1

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

**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 Prototyping Environment for Concurrent Systems

by

Jeffrey Thomas Wilson, 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

August 23, 1990

© Jeffrey Thomas Wilson 1990

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.

ISBN 0-315-60486-7

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

A Prototyping Environment for Concurrent Systems

submitted by Jeffrey Thomas Wilson, B.Eng.,
in partial fulfillment of the requirements for
the degree of Master of Engineering

Thesis Supervisor

Chair, Department of Systems and Computer Engineering

Carleton University
September 18, 1990

Abstract

Much software produced using established development methods cannot easily accommodate changes, forcing portions to be rewritten or discarded after installation. The *prototyping* development paradigm alleviates these problems by using an iterative process involving experimentation. To assist in the design of complex systems such as ones employing concurrency, tools must be provided which minimize the cost of experimentation. This thesis describes the conceptual basis for such tools by presenting models for two concurrent system building blocks: (1) time, and (2) interrupts. Timing issues in uniprocessor and distributed concurrent systems are discussed, as are potential solutions to some timing problems. A model for interrupts is given which allows them to be simulated by software. Different interrupt-to-processor configurations, interrupt priority interpretations, interrupt types, and interrupt servicing modes are detailed. Then, a discussion follows about the implementation of two operational tools, *ProtoTool* and *TimeLineTool*, which use these models to form an environment for prototyping concurrent systems. *ProtoTool* gives developers control over the execution of design prototypes and provides feedback about design-related issues. *TimeLineTool* displays graphically the temporal characteristics of a concurrent system design. In combination, they aid in reducing the time required to achieve a good design and help to uncover potential design errors. A typical concurrent system is used to evaluate both tools.

Acknowledgments

My supervisor, Dr. Gerald M. Karam, was a constant source of energy and enthusiasm. He has mastered (PhDed?) the art of stepping in when required while remaining at a distance when appropriate.

What can I say about Mary? She was always willing to listen to my complaints, to share the triumphs, and to comfort me during the down times. Without her encouragement, smiles, and hugs, I wouldn't have accomplished what I did.

I think just about everybody is guilty of taking their family for granted, and I'm no exception. I'd like to take this opportunity to publicly thank my parents and my brother for their support during the past two years. They knew when I needed to talk shop, but could sense when I needed to get away.

Thanks also are due Mary's family for their support. I knew I could count on them for anything.

Two years of work at a university gives one the opportunity to meet many students and professors. Thanks to all the members of the Systems and Computer Engineering Department for their formal and informal feedback. They helped to make the working environment very enjoyable.

If I had a looney for every time I told one of my friends I had to stay at school to get some work done, I could retire tomorrow. Thanks to them for being so understanding.

An often-forgotten group is the examining board. I'd like to thank them for taking time out of their busy schedules to read and critique my thesis.

Even students have to live somewhere, eat, and pay taxes. The financial support of NSERC, BNR, and Carleton University are greatly appreciated.

Contents

1 Introduction	1
1.1 The Softness of Software	1
1.2 Life Cycles and Concurrent Systems	2
1.3 Thesis Objectives	5
1.3.1 Information Gathering and Presentation	5
1.3.2 Developer Control of Prototype Execution	5
1.4 A Research Overview	6
1.4.1 Time Model	6
1.4.2 Interrupt Model	6
1.4.3 <i>ProtoTool</i>	7
1.4.4 <i>TimeLineTool</i>	7
1.5 Thesis Evaluation	8
1.6 Thesis Outline	8
2 Issues in Software Prototyping	10
2.1 Deficiencies of the Waterfall Model	10
2.2 Features of the Prototyping Model	12
2.3 Prototyping, Modeling, and Simulation	14
2.4 Current Prototyping Techniques	16
2.4.1 Very High Level Languages	16
2.4.2 Object-Oriented Programming Systems	18
2.4.3 Specification Languages	18
2.4.4 Algebraic Specifications	20
2.4.5 Special Prototyping Systems	21
2.4.6 Weaknesses of Current Prototyping Techniques	22
2.5 Requirements of an Automated Prototyping System	23
2.5.1 Prototyping Language Requirements	23
2.5.1.1 Computation Model	23
2.5.1.2 Abstraction	24
2.5.1.3 Miscellaneous Requirements	25
2.5.2 Prototyping Environment Requirements	26

