created, it may be passed a message containing any initial information required by the task; for example: buffer sizes or addresses of peripheral devices. This allows a generic server to configure itself before it begins operation. However, we note that this mechanism for run-time initialization of generic tasks is not equivalent to the inheritance concept supported by object-oriented languages.

To summarize: of the characteristics/benefits of object oriented programming, three (encapsulation, abstraction, and polymorphism) are directly supported by Unison's tasking model without requiring an object-oriented implementation language. Inheritance allows for increased code reusability and maintenance, but requires a language designed to support the concept.

6.3 Implementing Tasks as Objects

We have shown that, from the viewpoint of design, tasks may be considered as concurrent objects. In this section, we explore how these tasks might be defined in an object-oriented language which directly supported multitasking, and discuss whether this approach is feasible using Objective C and Unison.

To support the notion of concurrent objects, we first define a new abstract class: process. Factory methods would include "new", to create a new instance (process) and "free", to kill the process and dispose of the object.

```
=process:object { ..instance variables }

+(id)new { ... }
+free {    }
```

■.

Designers would create subclasses of the process class, one for each type of process required in a system. When the application started, an initial root task would create instances of the process objects, which would then start executing in parallel.

Note that many of the process subclasses would only have one instance (that is, one copy of the task would be present in the system). Task pools as defined in [8] would correspond to multiple instances of a process subclass.

Consider a simple real-time control system, as shown in Figure 6.1. A timer task handles interrupts generated by a real-time clock. This task provides an abstraction, encapsulating the details of the clock hardware. Periodically, it sends a message to a supervisor task, indicating that a unit of time has passed. At the appropriate time, the supervisor task will reply to messages sent by one or more controller tasks, informing each to obtain and process a sample from an analog-to-digital converter.

Implemented using our object-oriented concurrent pseudo-language, each type of task would correspond to a different process subclass. Each process subclass would define methods corresponding to the type of messages it can handle. For example, consider a timer class.

```
=timer:process { ... }

-rtcTick { ..update the internal clock, and send to
            the supervisor when an appropriate number
            of interrupts have occurred: }
    [supervisor timeHasElapsed]

}

-setrate:newrate {
    initialize the timer task, telling it the time
    interval it should monitor,
}

-init // the initial thread of control when a
```

Figure 6.1: Structure Graph: Simple Control System

```

        // process is created.
    { while TRUE
        { initial statements;
          [self await_messages];
          // block waiting for any message.
          // return here when the method corresponding
          // to a received message has been executed.
          other_processing;}
    }

```

Tasks would send messages to another task using the object's message sending syntax: the message would specify the name of the method that the receiving task should execute. (The expression `[self await_messages]` indicates that the task is ready to receive messages; instead of executing an "await_messages" method and immediately returning, the task would block, awaiting a message from any other task.)

This approach is not directly implementable using Objective C, as it ignores several aspects of object-oriented languages. In particular, the "object model" supported by Objective C's objects and Unison's tasks differ in a number of areas:

- Objects are passive. They do not start to compute until a message has been sent to them. Tasks compute in parallel, synchronizing when they exchange messages.
- Sending a message to an object is analogous to sending a message from one task to another. However, in Objective C a "receiving" object does not perform an explicit receive operation, unlike a receiving Unison task.
- When an object's method is invoked, it is not informed of the identity of the object which sent it the message. The sender is normally anonymous; if the recipient requires this information the identity of the sender must be passed as an argument of the message. In Unison, the message passing primitives always return the task id of the correspondent task.
- In Objective C, messaging is synchronous. The sender waits while the receiver computes and resumes execution only after the receiver's invoked method re-

turns. The receiver has no way of “releasing” a sender until it returns a value. At that point, the receiver is “exit’ed” (in the same manner as a return from a function call). This behaviour is similar to a Unison task which replies to a sending task only after all computation resulting from a received message is complete, and then immediately performs another *_receive*. Such a task does not exploit any concurrency within the system.

In an attempt to overcome some of these restrictions, we consider another approach to creating “task objects”. We want to be able to address Unison tasks as Objective C objects, and we want to exploit the language’s inheritance mechanism when implementing these objects.

In Figure 6.2 we show a task creating another task indirectly through a process class instance. The steps are as follows: First, a task (the parent) creates an instance of a process class. When the instance is initializing, it calls Unison’s *_create()* function to create the new task (the child). Execution of the new task would start with the root function passed to *_create()*. When initialization of the process instance has finished, it returns its id to the parent task. (Note that this is not the task id returned by *_create()*, but rather the object id as returned by the Objective C runtime support routines.)

There are a number of problems with this approach of “hiding” a Unison task behind a process instance. First off, the process instance merely serves as a means of mapping an object id to a Unison task id. The code for the task does not reside in the process class, so inheritance can’t be used in its implementation.

The parent task would communicate with the child task by sending a message to the corresponding process instance, which would in turn relay the message to the task. However, the newly created child task cannot address its parent as an object unless it knows the object id of its parent’s process instance. Under Unison, a child task can obtain the task id of its parent task via kernel primitive, and the ids of correspondent tasks are returned when the inter-task messaging primitives are called. For the scheme we propose here, we would need a means by which tasks would learn of the object ids of the other tasks with which they were communicating. Integrating this transparently would appear to necessitate merging the Unison kernel

Figure 6.2: Creating Tasks Through Objects

and Objective C run-time system.

As an alternative approach, we consider instantiating all objects within tasks, as shown in Figure 6.3.

The steps are as follows:

1. Use the Unison `_create()` primitive to create a new child task. A task id is returned to the parent, and the new task starts executing in parallel.
2. The root function of the new task creates a new process instance by sending a “new” message to the appropriate process class.
3. When the id of the new process instance is returned, an ‘init’ message is sent to that instance. This method contains the body of the task and normally would never return.

To summarize this approach: at this point we would have created a new task context, and within that context have created an instance of an object (a “process instance”) that contains the bulk of the code associated with the task. Control is passed to the object by executing an ‘init’ method. Once in this method, other methods are invoked by sending the appropriate message to ‘self’. Note that if a particular method isn’t an instance method of the process class, Objective C’s message processing routine will look for it in the superclass. As such, we now can exploit inheritance.

Also note that tasks would communicate by sending and receiving to task ids as defined by Unison, and not to object ids. A uniform syntax for message passing, both between tasks and between objects within tasks would be desirable but does not seem possible with this approach. However, this approach has the advantage that it should not require the language run-time support routines to be integrated with the operating system kernel.

6.4 An Example: Notifier Tasks

The merits of the approach presented in the previous section are best discussed by means of an example.

Figure 6.3: Creating Process Instances Within Tasks

Consider the case where a task initiates a hardware operation and then waits for a hardware interrupt to indicate the completion of processing. Under Unison, a task that waits to be signalled by an interrupt service routine normally runs at the same priority as the interrupt.

Since we may not wish the task to run at this priority, we can use a notifier task, which alternates between waiting for the interrupt and sending a message to the its parent task, to inform it if the interrupt. This is illustrated in Figure 6.4.

As shown in the following code, the notifier task sends to the main task to indicate that it is ready to work, and blocks (1). The main task replies to the notifier, just prior to programming the hardware (2). The main task then programs the hardware. Eventually, it attempts to receive a message from the notifier (3). Meanwhile, the notifier task is no longer send blocked, and waits for the hardware interrupt (4). When it occurs, the notifier sends to the main task (1) informing it that a hardware interrupt was received.

```
Notifier() {

    father = _father_id();
    for ( ; ; ) {

        request.MSG_SIZE = sizeof( request );          (1)
        reply.MSG_SIZE = sizeof( reply );
        _send( (char *)&request, (char *)&reply, father, port ... );
        /* Await interrrupt */
        _receive( (char *)&request, ... , signal_mask, ...); (4)
    }
}

main() {

    notifier_id = _create( NOTIFIER );
    ...
    /* wait for initial message from notifier */
    request.MSG_SIZE = sizeof( request );
    requestor = _receive( (char *)&request, port_mask, ... );
    ...
}
```

Figure 6.4: Structure Graph: Use of Notifier Tasks

```

    /* release the notifier */
    reply.MSG_SIZE = sizeof( reply );    (2)
    reply.RESULT = 0;
    _reply( (char *)&reply, requestor );
    ...
    /* initiate hardware operation */
    /* perform other operations, then await the notifier */
    request.MSG_SIZE = sizeof( request ); (3)
    requestor = _receive( (char *)&request, port_mask, ... );
    /* hardware interrupt was received */
    ...
}

```

A task similar to the interrupt notifier might be used in a real-time clock handler. A RTC_notifier task waits for interrupts from timer hardware, and after an appropriate number of interrupts have occurred, sends a TICK message to its parent task.

The skeleton code for the RTC_notifier is:

```

RTC_notifier() {

    parent = _father_id();
    request.MSG_SIZE = sizeof( request );
    request.TIMER_REQUEST_TYPE = TICK;
    reply.MSG_SIZE = sizeof( reply );

    for( ; ; ) {

        /* wait for the interrupt */
        _receive( (char *)&request, ..., RI_CLOCK_MASK, ... );

        /* if enough ticks have occurred... */
        _send( (char *)&request, (char *)&reply, parent, ... );
    }
}

```

The functions of the two notifiers are similar: both send notification of an interrupt to tasks which cannot block waiting for the interrupts. In the first case, the

interrupt is a direct result of the actions of the main task (that is, programming the hardware). In the second case, the clock interrupts occur independently of the actions of the parent task. (This is the reason that the order of the `_send()` and `_receive()` calls are in the reverse order in the two notifiers. The `RTC_notifier` could `_send()` first, if the correspondent task performed an initial “dummy” receive/reply to handle the “false” notification.)

As discussed in the previous section, we wish to encapsulate the bulk of a task’s code in an object in order to exploit inheritance. For this, we can define a notifier class:

```

_notifier_obj:... { int_num } // the signal mask for the
                               // interrupt is an instance
                               // variable

-run {
    father = _father_id();
    for ( ; ; ) {

        request.MSG_SIZE = sizeof( request );
        reply.MSG_SIZE = sizeof( reply );
        _send( (char *)&request, (char *)&reply, father, ... );
        _receive( ..., int_num, ...);
    }
}

-init:signal {
    int_num = signal;
    [self run]
}

=.
```

The notifier task becomes:

```

_notifier() {

    id = [_notifier_obj new]
    [id init:INT_NUM]
}
```

When the task is created, it creates an instance of the notifier class and sends it the init message, informing it of the signal on which it should wait for interrupts. This method then sends itself the run message. The corresponding method is an infinite loop encapsulating the algorithm for the task.

The notifier in the first example can use an instance of this task class directly. The RTC_notifier is a specialization of this class, in that it sends a specific TICK message to its parent task.

```
=RTC_notifier_obj:notifier_obj

-init:signal {

    request.MSG = TICK;
    [super init:signal]
}
=.
```

The RTC task is:

```
RT_Notifier() {

    id = [RT_notifier_obj new]
    [id init:RT_CLOCK]
}
```

Another matter must be considered. Notice that the general notifier task sends a message to its parent task after each hardware interrupt is received. The RTC_Notifier may receive several hardware interrupts before notifying its parent. As such, the subclass which implements this task would have to override the body of the run method, in addition to the init method.

Although both of these tasks are notifiers, a close examination shows that they differ significantly in their operation. This leads us to believe that there may be difficulties in using classes to inherit task behaviour.

In object-oriented programming, the methods defined for an object's class are more specialized than those provided by an object's superclass. Now, consider the

various task abstractions (administrator, transporter/courier, and so on [17]) used in concurrent system design. The general form of a task structure describes the control flow common to all tasks following that abstraction. Given a template for the general form of a task structure, we would like to add specialized routines, but start execution in the less specialized routines, calling the specialized routines as required.

As an example, the form of the proprietor task is:

```
Proprietor() {
    initialize();
    for( ; ; ) {
        requestor = _receive( (char *)&request, port,... );
        critical_service( request, requestor );
        _reply( (char *)&reply, requestor );
        non_critical_service( request, requestor );
    }
}
```

Given a template of this form, we would like to encapsulate general algorithm in a proprietor process class, with the specialized critical and non-critical service routines defined in a subclass. That is, the default methods for the proprietor services would be overridden in the subclass that inherits from the proprietor. The infinite loop, containing the general proprietor algorithm, would be contained in the proprietor class. The service routines specific to an application would be defined in a subclass. The general form of a proprietor class would be:

```
=Proprietor:... {...}

-init {
    initialize();  /* common to all proprietors
}

-run {           // proprietor algorithm
    for( ; ; ) {
        _receive(...);
        [self critical_service:request for:requestor];
```

```

        _reply(...);
        [self non_critical_service:request for:requestor];
    }
}
=.
```

The outline of a specialized proprietor, which inherits the general proprietor algorithm from the proprietor class, is:

```

=...:Proprietor {...}

-init {
    [super init] // code body in the superclass
    // specialized initialization code here
    initialize();
    [self run]
}

// specialized proprietor methods
-critical_service:request for:requestor{ ... }
-non_critical_service:request for:requestor{ ... }
=.
```

To summarize, we have attempted to define a “proprietor class” from which a hierarchy of proprietor objects could be derived. One of these objects would be instantiated inside each Uniscn proprietor task and contain the code common to all proprietor tasks. Inheritance would be used to share code describing similar behaviour among these tasks. Given that we do not want to change the underlying operating system (that is, we don’t want to develop an object-oriented Unison), we do not believe it is reasonable to implement the tasks themselves as Objective C objects. Instead, inter-task messages would use the system primitives provided by Unison, addressing tasks using the task ids managed by the operating system.

Should we attempt to generalize the proprietor task, and define a process class? In this case, the general proprietor and notifier classes would be subclasses of the process class. These classes would in turn have further specialized subclasses, as discussed above.

To evaluate this, consider the following points: First, what operations are common to all tasks? These are the things that would belong in a process class. One

might suggest that inter-task communications could be handled there. (As we shall see later, the send, receive and reply operations are best associated with entities that we call “message objects”. Similarly, interrupt signals can be handled within these objects). The key point is that inter-task messaging is common to all tasks, and as such this functionality should be provided by routines outside of the proprietor class.

Consider again the general form of a proprietor task:

```
Proprietor() {

    initialize();
    for( ; ; ) {

        requestor = _receive( &request, port,... );
        critical_service( request, requestor );
        _reply( reply, requestor );
        non_critical_service( request, requestor );
    }
}
```

We have shown that in an object-oriented implementation, the initialize, critical_service, and non_critical_service functions are tied to the functionality of a specific proprietor, so developing a general form of these functions which could be inherited is not reasonable. Methods for these routines would be provided in subclasses of the proprietor class. Inter-task messaging is common to all tasks, and so it should not be implemented in a proprietor class. A proprietor class would then simply contain the skeleton of the proprietor algorithm:

```
=Proprietor:... {...}

-init {
    initialize(); // common to all proprietors.
}           // subclass provides specialized
           // initializatic code.

-run {           // proprietor algorithm
    for( ; ; ) {
        [self receive];
```



```

        [self critical_service:request for:requestor];
        [self reply];
        [self non_critical_service:request for:requestor];
    }
}
=.
```

This leads us to believe that it is overkill to attempt to define a general proprietor class!

Earlier, we showed that Unison tasks share many of the characteristics of objects, except that they are developed without using inheritance. Let's look at this more closely. In the object-oriented world, inheritance and subclassing allows us to build objects that are increasingly specialized, yet benefit from the functionality of their superclasses. Methods tend to be short, and often send messages to other objects or themselves. A key aspect is that methods change the state of the object by affecting its instance variables.

In our attempts to build objects which capture the general form of particular task abstractions, we have used inheritance to concatenate discrete sequences of code into an overall algorithm. This approach is awkward. Inheritance seems to work best when it is used to inherit algorithms which manipulate a data structure. Using inheritance as a means to share general control algorithms does not seem promising.

To summarize, we feel that while tasks are a natural result of the object-oriented design of systems, they should not be implemented as part of a hierarchy of process classes. Objects are best exploited when designing the internals of a task. As we shall see in the remainder of this chapter, this includes using objects to encapsulate data structures and the Unison message passing mechanism.

Additionally, we feel that it is important to use the existing models of task structure (for example, the administrator, or transporter) during system design. We can exploit the similarity of these anthropomorphic process structures and objects. A programmer who is comfortable with these structures should be able to adapt to the object-oriented paradigm, since he/she has experience in designing a system in terms of autonomous entities.

6.5 Example: The Bounded Buffer Task

The bounded buffer problem is a well known example in the study of concurrent programming. An object-oriented design of this concurrent system was introduced in Chapter 4; the structure graph is repeated in Figure 6.5. From the description of the problem, the consumer, producer and buffer objects are easily identified as the concurrent entities. Following the conclusions drawn in the previous section, these objects would be implemented as Unison tasks. The inter-process communication mechanisms provided by Unison are sufficient to implement the protocol between the tasks; additional active objects (tasks) are not required to realize this particular implementation.

As shown, the producer and consumer tasks send messages to the buffer task to store and retrieve data, respectively. The buffer uses separate ports to receive messages from the producer and consumer. By masking the appropriate port, the buffer task can choose to ignore messages from the producer when the buffer is full; similarly, the buffer task elects to receive messages from the consumer only when the buffer is not empty.

Having completed the design of the tasking portion of this system, we now turn to the internal design of the concurrent objects, concentrating on the bounded buffer. In this section, we develop several versions of the bounded buffer task. Our first attempt sketches a solution using the C language. We then illustrate several refinements of this solution, using Objective C to explore object-oriented implementations of the task.

6.5.1 First Attempt (Conventional)

The skeleton of a bounded buffer task, coded in C, is shown below. After initializing the buffer data structure, the task enters an infinite loop, processing *PUT* requests from the producer and *GET* requests from the consumer ².

²In this example, there is no need for explicit *PUT* and *GET* requests to be encoded in the messages sent to the buffer task. The message type can be deduced from the port through which a message is received. However, this example illustrates that different types of messages can be

2 of /de 2

MicroD

Figure 6.5: Structure Graph: Bounded Buffer, without Objects

The code for the task is:

```

#define SIZE 10                /* buffer size */
typedef ITEM ..                /* type of item to be buffered */

Bounded_buffer(){

unsigned in_index, out_index;  /* pointers into buffer array */
unsigned count;                /* count of items in buffer */
ITEM buffer[SIZE];

    unsigned sender;           /* task ids */
    struct ... request, reply; /* msg structures */

    /* initialization - set up indices into the ring buffer */
    /* Initially, the buffer is empty */

    in_index = out_index = 0;
    count = 0;

    /* initially, accept only requests from the producer */

    port_mask = PRODUCER_MASK;

    /* loop forever, processing requests */

    for( ; ; ){
        request.MSG_SIZE = sizeof( request );
        reply.MSG_SIZE = sizeof( reply );
        /* wait for a message */
        sender = _receive( ( char * )&request, port_mask,
                           NOSIGS, &port, FOREVER );

        switch( request.MSG_TYPE ){
            /* gotcha! what type of request? */

            case PUT:
                /* from the producer */

```

sent through a single port; in this case, the communicating tasks must agree on the format used to encode the type indicator into the message.

```

        _reply( (char *)&reply, sender );
        buffer[ in_index ] = request.ELEMENT;
        in_index = ( in_index + 1 ) % SIZE;

        count++;                /* up contents count */
        if( count == SIZE )
            /* buffer full, accept only GET requests */
            port_mask = CONSUMER_MASK;
        else port_mask = PORT_MASK | CONSUMER_MASK;
            /* accept from producer or consumer */
        break;

    case GET:
        /* from the consumer */
        reply.RESULT = buffer[ out_index ];
        _reply( ( char * )&reply, sender );
        out_index = ( out_index + 1 ) % SIZE;

        count--;                /* decrement contents count */
        if( count == 0 )
            /* buffer empty, accept only PUT requests */
            port_mask = PRODUCER_MASK;
        else port_mask = PRODUCER_MASK | CONSUMER_MASK;
            /* accept requests from either */
        break;

    }
}
}

```

6.5.2 Second Attempt (Buffer as an Object)

Now we can start rewriting the task to make use of objects. The obvious choice is to encapsulate the buffer in an object.