2.6 Prototyping Terminology	28
3 An Overview of MLog	29
3.1 OOP Concepts in MLog	29
3.2 Concurrency in MLog	30
3.3 Prolog, OOPS, and MLog	32
3.4 MLog Implementation	32
4 Timing in Concurrent Systems and in <i>ProtoTool</i>	34
4.1 Timing in Concurrent Systems	34
4.1.1 Uniprocessor Systems	35
4.1.2 Distributed Systems	35
4.1.3 Timing Constructs	38
4.1.3.1 Delay Statement	38
4.1.3.2 Timed Receive	38
4.1.3.3 Timed Send	39
4.1.3.4 Conditional Send	40
4.1.3.5 Task Synchronization Without Message Passing	40
4.2 Timing Implementation in <i>ProtoTool</i>	41
4.2.1 Basic Implementation	41
4.2.2 Timed Interaction Facilities	43
4.2.2.1 waitToAccept/6	43
4.2.2.2 waitForInterrupt/6	44
4.2.2.3 waitForInteraction/7	44
4.2.2.4 waitToSend/6	45
4.2.2.5 delay/2	46
4.2.3 Selected Timing Algorithms in <i>ProtoTool</i>	47
4.2.3.1 Timed Interactions	47
4.2.3.2 Scheduling Objects After Timeouts	49
4.2.3.3 Incrementing the Current Time	50
4.2.4 Timing Example	50

5 Interrupts in Concurrent Systems and in <i>ProtoTool</i>	54
5.1 Interrupts in Concurrent Systems	54
5.1.1 Interrupt-to-Processor Connector Configurations	55
5.1.1.1 Single-Line Interrupts	55
5.1.1.2 Multiple-Line Interrupts	56
5.1.1.3 Vectored Interrupts	56
5.1.2 Interrupt Priority Interpretations	57
5.1.3 Interrupt Occurrence Distributions	57
5.1.4 Queuing of Interrupts	58
5.1.4.1 Immediate Interrupts	58
5.1.4.2 Queued Interrupts	58
5.1.4.3 Timed Interrupts	59
5.1.5 Interrupt Tasks	60
5.2 Interrupts in <i>ProtoTool</i>	60
5.2.1 <i>ProtoTool's</i> Interrupt Models	60
5.2.2 <i>ProtoTool's</i> Interrupt Types	62
5.2.2.1 Periodic Interrupts	64
5.2.2.2 Once Interrupts	64
5.2.2.3 Repeat Interrupts	64
5.2.2.4 Ivarread Interrupts	64
5.2.2.5 Ivarwrite Interrupts	65
5.2.3 <i>ProtoTool's</i> Interrupt Handlers	65
5.2.4 <i>ProtoTool's</i> Interrupt Information	67
5.2.5 Interrupt Implementation in <i>ProtoTool</i>	69
5.2.5.1 Queuing in <i>ProtoTool</i>	69
5.2.5.2 <i>ProtoTool's</i> Interrupt Occurrence Algorithm	72
5.2.5.3 Interrupt Example	74

6 ProtoTool	78
6.1 Software Structure	78
6.2 Execution Control	81
6.2.1 Simulation Modes	81
6.2.2 Execution Suspension and Resumption	81
6.2.3 Virtual Processor Configuration	82
6.2.4 Time Lines	83
6.2.5 Interrupts	83
6.3 Information Presentation	84
6.3.1 System and Application Status	84
6.3.2 Display of the Next Object	85
6.3.3 Other Information	86
7 TimeLineTool	88
7.1 Motivation and Overview	88
7.2 Event Generation	89
7.3 The Display	89
7.4 Display Modes	91
7.4.1 Real-Time Mode	91
7.4.2 File Mode	92
7.5 Display Resolution	92
7.6 Display Movement	93
7.7 Colour and Monochrome Displays	93
7.8 Supported Events	94
7.9 Implementation	98
7.9.1 Software Structure	98
7.9.2 Event Data Format	99
7.9.3 Event Data Reading Algorithms	100

8 Evaluation of <i>ProtoTool</i> and <i>TimeLineTool</i> Using The Lift Control Problem	102
8.1 The Lift Control Problem	102
8.2 The Lift Control System Design	103
8.3 Design Examples and Evaluation	106
8.3.1 An Error-Free Design	106
8.3.2 A Design With Interrupt Priority Allocation Errors	109
8.3.3 A Design With Software Deadlock	112
8.3.4 A Design With Interrupt Deadlock	115
9 Suggested Improvements to <i>ProtoTool</i> and to <i>TimeLineTool</i>	120
9.1 <i>ProtoTool</i> Improvements	120
9.1.1 Interrupts	120
9.1.2 Gathered Information	122
9.1.3 Miscellaneous	123
9.2 <i>TimeLineTool</i> Improvements	124
9.3 Other Tools	126
10 Conclusions	128
10.1 A Research Summary	129
10.1.1 Time	129
10.1.2 Interrupts	129
10.1.3 <i>ProtoTool</i>	129
10.1.4 <i>TimeLineTool</i>	130
10.2 Further Research	133
References	134
A An MLog Solution to the Lift Control Problem	140