Notice that in the previous version of the task, knowledge of the structure of the buffer is scattered throughout the code (that is, the variable definitions at the top, the initialization statements, and the buffer manipulation in the PUT and GET sections of the switch statement). If we wanted to change the structure of the buffer

from an array to a linked list, we would have to carefully edit the code, ensuring that all references to the buffer were altered correctly.

Using the object-oriented approach, we define a new class, `BoundedBuffer`, which contains the bounded buffer data structure and stores elements of type `ITEM`. Instance methods include:

```
-add:(ITEM)anElement    to add an element to the buffer;

-(ITEM)remove           to return the next element;

-(unsigned)size         to return the count of the number
                        of elements in the buffer;
```

As a consequence, there are a number of changes to the buffer task. An instance of the `BoundedBuffer` class, `aBuffer`, is used to store the items received from the producer task. Rather than manipulate the buffer data structure directly, the buffer task sends messages to `aBuffer` to store and retrieve data.

The structure diagram for this version is shown in Figure 6.6. The use of the `Bounded Buffer Class` is illustrated. Here is the code:

```
/* bounded buffer - using the BoundedBuffer class */

extern id BoundedBuffer          /* factory object for the buffer */

#define SIZE 10                  /* buffer size */
typedef ITEM ...                 /* type of item to be buffered */

Bounded_buffer(){

    unsigned sender
    struct ... request, reply;
    id aBuffer;

    /* initialization */

    aBuffer = [BoundedBuffer new:SIZE]; /* create the buffer */
```

Figure 6.6: Bounded Buffer, using the Buffer Class


```

/* loop forever, processing requests */

port_mask = PRODUCER_MASK    /* initially, accept requests
                               to store data */

for( ; ; ){
    request.MSG_SIZE = sizeof( request );
    sender = _receive( ( char * )&request, port_mask,
                      NOSIGS, &port,FOREVER );
    switch( request.MSG_TYPE ){
        case PUT:                /* from the producer */
            _reply( (char *)&reply, sender );
            [aBuffer add:request.ELEMENT];
            if( [aBuffer size] == SIZE )
                /* buffer full, accept only GET requests */
                port_mask = CONSUMER_MASK;
            else port_mask = PRODUCER_MASK | CONSUMER_MASK;
            /* accept from producer or consumer */
            break;

        case GET:                /* from the consumer */
            reply.RESULT = [aBuffer remove];
            _reply( ( char * )&reply, sender );
            if( [aBuffer size] == 0 )
                /* buffer empty, accept only PUT requests */
                port_mask = PRODUCER_MASK;
            else port_mask = CONSUMER_MASK | PRODUCER_MASK;
            /* accept requests from either */
            break;
    }
}
}

```

The implementation of the BoundedBuffer class is not important to this discussion. The important aspect of this class is that it maintains a fixed sized buffer capable of storing items of data ³. In this example, the type of the buffered elements has been typedef'ed to 'ITEM'. Internally, the BoundedBuffer class might maintain

³In practice, a Buffer class would likely be implemented as a subclass of the Queue or Collection classes provided with Objective C. However, as instances of these classes become full, they attempt

a queue of elements of type ITEM, or it could contain a queue of objects. In the latter case, the `add:` method would convert the element into an object, which would then be enqueued.

The advantages of making the buffer an object are:

- the implementation of the buffer is hidden from the body of the task.
- the buffer object inherits methods from its superclasses.

6.5.3 Third Attempt (Messages as Objects)

A further refinement is to treat inter-task messages as objects. Unison's request and reply buffers are usually structures or unions of structures. However, in a typical application they are not treated as abstract data types, in the same manner as a stack or queue. Knowledge of the names and types of the structure fields are scattered throughout the code.

The idea that messages should know how to send, receive and reply themselves fits well with concept of object-oriented programming (that is, that the state of an object should only be altered by that object's methods).

Send, receive and reply operations are common to all inter-task messages, while the meaning of the structure fields vary from message to message. This suggests specialization and subclassing.

We define an `InterTaskMessage` class:

```
=InterTaskMessage:Object {  struct ... request;
                           struct ... reply;
                           unsigned sender, receiver; }
```

Its instance variables include the structures for the reply and request buffers. The instance methods include:

to expand in order to contain additional items. For our purposes, in order to obtain a buffer class with a fixed storage capacity, `BoundedBuffer` was implemented as a subclass of `Object`, none of Objective C's container classes were used.

```

~(unsigned)sendTo:(unsigned)taskId thru:(unsigned)portMask
// send a message to a specified task

~(unsigned)receiveFrom:(unsigned)portMask
// receive a message through the specified ports

~(unsigned)replyTo:(unsigned)taskId
// reply to the specified task

```

As the names imply, these methods map directly into calls to the corresponding Unison functions. The methods return the task ids that are returned from the Unison primitives.

Next, for the bounded buffer task we define a subclass BufferMsg. This subclass defines methods for storing and removing the items from the fields of the message structures.

```

=BufferMsg:InterTaskMessage

~(ITEM)retrieveElement{...} //retrieve element from the
                             request structure

~(void)storeElement:(ITEM)anElement{...}
                             //store the element in the
                             reply structure
=

```

The major problem with this approach is that the InterTaskMessage class contains the request and reply buffers. These structures will undoubtedly change with each application, hence the code for the class would have to be edited and recompiled each time it is used.

The solution is to move the request and reply structures into the BufferMsg subclass. Now the problem is: what do the sendTo, receiveFrom and replyTo methods use as buffers for the Unison message passing calls? When the InterTaskMessage class is compiled it has no knowledge of any subclass that may reference it and so cannot access the instance variables defined there.

One possible solution is illustrated using the code for the `receiveFrom:` method (in the `InterTaskMessage` class):

```
-(unsigned)receiveFrom:(unsigned)portMask { char *p;

    p = _getvec( [self requestSize] );
    *(int *)p = [self requestSize];
    sender = _receive( p, portMask, ... );
    [self copyFrom:p];
    _freevec(p);
    return sender; }
```

Access to the request structure defined in the `BufferMsg` subclass occurs by sending messages to “self”. The `requestSize` and `copyFrom:` methods are implemented in the `BufferMsg` subclass.

In the `receiveFrom:` method, a block of memory is allocated using `_getvec()`. `[self requestSize]` replies the size of the request buffer, and is implemented as:

```
return( sizeof( request ) );
```

Next, Unison’s `_receive()` function is called. When an inter-task message has been received, ‘copyFrom:p’ is sent to ‘self’. This method copies the received message from the buffer pointed to by `p` into the request structure defined in `BufferMsg`:

```
-(void)copyFrom:(char *)p{ struct REQUEST *r;
    r = (struct REQUEST *)p;
    request.MSG_SIZE = r->MSG_SIZE;
    request.MSG_TYPE = r->MSG_TYPE;
    request.ELEMENT = r->ELEMENT; }
```

Once the message has been copied `receiveFrom:` deallocates the memory block (via `_freevec()`) and returns the id of the correspondent task.

While this approach is quite tidy, it is inefficient. There is a performance penalty associated with the memory allocation calls, the repeated messages to self, and copying the fields of the message structure.

A more efficient implementation of `receiveFrom:` is:

```
-(unsigned)receiveFrom:(unsigned)portMask { char *p;
    p = [self addressOfRequestStruct];
    sender = _receive( p, portMask, ... );
    return sender; }
```

The `addressOfRequestStruct` method (implemented in `BufferMsg`) returns a pointer to the request buffer instance variable, cast as “pointer to char”. This pointer is used as an argument to the `_receive()` call. With this, the `InterTaskMessage` class is accessing an instance variable of its subclass through a pointer: Unison’s `_receive()` primitive transfers the received message directly to the structure defined in `BufferMsg`.

The third version of the bounded buffer task, using `InterTaskMessage` objects, is shown in Figure 6.7.

[illegible]

Figure 6.7: Bounded Buffer, Using Message Objects

```

/* loop forever, processing requests */

for( ; ; ){
    sender = [aMessage receiveFrom:port_mask];
    switch( [aMessage type] ){
        case PUT:                                /* from the producer */
            [aMessage replyTo:sender];
            [aBuffer add:[aMessage retrieveElement]];
            if( [aBuffer size] == SIZE )
                /* buffer full, accept only GET requests */
                port_mask = CONSUMER_MASK;
            else port_mask = PRODUCER_MASK | CONSUMER_MASK;
            /* accept from producer or consumer */
            break;

        case GET:                                /* from the consumer */
            [aMessage storeElement:[aBuffer remove]];
            [aMessage replyTo:sender];
            if( [aBuffer size] == 0 )
                /* buffer empty, accept only PUT requests */
                port_mask = PRODUCER_MASK;
            else port_mask = PRODUCER_MASK | CONSUMER_MASK;
            /* accept requests from either */
            break;
    }
}
}

```

Message objects are useful for more than encapsulating the details of inter-task communication. Consider an administrator task which accepts requests from a number of client tasks and worker tasks. Suppose that a client task sends a request when all the workers are busy. The administrator, if implemented in C, must extract the details of the work request from request buffer and enqueue that information, along with the task id of the client, in a “work requests pending” queue. When a worker arrives, this information is dequeued, and the request is processed. In a similar fashion, the administrator maintains a queue of available workers when there are no pending requests from clients.

Using message objects, the administrator can be simplified. A new instance of

the `message` class is created for each message that is received. If no workers are available when a client sends a request, then the entire message object is enqueued. (In practice, this is not an expensive operation, as it is only the pointer to the instance variables that is enqueued.) When the request is eventually processed and the client replied to, then the message object is freed.

6.5.4 Fourth Attempt (Buffering of Objects)

A further refinement to the buffer task is possible. Notice that in the second version, both the bounded buffer task and the `BoundedBuffer` class knew the primitive type of the elements that are buffered. Similarly, in the third version, the `BufferMsg` class and the `BoundedBuffer` class both require the typedef for `ITEM` in order to know the size of the buffered elements. Although a typedef has been used so that the code can be easily modified, it is unfortunate that the two classes must be modified if we wish to buffer a different type of element. This could be avoided if the elements themselves were objects, and `BoundedBuffer` were designed to store any object. As noted at the beginning of this chapter, one of the drawbacks of using a hybrid object-oriented language is that we must deal with both the primitive types and the objects supported by the language.

At this point, we could consider reworking the `InterTaskMessage` so that object ids were transferred between tasks. Note, however, that an object id is merely a pointer to the data structures which implement an object. If we send an object from one task to another, the receiving task does not receive a copy of all of the object's instance variables. Instead, it receives a copy of the pointer to those variables. In a concurrent environment, this approach violates mutual exclusion, in that two tasks will have a pointer to the same locations in memory (the shared object).

The solution we propose elaborates on our previous discussion about administrator tasks. We redesign the `BoundedBuffer` class so that it stores object ids. We alter the buffer task so that, instead of obtaining elements from `aMessage` and sending those elements to `aBuffer`, it stores the entire `aMessage` object in `aBuffer`. (Actually, it is the object ids, the pointers to the object structures, which are stored in `aBuffer`). The code for the buffer task is changed to instantiate a new instance

of **aMessage** within its main loop. If the received message is a PUT request, then **aMessage** is stored in **aBuffer**. When a GET request is received, a message instance is obtained from **aBuffer** and told to reply itself to the consumer task. Notice that there are two message instances which must now be deallocated: the message instance that replied itself, and the message instance which received the GET message. The code for this fourth attempt follows, its iconic representation is in Figure 6.8.

```
/* bounded buffer - buffer object buffers message objects */

extern id BoundedBuffer;      /* factory object for the queue */
extern id BufferMsg;          /* factory id for this task's msgs */

#define SIZE 10                /* buffer size */

Bounded_buffer(){

    unsigned sender;
    struct ... request, reply;
    id aBuffer, aMessage, removedMessage;
    int i;

    /* initialization */

    aBuffer = [BoundedBuffer new:SIZE];    /* create the buffer */

    port_mask = PRODUCER_MASK;             /* initially, accept only
                                             requests to store data */

    /* loop forever, processing requests */

    for( ; ; ){
        aMessage = [BufferMsg new];
        sender = [aMessage receiveFrom:port_mask];
        switch( [aMessage type] ){
            case PUT:                        /* from the producer */
                [aMessage replyTo:sender];
                [aBuffer add:aMessage];
                if( [aBuffer size] == SIZE )
                    /* buffer full, accept only GET requests */
                    port_mask = CONSUMER_MASK;
```

Figure 6.8: Bounded Buffer, Buffering Message Instances

```

        else port_mask = PRODUCER_MASK | CONSUMER_MASK;
        /* accept from producer or consumer */
        break;

    case GET:                                /* from the consumer */
        removedMessage = [aBuffer remove];
        [removedMessage replyTo:sender];
        [removedMessage free];
        [aMessage free];
        if( [aBuffer size] == 0 )
            /* buffer empty, accept only PUT requests */
            port_mask = PRODUCER_MASK;
        else port_mask = PRODUCER_MASK | CONSUMER_MASK;
        /* accept requests from either */
        break;
    }
}
}

```

For this approach to work, we need a way for objects contained in the producer task to convert themselves into instances of `BufferMsg`, so that they can be sent to the buffer task. Similarly, when the consumer task receives a `BufferMsg` instance from the buffer task, the original object must be extracted from the message instance.

There are two approaches to support this. First, those objects that are to be transferred between tasks could provide methods which convert the object's instance variables into values of the appropriate primitive C types. These values would then be sent to an instance of a message class (a subclass of `InterTaskMsg`) which is designed for that object. The message class is responsible for storing the primitive values into the structures used for inter-task messaging. To reconstruct an object instance after it has been sent between tasks, the message class must provide methods which return primitive values from the messaging structures. These values are used to initialize the instance variables of a new instance of the corresponding object.

This approach was illustrated with our `BufferMsg` class, which contained `retrieveElement` and `storeElement` methods to handle the primitive types that were passed between tasks.

A more elegant approach is to provide objects with the capability of transforming or mutating themselves into instances of an appropriate message subclass. Similarly, each message subclass must be able to transform itself into its corresponding object. After a message object has received an inter-task message, it must be able to instantiate an object of the appropriate form, initialized with values extracted from the fields of the received message.

6.5.5 Summary

This section has illustrated the design of an object-oriented buffer task. In the first attempt no objects were used. The data structures used for Unison's message passing, the buffer data structure, and the controlling algorithm for the task are tightly bound.

In contrast, in a "pure" object-oriented approach, all entities manipulated within the task are objects. The intertask messages, the buffer, and the task algorithm are loosely coupled. The price for this is the overhead required by Objective C. To take full advantage of the object-oriented paradigm, within the task all entities are viewed as objects. The mapping to the underlying primitive types supplied by C, which is required for the inter-task messaging, is hidden within the objects.

It appears that the object-oriented approach to design and implementation blends well with the techniques currently used to design Unison applications.

At the "system" or "process" level, the application is designed as a group of communicating tasks. While tasks would not be directly implemented using Objective C classes, the task model shares many of the characteristics of Objective C objects. Tasks communicate by sending and replying messages. Tasks support abstraction and information hiding. The possibilities for inheritance are limited, but we believe this is not crucial at this level.

The next step is the design of the internals of a task. Here, we can exploit the use of Objective C objects. Data structures, including those used to support inter-task messaging, are implemented as objects. The C code in the task body provides the overall controlling algorithm for the task.

Chapter 7

Objective C Support for Unison

7.1 Language Support for Embedded Systems

Embedded systems are often unable to support the development of the software that runs on them. They may lack the hardware (for example, memory or disk storage) or software (for example, an operating system) that is required to run the software development tools. In this case, applications may be developed on a *host* computer; the binary image is then transferred to the *target* embedded system for execution. This requires that the executable software have no dependencies on features present only on the host system.

In this light, we describe the development of C programs [24] intended to be run under Unison [31, 32] in a multiprocessor environment. Programs are compiled and linked on a Sun workstation, running under the Unix operating system [25], then downloaded to the multiprocessor target hardware. The target system consists of a number of single board computers based on the Motorola 68010 microprocessor [30], housed in a VMEbus chassis [39]. There are a number of restrictions when developing Unison applications in this environment:

- Unison applications cannot make Unix system calls. Since program object modules are linked on a Sun workstation, it is possible to successfully link Unison applications which make these calls; however, as Unix is not present on the target system these programs will fail to execute properly.

- Care must be taken as to which routines in the *standard C library* are linked to a Unison application. C is a small language. Input and output operations are not provided as part of the language. Rather, I/O routines are provided in an independently compiled library which are linked with the application code. Similarly, C does not provide statements to manipulate character strings; these routines are also provided as part of the standard library. The rationale for this approach is that it is possible to change the implementation of the libraries without altering the compiler.

Unison applications may use those library routines which are not themselves dependent on Unix. For example, the *strlen()* function, which returns the length of a character string, may be called as its implementation does not make calls to the operating system (or call functions which in turn make system calls). However, output through *printf()* is not supported, because this function references output routines provided by Unix.

- The C compiler may generate calls to undocumented support routines which are machine dependent, so the programmer must avoid writing C code which results in these calls. For example, the C compiler generates a function call when long integer values are multiplied; this routine does not refer to any other functions, and so can be linked without problem into a Unison application. In contrast, the floating point support routines are tied to both the Sun's operating system and hardware; hence floating point operations are not supported by Unison.

7.2 Issues with Objective C

In order to prototype our object-oriented real-time applications, we must be able to create these applications in the host environment, and then execute them on the target hardware. Obviously, we face the same constraints as discussed in the previous section. In addition, Objective C [34, 33] extends the C language, providing a syntax for supporting object-oriented programming. We must verify that the code generated by this compiler is suitable for our purposes. There are

run-time issues which are not present with traditional C programs. Specifically, the run-time binding of messages to method addresses has no counterpart in C. Instances are created and deleted throughout the life-time of an object oriented program; we must provide the necessary mechanism to support this in the target environment. Objective C's run time error handling is far more comprehensive than that provided by C implementations, and this also should be provided in the target implementation.

Objective C was chosen as the prototyping language, in part because the source code for the run time support routines and the class library is provided with the compiler. This allows us to provide support for Unison by modifying the code, rather than "patching" binary images intended for use under Unix.

Objective C is a preprocessor, which converts expressions written in the Objective C syntax into standard C code. The compiler generates the data structures required to support instances, classes, and inheritance, as well as calls to routines to support the run-time binding of messages and methods. The resulting C source is then passed to the Unix C compiler to complete the compilation process.

As such, the code generated by the Objective C compiler contains no dependencies on Unix other than those that occur in the equivalent C source. As discussed earlier, we can verify whether a particular C source file will result in code that can execute successfully on the target hardware.

7.3 Class Definitions and Message Expressions

As described in Chapter 2, Objective C extends the C syntax to allow the definition of classes. The Objective C compiler converts the names of the factory and instance methods defined in a class source file into C function definitions. The keyword arguments of a method are defined as the formal parameters to the corresponding function. Each method is translated into the C code for the function.

When a class is compiled, several C structures are created to support the creation of objects; these are described in the next section.

The Objective C compiler recognizes message expressions enclosed in square

brackets, for example:

```
[ AClass new ];
[ anInstance doThis ];
[ anotherInstance doThis:arg1 with:arg2 ];
```

These expressions are translated into calls to one of two *messenger* functions, *msg()* and *msgSuper()*. These functions implement the dynamic (run-time) binding, determining which method should be executed when object receives a message. *msg()* is passed a pointer to the receiving object and the message selector; as described in the next section, this provides sufficient information to locate the function associated with the message. The *msgSuper()* function performs a similar messaging operation, but is called only when a message is sent to *super*. This function is passed a class and a selector as arguments.

7.4 Object Implementation

Now, we briefly examine how objects are implemented in Objective C.

The language adds a new type, *id*, to the base types provided by C. All instance and class objects are stored in variables of this type; that is, each variable of this type is the *id* of some object. As an example, we create a new instance of the class *SomeClass* by sending it a *new* message; the returned instance is stored in variable *anInstance*:

```
extern id SomeClass;
id anInstance;
anInstance = [ SomeClass new ];
```

Figure 7.1 shows how objects are implemented by the Objective C compiler.

The object *anInstance*, of type *id*, is a pointer to a C structure that contains all of the instance variables of the object. The instance variables defined in a class are concatenated with any instance variables inherited from the superclasses into a single structure. This is referred to as the *private part* of an instance. For each

Figure 7.1: C Structures Implementing Objects

instance that is created, a contiguous block of memory large enough to hold the object's instance variables is allocated, and the pointer to that block is used as the object's id.

The first entry in the private part of an object is the *isa* variable¹. This contains a pointer to a structure known as the *shared part* of the instances, so called because the *isa* variables for all instances of a class point to a single shared structure.

This structure contains a number of fields, including another *isa* variable (discussed below). The *clsSuper* field is a pointer to shared structure of this object's superclass; it provides the mechanism for inheritance of instance methods. The *clsDispTable* field is a pointer to the dispatch table of instance methods. The dispatch table contains a pair of pointers for each instance method defined in the class. The *cmd* field points to a string containing the name of the method (the selector). The *imp* field is a pointer to the function which implements the method.

We now consider the operation of the Objective C messenger when a message is sent to an instance, for example:

```
[ anInstance thisSelector ];
```

The receiver of the message, *anInstance*, is a pointer to the instance variables of that instance. The first of these, the *isa* link, points to that shared structure for the all instances of the instance's class. Within this structure, the messenger locates the dispatch table of instance methods, and compares the message sent to the instance, *thisSelector* with each of the selector names in dispatch table. If a match is found with one of the entries in the table, then the corresponding function is called through the function pointer, to invoke the instance's method.

If the method *thisSelector* is not defined in the class *SomeClass*, but instead was inherited from a superclass, the messenger will fail to locate a match in the dispatch table. If this occurs, the *clsSuper* link is followed to reach the shared structure for the immediate superclass. Then, the dispatch table of instance methods defined by this class is searched. This continues until either the appropriate method has been found, or the dispatch table of instance methods for the Object class has

¹*isa* is the single instance variable defined in the Object class. All instances inherit from the Object class, therefore all instances have *isa* as one of their instance variables.

been searched without success. In the latter case, the instance cannot respond to the message that was sent to it, and a *doesNotRecognize* message is sent to *self* to handle the error.

Factory objects (classes) are, like instances, identified by variables of type *id*. As noted above, an instance *id* points to the instance's private structure, that is, its instance variables, which in turn points to the shared structure for all instances of that class. In a similar fashion, the *id* for a class is a pointer to the structure shared by the instances. This is shown in Figure 7.1. *SomeClass* points to the same structure that is pointed to by *anInstance*'s *isa* link. The *isa* link in this structure points to an identical structure, referred to as the *shared part of the class*. Within this structure, the *clsDispTable* field points to the dispatch table of class (factory) methods. The *clsSuper* field points to the shared class structure for the immediate superclass, allowing inheritance of factory methods.

When a message is sent to a class:

```
anInstance = [ SomeClass new];
```

the run-time binding of selector to method code is handled in a manner similar to that used for instances. In this case, the receiver of the message is the class *SomeClass*. This pointer is traced by the messenger to the class' *isa* link, which in turn is traced to its shared structure. The dispatch table of class methods is searched for the *new* selector. If this method was not over-ridden by *SomeClass* and was instead inherited from a superclass, then a matching table entry will not be found, so the class methods defined in the immediate superclass will be located by following the *clsSuper* link. The factory methods dispatch table for class *Object* contains a *new* method; this will be found and executed if none of the subclasses between *Object* and *SomeClass* over-rode this method.

When *new* is executed, space for the instance's private structure is allocated, and the *isa* instance variable is initialized to point to the instance's shared structure. The pointer to the new private structure is returned as the *id* of the new instance.

Given this implementation of objects, there are two points to consider regarding support for Objective C programs under Unison:

- The Objective C compiler creates objects in terms of linked C structures, which can in turn be compiled using the Sun's C compiler. There are no dependencies on a particular operating system or C compiler implementation. As "traditional" Unison applications are built using the Sun compiler, subject to the restrictions described earlier, the code generated by Objective C should not pose difficulties.
- The structures that describe classes are created and initialized when an application is compiled and linked. However, instance creation requires the ability to allocate arbitrary blocks of memory to hold the instance variables. Unison directly supports dynamic memory management, and so can support the run time creation of instances. This is discussed further in Section 7.6.1.

7.5 Porting the Class Library

We have seen that the code generated by the Objective C compiler is suitable for use in a Unison application. It is not necessary to edit the C code produced by the compiler to remove any dependencies on Unix prior to submitting this code to the C compiler.

Now, we consider the changes to the source files supplied with the compiler, in order for these routines to run under Unison. The sources include the code for the *Foundation classes*, written in Objective C, and a number of support functions written in C and assembly language, which are called by the class library.

The majority of the changes to the class library reflected differences in the input and output routines provided with Unix and Unison. These are discussed next.

7.5.1 I/O Support in Unix and Unison

Input and output in Unix is provided through the abstraction of *stream I/O*, in which files and devices are accessed as a sequence of bytes [24]. No additional record or block structure is imposed on the I/O streams by the operating system. This model provides uniform support for I/O to disk files and serial devices (terminals,

modems, and the like); the same set of system calls is used to access files and serial connections. Unix provides a system call to “open” a file or device for access; this call associates a *file descriptor* with the opened connection. Further references to an open file refer make use of this file descriptor. Unix provides *read()* and *write()* calls to transfer an arbitrary number of bytes to a specified file descriptor. These calls are entry points into the operating system, and provide what is known as *unbuffered I/O*.

C programs which contain Unix system calls are portable to other operating systems only if identical system calls are provided. To remove dependencies on a particular operating system, C implementations are supplied with a *standard library* of I/O routines [24] which are built upon the I/O primitives supplied by the operating system. These routines provide *buffered I/O* and support the stream abstraction described above. Files and devices are accessed through *file pointers*. (A file pointer is a pointer to a structure containing information about the file. Under Unix, this information includes the file descriptor used by the Unix system calls.) There are functions in the standard library to read and write single bytes (*getc()*, *fgetc()* and *putc()*, *fputc()*), strings of characters (*fgets()* and *fputs()*), or arbitrarily sized buffers of bytes (*fread()* and *fwrite()*).

Under Unix, three streams are open when a program starts execution. The file pointers for these streams are referred to as the standard input (*stdin*), standard output (*stdout*), and standard error (*stderr*). Normally, these are connected to the same computer terminal. A number of the functions in the standard library use *stdin* and *stdout* implicitly: *getchar()*, *putchar()*, *gets()*, and *puts()*, among others.

As described in Chapter 4, Unison’s input and output services are supported by the concept of *servers*. Unison provides a set of I/O routines based on the stream I/O model provided by Unix. The kernel has a set of I/O primitives which provide unbuffered input and output. A client task opens a connection with a server by calling the *_open()* primitive. Information about the opened connection is stored in a *user connection block (UCB)*; a pointer to a UCB is returned on successful completion of *_open()*.

Unison has a standard I/O library which supports buffered input and output,

analogous to its Unix counterpart. These routines are built upon the kernel I/O primitives, and refer to open streams via pointers to UCBs (in the same way that the routines in the standard C library use file pointers).

There are a number of differences between the I/O support in Unix and Unison:

- The names of the Unison I/O primitives and standard I/O functions are prefaced with an underscore. As such, the counterparts to Unix's *read()* and *write()* system calls are called *_read()* and *_write()* in Unison. Similarly, the *_getc()* and *_putc()* functions in Unison's standard library are equivalent to *getc()* and *putc()* supplied with the Unix C compiler.
- The Unix system I/O calls make use of file descriptors, while calls to the standard library specify file pointers. With Unison, calls to both the kernel's I/O primitives and the Unison library functions use pointers to UCBs to identify the open stream.
- Unix programs may assume that the *stdin*, *stdout*, and *stderr* streams have been opened by the operating system. In contrast, each Unison task that wishes to perform I/O must open a connection with a server. Two functions, *_select_input()* and *_select_output()*, are used to select the default input and output streams, on a per task basis. These streams may then be referred to as *_stdin* and *_stdout*, and are used implicitly by some library functions, including *_getchar()*, *_putchar()*, *_printf()*, and *_scanf()*. Unison does not provide a *_stderr* connection.
- Under Unix, *stdin*, *stdout*, and *stderr* are “#defined” as pointers to three file pointer structures. In Unison, *_stdin* and *_stdout* are functions which return a pointer to the current default input and output connections.

Part of porting the Objective C class library involved rewriting calls to the C standard library to the equivalent calls Unison standard I/O functions. The rewritten methods assume that the task context in which they run has previously opened connections to a server, and that default input and output streams have been selected via *_select_input()* and *_select_output()*. Without this assumption, it would

be necessary to open a new connection to a server each time a method wanted to perform I/O. In addition to the inefficiency of this approach, an individual method has no knowledge of what servers have been installed in system. As such, it is preferable that each task in which the class library is used be responsible for setting the default I/O connections with the appropriate server.

As Unison does not provide a standard error stream, any methods which wrote to *stderr* were rewritten to use Unison's *_stdout* stream.

7.6 Substrate Functions

Run-time support for Objective C is provided by a number of *substrate functions*, written in C and assembly language. These functions are called by the *Foundation classes* supplied with the language, and support memory management, file input/output, error handling, and run-time binding. For the most part, these routines provide the interface between the class library and the standard C library.

Many of the substrate functions are referenced indirectly through “dispatch vectors” that contain pointers to the functions. For example, memory allocation is handled by the function *_alloc()*; its dispatch vector is initialized as follows:

```
extern id __alloc();
id(* _alloc)() = __alloc();
```

These vectors are initialized when an application is compiled and linked. Although a program may install a new function at run-time by placing a pointer to the appropriate routine in the vector, the programmer may instead choose to link in new substrate functions, rather than use those provided with the language. We have followed the latter approach when porting Objective C to Unison.

Some of the existing substrate functions are suitable for use under Unison; for these, we merely verified their operation. Several of the functions required extensive modification in order to run under Unison.

Obviously, when porting C code from Unix to another operating system, a number of factors influence the extent to which the code must be altered. “Portable” C

programs do not make Unix system calls; instead, they use the functions provided by the standard library. Given this, we must then consider how closely the standard library provided with the target operating system resembles that supplied with Unix. In the previous section, we discussed the differences between the I/O routines supplied with the Unix and Unison standard libraries. The substrate functions make use of these I/O functions, as well as a number of other standard routines.

During the initial prototyping, the substrate functions were ported to the Harmony operating system. Midway through porting, Unison became available, and our efforts were shifted to that system. Unison's support libraries follow those provided by Unix much more closely than Harmony's, and as one might expect, this simplified porting considerably.

The following sections are not meant to provide an exhaustive guide to the port. Rather, we provide an overview of the changes to a number of the substrate functions, to illustrate the issues that were considered and highlighting the tradeoffs required to move the routines from the Unix environment to Unison.

7.6.1 Memory Management

Three routines are provided to handle the memory allocation for the instances' private structures:

- `_alloc()` is called to allocate memory when instances are created. This routine called the Unix `malloc()` routine, and was changed to call the similar `_getvec()` function provided by Unison.
- `_dealloc()` deallocates the memory associated with a freed object. The original code called the Unix function `free()`; the modified code calls Unison's `_freevec()`.
- `_realloc()` changes the amount of memory allocated to an object, via the Unix call `realloc()`. An equivalent routine is not provided by Unison, hence it is necessary to determine the size of the current buffer through a call to `_sizeof()`, acquire a new memory buffer via `_getvec()`, copy the contents of the original buffer there, and release the old buffer by calling `_freevec()`.

The memory management routines support *allocation tracing*. As calls are made from the class libraries to these substrate functions, appropriate traces are output, allowing the user to monitor the run-time behaviour of the program. The existing code defines:

```
IOD allocIOD = stdout;
```

IOD is a typedef for a Unix *file pointer*, so this declaration indicates that the allocation traces are to be written, by default, to the Unix *standard output* (normally the computer's display). This code was changed so that output is sent to Unison's `_stdout` UCB. The assumption here is that the task context in which these routines are executing have an open UCB to an appropriate server, as described in Chapter 4.

7.6.2 Filing Support

Four functions are provided which call routines in the ASCII filer class, which converts objects between their internal format and a representation as ASCII characters. This supports long term storage of objects using the host computer's file system. As the current release of Unison does not support a file system, the routines `_fileIn()`, `_fileOut()`, `_storeOn()`, and `_readFrom()`, were rewritten to always return an appropriate error indication.

7.6.3 Error Handling and Program Termination

The substrate function `_error()` is called when a critical error occurs, to print an error message and then terminate the program via a call to `bye()` (described below). The Unix implementation writes the message, by default, to the *standard error* port, although this can be altered by the user. The modified version writes to Unison's `_stdout` UCB, and assumes that an active server connection exists.

`bye()` calls any exit handler functions which have been installed by the user, the calls the standard C function `exit()` to terminate the program. Under Unison, this function calls the exit handlers, sends a message to `_stdout`, then enters an infinite loop.

The `__settrap()` and `__settrapList()` functions are used to set traps for Unix signals. These routines are not supported under the Unison implementation.

7.6.4 Messaging Support

Earlier, we described how the run-time binding was provided by the messaging functions `msg()` and `msgSuper()`. In practice, the navigation through the dispatch tables in order to locate functions at run-time is somewhat time consuming. To improve the efficiency of the dynamic binding, the messenger routines maintain a cache to speed up message dispatching. This cache contains the addresses of the most recently accessed method functions. Each slot in the cache is conceptually of the form (class, selector, method-address). `msg()` and `msgSuper()` derive the first two items from the arguments passed to them, and use these to locate the pointer to the function implementing the method. If the method is not found, then the dispatch tables are searched, as discussed earlier. The cache is then updated with the address of the method.

In a multitasking environment, the cache is a critical region. While the cache is being accessed in the context of one task, an interrupt may occur. Then, another task may be scheduled which in turn calls the messenger and accesses the cache. To prevent the cache from being corrupted, the cache is locked via the 68010 TAS (test-and-set) instruction to prevent reentrant accesses while the cache is tested or updated. If the messenger finds that the cache is locked, then this forces it to search the dispatch tables for the method. This search is the time-consuming portion of the dynamic binding, and is to be avoided if possible.

A number of other approaches to protecting the cache, while minimizing the search of the dispatch tables, are possible. When executing on the target hardware, Unison programs run in the *supervisor mode* of the 68010. As such, it is possible to lock out interrupts directly prior to accessing the cache, by altering the processor's status word. Context switching is then prevented during operations on the cache, so reentrant accesses are prevented. This approach does have drawbacks, however. The Unison simulator runs on the Suns, executing the Unison application and kernel as a single Unix process. This process is run in *user mode*, so modification of the

processor status word by the program is prohibited.

The Unison kernel provides `_enable()` and `_disable()` functions to lock out interrupts. The differences between the implementation of interrupt lockout in the target and simulator versions of a program are handled transparently. Our versions of the messaging routines were rewritten to call these functions, in order that the same code could be used in the two environments. While this incurs the overhead of a function call each time we wish to enable or disable interrupts, this is not overly expensive: the message passing primitives within the Unison kernel make use of the same functions during critical sections of code.

7.7 Deallocation of Resources

In a typical Unison application, tasks obtain and release resources as the program executes. A task can obtain blocks of contiguous memory on the processor on which it executes by calling the kernel's `_getvec()` primitive. A block of memory that is no longer required can be returned to the operating system by calling the `_freevec()` primitive. A storage block may be freed only by the task which initially obtained it from the storage pool. Open connections to servers are also considered to be resources. An connection can be closed only by the task which opened it. Furthermore, tasks are maintained in a hierarchial relationship. Each task is considered to be the child of the task which created it.

When a task is destroyed, all of the resources allocated to it are returned to the operating system. Any open server connections are closed, and the blocks of memory the task obtained from the kernel are automatically returned to the storage pool. In addition, all of the descendants of the task are destroyed, and any memory blocks or server connections allocated to those tasks are released.

Object-oriented environments often provide automatic garbage collection [11]. The lifetime of an object is determined by the environment, not by the application program. Using a garbage collection scheme, any memory that has been allocated to an object is automatically reclaimed when the object is no longer referenced.

Objective C does not provide for automatic grabage collection. It is the pro-

gram's responsibility to deallocate objects when they are no longer required.

We have seen how Objective C's objects are implemented as linked C structures. When an object is instantiated, space for its instance variable is allocated through the `_alloc()` substrate function, which in turn calls Unison's `_getvec()` primitive. To the Unison kernel, each object instance is a block of memory which have been allocated to a particular task. As such, the rules for resource deallocation apply to objects. The task in which an object is instantiated is responsible for deallocating the instance. When a task is destroyed, any objects belonging to that task or one of its descendant tasks is destroyed.

As described in Chapter 6, passive objects are contained within Unison tasks. Object pointers are not passed between tasks, and object instances are not shared between tasks. Given this, Unison's scheme for resource deallocation does not conflict with Objective C's implementation of objects. We have avoided the possibility that an object id would be left pointing to a block of memory that had been returned to the storage pool or reallocated.

Chapter 8

Prototype Implementations

In this chapter, we discuss prototyping simple applications which make use of our object-oriented framework for Unison. Tradeoffs and limitations involved in object-oriented development are discussed.

8.1 Tradeoffs Using Object-Oriented Development

In previous chapters, we have described the development of an object-oriented framework for real-time applications. We have discussed the benefits of object-oriented design and development; specifically, the object-oriented paradigm provides a natural model for design, flexibility in prototyping implementations, and supports reduced maintenance costs and high reuse of code. This discussion has ignored two aspects central to real-time embedded systems, namely, the memory usage and execution overhead incurred by the object-oriented platform.

The development of hardware for real-time systems has kept pace with the rapid evolution of computer systems in general. Although the capabilities of single board, microprocessor-based computers have increased in the past 15 years, they do not present as rich an execution environment as, say, the workstations used as the host development systems for our research. Our target environment consists of single board computers based on the Motorola 68010 microprocessor [30]. While each of these boards contains up to 1 megabyte of random access memory, there

is no hardware and software support for virtual memory as provided by the Sun workstations. The megabyte of RAM constrains the size of application that a target board will support.

In a multiprocessor configuration, each processor occupies a portion of a 16 megabyte linear address space. Each processor can access the RAM on the other cards. All communication between target boards is across a shared backplane bus known as a VMEbus [39]. Each board wishing to access memory on another target card must request permission to use the bus; one board in the multiprocessor arbitrates between multiple bus requests, and grants use of the bus appropriately. The speed with which a processor can access a location in memory is dependent, in part, on both the frequency of the microprocessor clock (which determines the amount of time required for each instruction cycle) and the hardware which supports memory accesses across the bus. Obviously, increased requests for bus accesses may lead to contention, limiting the performance of the multiprocessor.

With this in mind, we now consider our object-oriented framework in terms of the constraints provided by the target hardware.

8.1.1 Memory Requirements

Previously, we have described porting the Objective C run time routines to Unison. At a minimum, any object-oriented application will require that the Object class and the substrate functions be linked into the code. Of course, this code is not required for a standard Unison/C application, so it is overhead directly incurred by the object-oriented approach. Compiled on the Sun workstation using the Sun OS 3.5 C compiler, the minimal set of object run-time support routines are approximately 25,000 bytes in size. This is somewhat smaller than the foundation library provided for Unix, in large part because of the code removed while porting the routines to Unison.

As outlined in Chapter 7, each class in an application requires the two “shared” structures which describe the class, as well as the dispatch tables for the factory and instance methods defined by the class and pointed to by the structures. The amount of memory required for this varies, depending, in part, on the size of the

dispatch tables.

The additional code bulk necessitated by the object support routines is misleading, because it does not take into account the code sharing supported by the class hierarchy. As discussed in Chapter 7, the functions which implement the instance methods for a class are shared by all instances of that class. Similarly, any two classes share the methods inherited from their common superclass and any classes above it in the hierarchy. Methods defined in a subclass add code bulk, as do any methods which the subclass overrides. However, the “cost” of methods inherited from a superclass is the memory required for the structures contain the pointers to the superclass, as described in the previous chapter. In this way, a narrow, deep inheritance tree maximizes the code sharing, in comparison with a broad, shallow tree.

In addition to the minimal set of run-time support routines, all of our applications use the `InterTaskMessage` class to encapsulate the inter-task message passing. We have found that this results in a net reduction of code bulk, when compared with the equivalent Unison application coded in C. In the latter case, the “housekeeping” associated with calls to `_send()`, `_receive()`, and `_reply()` is repeated throughout the code, whereas this detail is encapsulated in the methods provided by the `InterTaskMessage` class. This is illustrated in the code for the bounded buffer example, presented earlier.

As discussed in Chapter 7, when an instance is created the memory for its private structure is allocated dynamically. We can ignore the memory required by the instance variables; in the corresponding C program these would be identical data structures, either allocated dynamically or contained in “auto” variables. The additional cost incurred by the private structures are the field containing the *isa* link to the instance’s class, and the object id itself (that is, the pointer to the private structure). This amounts to two longwords (eight bytes) per instance.

In our prototyping, we have not used any of the container classes, that is, the classes that can hold other objects. In Objective C, Containers are implemented by using an array to hold the ids of the objects stored in the container. This implementation permits quicker access to the stored ids than tracing through pointers

implementing a linked list of ids. When we add an object to a full collection, the collection attempts to expand its capacity. With the current implementation, this involves allocating a new array (usually twice the size of the full array), copying all of the ids from the old array into the new one, and deallocating the first array. A pointer-based implementation would be easier to expand, but the existing approach trades quick access to a stored object for additional execution cost the collection becomes full. On our target system, there is another potential problem. As noted earlier, each single board computer has a maximum of one megabyte of RAM. All object instances accessed by a processor must reside in that board's memory. When a collection attempts to increase its capacity, it must keep the existing array until a larger array has been allocated and initialized. For collections containing a sufficiently large number of objects, the available memory on a target card may be inadequate to accomplish this, even though there is enough memory to hold the expanded array alone.

In addition to the bounded buffer designs described earlier, we have prototyped a number of versions of a simple two-task system, in which a parent task creates a single child task. The two tasks then repeatedly execute a send/receive/reply cycle. This application was run in both a uniprocessor and multiprocessor configuration. With this application, we attempted to examine the overhead incurred by encapsulating the inter-task messaging routines in objects.

For our prototype applications, we found Objective C implementations were larger than the equivalent C programs by 40% to 180%. Note that these sizes reflect the fact that the applications have trivial functionality, so the code directly concerned with the application is small compared with the object support routines. In a realistic application, we expect that these support routines would form a lower percentage of the total code bulk.

8.1.2 Execution Penalties

A real-time application reacts to external events, and as such, the application code must execute at a speed sufficient to keep up with the rate at which these events occur. A number of factors affect the speed of execution, including the clock speed

of the processor, the number of “wait states” on accesses to memory or peripherals, arbitration delays when accessing the common bus, and the efficiency of the real-time kernel. With all things being equal, we can state the obvious: as the number of instructions increases, the time required to execute the program grows.

The major performance cost associated with Objective C applications is the time required to perform the run-time binding of messages to methods. When a message is sent to an object the run-time binding involves, at a minimum, a call to the messenger function, locating the function implementing the appropriate method in the cache, and calling that function. If the method is not located in the cache, additional processing is required to scan the dispatch tables belonging to the object and its superclasses. In this case, methods declared in subclasses require less time to find than those declared in superclasses, because the dispatch tables are searched starting with the table belonging to the instance, and moving up through the class hierarchy to class Object. This is a tradeoff with the memory requirement discussed earlier: in general, methods declared higher in the hierarchy are shared by more subclasses, reducing the code bulk. Once a method has been placed in the cache, the cost to access it is independent of its location in the class hierarchy.

When an instance is created, the kernel is requested to allocate memory for the instance’s private structure, and pointers into the class hierarchy are initialized. Whether this is significant depends on the rate at which instances are created and whether the equivalent C program dynamically allocates its data structures, or if all data structures are allocated at when the code is compiled.

In a multiprocessor configuration, the application code for an executable processor image is linked to its own copy of Unison and the Objective C support routines. The code and data structures for classes and instances reside on the same target card as the application code which uses them. Each processor image has its own messenger and cache, which handles run-time binding on that processor only. We have confirmed, then, that the object support routines do not implicitly require accesses across the VMEbus (although, of course, an individual method may access memory on another processor card). Increased inter-object messaging will not directly bottleneck the shared bus.

Following the client-server model of program design, all resources are controlled by tasks, so applications programs will not normally access memory on another target card. All VMEbus accesses are thus concerned with inter-task message passing as supported by Unison. Each use of one of the methods in our `InterTaskMessage` class results in the same amount of intertask communication as the equivalent C application running under Unison, so we conclude that this class does increase the load on the VMEbus.

For our prototype implementations, an Objective C implementation was typically 18% to 25% slower than the equivalent C program. Whether this is acceptable depends on the requirements of a particular application. We note, however, that the sources of the performance penalty can be explicitly identified. The statements in each Objective C method execute at the same speed as compiled C code. Object messaging imposes the additional overhead, as message sends are implemented via calls to the messenger support functions. The message cache is used to avoid searching the class hierarchy for method code, each time a message is sent. In a real-time environment, it might be worthwhile to “seed” the cache with the addresses of the most frequently used methods of those objects with “long” life-times, as part of the initialization phase. Research into this is beyond the scope of this thesis.

Chapter 9

Conclusions

9.1 A Summary of Thesis Research

Real-time, concurrent, embedded systems are among the most complex of all software systems to correctly design and implement. One common approach to developing these systems is to write the software using the C language and run the application under a real-time kernel. Such an approach can be time-consuming, increasing costs during both the initial implementation and long term maintenance phases. This is particularly true if the system is designed and implemented using a variety of ad-hoc approaches. In this thesis we have explored one approach to reducing the life cycle costs of embedded systems: the application of the object-oriented paradigm to the design and implementation of real-time software. We have provided a framework for the development of applications written in Objective C, a hybrid object-oriented language, running under the Unison real-time operating system.

Individual research efforts included: an examination of the approaches used to design real-time and object-oriented programs, and the synthesis of a design methodology that draws upon approaches used in both fields; augmenting an existing iconic notation, used to describe concurrent systems, with extensions that support the semantics of object-oriented design; implementation of support routines to allow programs written in Objective C to run under the Unison operating system, in a real-time, multiprocessor environment; and an examination of the related perfor-

mance issues in the embedded system environment.

Specific accomplishments include the following:

- Buhr's iconic notation for describing concurrent systems [8] has been extended to support object-oriented design. In addition to providing icons which describe objects and the data and methods encapsulated within them, our notation allows us to depict messaging between objects, inheritance, and dynamic creation of instances;
- we have suggested a design model for real-time systems in which concurrent objects are directly implemented using the multitasking facilities of a real-time operating system. Within each task, passive objects are used to implement the data structures required by the task, and to encapsulate the mechanisms for intertask communication and synchronization. A practical example of this method has been illustrated using Objective C, a hybrid object-oriented language, and Unison, a multitasking, multiprocessor operating system. Our iconic notation was used to illustrate the designs;
- Objective C has been adapted for use in the Unison environment. This involved modifying the code which supports run-time binding of objects and methods, as well as altering a number of the substrate functions which interface the run-time system to the underlying operating system;
- the Objective C class library has been extended to support messaging between Unison tasks;
- validation and testing of the above items has been performed using a target multiprocessor system containing multiple single board computers.

Assessment of Stated Goals

We believe that concurrent systems are best developed using an easily understood design methodology. Unison supports an anthropomorphic, client-server, model of design. This approach maps well into the iconic design notation developed in *System Design With Ada*[8] for designing systems using the concurrency model supported

by Ada. The use of iconic notation to express system design has found acceptance within the Ada development community, and we believe this approach is of use with other design paradigms, as long as the notation adequately captures the semantics of the paradigm.

Our iconic notation allows us to depict graphical models of object-oriented systems. We feel this approach can be applied during the “brainstorming” phase that accompanies system design, as a means of documenting or presenting a completed design, and as a tool to teach concepts behind object-oriented programming.

The anthropomorphic approach to concurrent system design shares a number of characteristics with the object-oriented paradigm. This has lead to what we believe is a natural model for the design of object-oriented concurrent systems, incorporatating both active (concurrent) and passive (non-concurrent) objects. For implementation purposes, the concurrent objects map cleanly into Unison tasks. Non-concurrent objects are contained within these tasks, and are implemented using the Objective C language. Our iconic notation supports this object-oriented design, both at the level of active objects (Unison tasks) and passive objects (Objective C objects).

The modified Objective C run-time support library allows object-oriented applications to be prototyped in a multiprocessor target system. The results of initial benchmarks are promising; the performance penalties and additional memory requirements imposed by the object support routines are sufficiently low that our framework appears to be acceptable for use in embedded systems development.

Assessment of Engineering Value

We realize that it is not the intent of this research to develop a commercial product; nonetheless, we feel that it is important to examine the extent to which our framework is applicable to practical development of real designs.

- **Acceptance of the Design Culture**

Underlying this research is the notion that applications pass through a design phase, supported in part by an iconic notation to describe and reason about

the designs. The software engineer must accept that this is a valid and more productive approach than incremental “hacking” towards a running system.

- **Development in Objective C**

The ‘C’ language is currently in vogue as a systems development language. This is not a language which requires or enforces “good” design; maintainable, reliable systems are more the result of the programmer’s discipline and sense of design. Our framework requires that the software engineer become familiar with the Objective C language. Implementation in Objective C (or any other object oriented language) requires an approach which appears foreign to developers familiar only with procedural languages. We feel that there are sufficient benefits to this approach. The potential for code reuse provided by the Objective C’s inheritance mechanism should decrease initial development effort, while the additional structure and “firewalls” imposed by the object paradigm should lead to more robust and maintainable software. The results of our initial benchmarks indicate that many of the concerns with the “inefficiency” of object-oriented programming are overstated.

- **Classes for Real-Time Applications**

To support rapid prototyping of real-time applications we require a set of object classes suitable for this area. The suitability of the existing class library remains an open question, and is suggested as an area for further research.

9.2 Further Research

In this section, we identify a number of areas for further research.

Our object-oriented framework has yet to be applied to any useful concurrent system designs. The simple test cases described in this thesis provide what we believe to be a worst case evaluation of the performance penalties incurred by our Objective C support routines. It would be informative to implement a comprehensive real-time system twice: once using the traditional C/Unison approach, and again using our Objective C framework. This should point us to the areas to explore in order to

further reduce the run-time costs associated with our framework. Development of a few practical applications would also serve to identify those classes which would be most useful for real-time systems, leading to the development of a class library for embedded system applications.

Our iconic notation for object-oriented systems presents a somewhat limited view of the structure of object-oriented systems. Each diagram presents a snapshot of the system at some point in time; relatively little temporal information is shown. Object-oriented systems tend to be highly dynamic, with instances created and deleted over the lifetime of the system. A means of connecting a sequence of these snapshots to provide an easily understood description of the temporal behaviour of object-oriented systems would be desirable.

We believe it would be worthwhile integrating our notation for object-oriented design into one of the CASE (computer aided software engineering) tools currently under development within the department. In particular, research in the area of design animation suggests approaches to dealing with the question of describing temporal behaviour. In addition, we can imagine a scheme in which it would be possible to browse through the source code in the class library by navigating through the iconic representation of the class hierarchy.

References

- [1] G.A. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, The MIT Press, Cambridge, MA, 1986.
- [2] P. America, "POOL-T: A Parallel Object-Oriented Language," *Object-Oriented Concurrent Programming*, ed. A. Yonezawa and M. Tokoro, The MIT Press, Cambridge, MA, 1987, pp. 199-220.
- [3] B.N. Bershad, E.D. Lazowska, H.M. Levy, "Presto: A System for Object-Oriented Programming," *Software - Programming and Experience*, vol. 18, no. 8, August 1988, pp. 713-732.
- [4] G. Booch, *Software Components with Ada*, Benjamin/Cummings, Menlo Park, CA, 1984.
- [5] G. Booch, "Object Oriented Development," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 2, February 1986, pp. 211-221.
- [6] G.M. Birtwistle, O-J Dahl, B. Myhrhaug, K. Nygaard, *Simula Begin*, Petrocelli/Charter, New York, NY, 1973.
- [7] R.J.A. Buhr, *Machine Charts for Visual Prototyping in System Design*, Technical Report SCE-88-2, Department of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, Canada, August 1988.
- [8] R.J.A. Buhr, *System Design With Ada*, Prentice-Hall, Englewood Cliffs, NJ, 1984.
- [9] D.R. Cheriton, *The Thoth System: Multi-Process Structuring and Portability*, Elsevier Science, 1982.
- [10] D.R. Cheriton, "Thoth, a Portable Real-Time Operating System," *Communications of the ACM*, vol. 22, no. 2, February 1979, pp 105-115.
- [11] B.J. Cox, *Object-Oriented Programming*, Addison-Wesley, Reading, MA, 1986.

- [12] B.J. Cox, "Message/Object Programming: An Evolutionary Change in Programming Technology," *IEEE Software*, vol. 1, no. 1, January 1984, pp. 50-61.
- [13] F. DeRemer, H.H. Kron, "Programming in the Large Versus Programming in the Small," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 2, June 1976, pp. 80-86. .
- [14] Digitalk, Inc., *Smalltalk/V Tutorial and Programming Handbook*, Los Angeles, CA, 1986.
- [15] T. DeMarco, *Structured Analysis and System Specification*, Yourdon, New York, NY, 1978.
- [16] W.M. Gentleman, *Using the Harmony Operating System*, ERB-966, NRCC No. 27469, National Research Council Canada, Ottawa, Ontario, Canada, December 1983, revised March 1987.
- [17] W.M. Gentleman, "Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept," *Software - Practice and Experience*, vol. 11, no. 5, May 1981, pp. 435-466.
- [18] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, MA, 1983.
- [19] J. Guttag, "Abstract Data Types and the Development of Data Structures," *Communications of the ACM*, vol. 20, no. 6, June 1977, pp. 396-404.
- [20] P. Brinch Hansen, "The Programming Language Concurrent Pascal," *IEEE Transactions on Software Engineering*, vol. SE-1, no. 2, June 1975, pp. 199-207.
- [21] D.J. Hatley, "The Use of Structured Methods in the Development of Large Software-Based Avionics Systems," *Proceedings A.I.A.A./I.E.E.E. 6th Digital Avionics Conference*, Baltimore, MD, December 1984.
- [22] D.H.H. Ingalls, "The Smalltalk-76 Programming System: Design and Implementation," *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pp. 9-16, Tuscon, AZ, January 1978.
- [23] J.C. Kelly, "A Comparison of Four Design Methods for Real-Time Systems," *Proceedings of the 9th International Conference on Software Engineering*, Monterey, CA, April 1987, pp. 238-252.
- [24] B. Kernighan and D. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

- [25] B. Kernighan and R. Pike, *The Unix Programming Environment*, Prentice-Hall, Englewood Cliffs, NJ, 1984.
- [26] H. Liebermann, "Concurrent Object-Oriented Programming in Act 1," *Object-Oriented Concurrent Programming*, ed. A. Yonezawa and M. Tokoro, The MIT Press, Cambridge, MA, 1987, pp. 9-36.
- [27] B. Liskov, S. Zilles, "Specification Techniques for Data Abstractions," *IEEE Transactions on Software Engineering*, vol. SE-1, no. 1, March 1975, pp. 7-19.
- [28] S.A. MacKay, W.M. Gentleman, D.A. Stewart, M. Wein, "Harmony as an Object Oriented Operating System," NRCC 29636 National Research Council Canada, Ottawa, Ontario, Canada, January 1989.
- [29] B. Meyer, "Reusability: The Case for Object Oriented Design," *IEEE Software*, vol. 4, no. 2, March 1987, pp. 50-64.
- [30] Motorola Semiconductor Products Inc., *MC68010 16-Bit Virtual Memory Microprocessor*, ADI-942-R1, Austin, TX, August 1983.
- [31] Multiprocessor Toolsmiths Inc., *Unison User's Guide, Software Release 1.1*, Kanata, Ontario, Canada, November 1987.
- [32] Multiprocessor Toolsmiths Inc., *Integration Toolkit Reference Guide, Software Release 1.1*, Kanata, Ontario, Canada, November 1987.
- [33] Productivity Products International, Inc., *Objective-C Foundation Library, Release 3.3*, Sandy Hook, CT, November 1986.
- [34] Productivity Products International, Inc., *Objective-C Reference Manual, Release 3.3*, Sandy Hook, CT, November 1986.
- [35] D.L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, December 1972, pp. 1053-1058.
- [36] T. Rentsch, "Object Oriented Programming," *SIGPLAN Notices*, vol. 17, no. 9, September 1982, pp. 51-57.
- [37] B. Stroustrup, "What is Object-Oriented Programming?," *IEEE Software*, vol. 5, no. 3, May 1988, pp. 10-20.
- [38] U.S. Department of Defence, *Reference Manual for the Ada Programming Language*, MIL-STD-1815a, 1983.

- [39] IEEE Microprocessor Standards Committee, P1014 Standard Committee, *Standard Specifications For: Versatile Backplane Bus*, Draft Proposed Standard P1014/D1.0, February 1985.
- [40] P.T. Ward and S.J. Mellor, *Structured Development for Real-Time Systems*, vols. 1-3, Yourdon Press, 1985.
- [41] P.T. Ward, "The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 2, February 1986, pp. 198-210.
- [42] P. Wegner, "The Object Oriented Classification Paradigm," *Research Directions in Object Oriented Programming*, ed. B. Shriver and P. Wegner, The MIT Press, Cambridge, MA, 1987.
- [43] N. Wirth, *Programming in Modula-2*, Springer-Verlag, New York, NY, 1982.
- [44] Xerox Learning Research Group, "The Smalltalk-80 System," *Byte*, vol. 6, no. 8, August 1981, pp. 36-48.
- [45] A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda, "Modelling and Programming in an Object-Oriented Concurrent Language, ABCL/1," *Object-Oriented Concurrent Programming*, ed. A. Yonezawa and M. Tokoro, The MIT Press, Cambridge, MA, 1987, pp. 55-89.
- [46] Y. Yokote and M. Tokoro, "Concurrent Programming in ConcurrentSmalltalk," *Object-Oriented Concurrent Programming*, ed. A. Yonezawa and M. Tokoro, The MIT Press, Cambridge, MA, 1987, pp. 129-158.
- [47] E. Yourdon and L.L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice-Hall, Englewood Cliffs, NJ, 1979.

Appendix A

Unison Icon Set

The set of icons used to describe the tasking structure of Unison programs is presented in Figure A.1 . Each icon is described below:

task — a Unison task;

task pool — several Unison tasks instantiated from the same root function;

task creation — implied call to the *create()* function, through which a running task instantiates another task;

library — corresponds to an object file of C functions, or an archive file containing several object files, with one or more externally visible functions. Typically used to depict calls to the Unison standard I/O library, or calls to system functions not associated with the inter-task communication;

library external function — a C function that is visible outside of its object or archive file; specifically, a function that is not declared to be static;

function — a C function;

ISR — an interrupt service routine;

port — a Unison task port, to which messages are sent and through which messages are received.

shared data area — a program variable shared between an ISR and a task;

selective receive — a *receive()* operation with the function's port mask and signal mask fields set to indicate that the task is willing to receive a message through any of several ports, and accept any of several signals;

guard — represents a bit in the port mask used by *receive()*; the bit may be set or cleared to enable or disable message receives via the corresponding port;

message send/receive/reply — a synchronous inter-task message transfer via the *send()*, *receive()*, and *reply()* system calls.

signal — a call to *signal_interrupt()* by an interrupt service routine, in order to asynchronously schedule a task;

interrupt — a hardware interrupt;

Figure A.1: Unison Icon Set: Inter-task Level

Figure A.2 depicts the set of icons used to describe the internal structure of a Unison task. A brief description of each icon follows:

root function — the function that is executed when a task is instantiated, via *create()*.

function — a C function;

library — corresponds to an object file of C functions, or an archive file containing several object files, with one or more externally visible functions. Typically used to depict calls to the Unison standard I/O library, or calls to system functions not associated with the inter-task communication (see Note 1 below);

library external function — a C function that is visible outside of its object or archive file; specifically, a function that is not declared to be static (see Note 1 below);

data area — a static C variable, declared outside the scope of all functions associated with a task;

function call — a call to a C function;

Note 1 – Normally, we do not consider libraries to be part of a the internal structure of a task, preferring to show them as being external to all tasks, as in Figure A.1. However, when designing the internals of a task, it is often useful to show the calls from functions within the task to external library routines, as shown in Figure A.2. Library icons shown in a diagram of the task internals are also shown in the diagrams describing the task level structure.

Figure A.2: Unison Icon Set: Intra-task Level

END

03-03-90

FIN