List of Figures

1.1	The Waterfall Development Paradigm	3
1.2	The Prototyping Development Paradigm	3
2.1	Prototyping's three primary uses	13
3.1	The MLog Rendezvous	31
4.1	A Distributed System with a Network Time Server	35
4.2	A Distributed System with Synchronized Clocks	36
4.3	A Distributed System with Independent Clocks	37
4.4	A Distributed System with Task Migration	38
4.5	MLog Interactions and Timeouts	47
4.6	The Client-Server Timing Example	51
5.1	Single-Line Interrupt Configuration	55
5.2	Multiple-Line Interrupt Configuration	56
5.3	Vectored Interrupt Configuration	56
5.4	Example Processing of an Immediate Interrupt	58
5.5	Example Processing of a Queued Interrupt (Queue Size = 3)	59
5.6	Example Processing of a Timed Interrupt	59
5.7	The Interrupt Class Hierarchy	63
5.8	The Interrupt Handler Class Hierarchy	65
5.9	MLog's and <i>ProtoTool's</i> Queuing Structure	70
5.10	Example Management of the Current Application Queue	71
5.11	The Temperature Gauge Interrupt Example	74
6.1	<i>ProtoTool's</i> Structure	79
6.2	The <i>ProtoTool</i> Display	80
7.1	The <i>TimeLineTool</i> Display	90
7.2	<i>TimeLineTool</i> in Real-Time Mode	91
7.3	<i>TimeLineTool</i> in File Mode	92

7.4	Creation and Deletion Events	94
7.5	Activity Status Events	95
7.6	Synchronous Interaction Events	95
7.7	Anonymous Synchronous Interaction Events	96
7.8	Asynchronous Interaction Events	97
7.9	Interrupt-Specific Events	97
7.10	Miscellaneous Events	98
7.11	<i>TimeLineTool's</i> Software Structure	99
8.1	The Lift Control System Design	104
8.2	The State Machine Agenda of Lift Managers	105
8.3	<i>TimeLineTool</i> In The Error-Free Design	107
8.4	<i>ProtoTool</i> In The Error-Free Design	108
8.5	<i>TimeLineTool</i> In The Design With Interrupt Priority Allocation Errors . . .	110
8.6	<i>ProtoTool</i> In The Design With Interrupt Priority Allocation Errors	111
8.7	<i>TimeLineTool</i> In The Design With Software Deadlock	113
8.8	<i>ProtoTool</i> In The Design With Software Deadlock	114
8.9	<i>TimeLineTool</i> In The Design With Interrupt Deadlock	116
8.10	<i>ProtoTool</i> In The Design With Interrupt Deadlock	117
9.1	An Example of <i>ProtoTool's</i> Timeout Scheduling	123
9.2	A Potential Solution to the Single-Waiting-Line Problem	125

Chapter 1

Introduction

1.1 The Softness of Software

The two components of computer systems, *hardware* and *software*, derive their names from the resistance they exert when modifications are required. Hardware normally provides a rigid appearance, making all but trivial changes difficult. In particular, adding new functions is problematic unless appropriate steps are taken during design. Conversely, software should be pliable, allowing minor and major refinements with relative ease compared to hardware. However, such is not always the case. Many software systems are not soft, forcing significant portions to be rewritten entirely in order to add new capabilities. Three reasons for this are:

- size — some projects are too large to easily accommodate alterations;
- age — older software frequently contains numerous patches, meaning major changes may stretch it beyond its limits;
- requirements — most systems are designed and implemented with a single requirements set in mind and fail to consider potential additions.

In general, a system's hardness is proportional to its size. One way to overcome the problems associated with a large system is to modularize the code as much as possible, creating smaller subsystems. Age also tends to harden software because often modifications are not done within the context of the design and do not anticipate future enhancements. The effects of short-sighted requirements can be lessened by using an improved development paradigm. The challenge presented to us is how to ensure new software is and remains soft.

1.2 Life Cycles and Concurrent Systems

A *life cycle* or *development paradigm*, is the set of phases through which a project progresses from the point it is conceived to the period when it is maintained while operational. Life cycles designed for software development normally contain the following six phases:

- *specification*: determine what the system will do;
- *design*: establish how the specification will be met;
- *coding*: implement the design;
- *testing*: verify the design and the code;
- *delivery*: install the tested system in the target environment;
- *evolution*: validate the delivered system and make necessary changes.

In the development paradigm employed currently by most software developers, the *waterfall model* [Royc70, Boeh76], these phases are distinct, as figure 1.1 illustrates. In particular, each phase is completed before the next is started, and feedback to an earlier phase is considered undesirable, performed only when the output from a phase is incorrect. Thus, erroneous decisions early in the life cycle can reduce greatly the project's quality and can increase significantly its cost. The waterfall model's strictly sequential nature is one of the causes of "hard" software.

Figure 1.2 shows the *prototyping life cycle*, the thrust of which is to uncover errors and inconsistencies early in a project's life. Employing prototyping improves upon the waterfall model by lowering the risk that the output from the critical specification and design phases is incorrect, and adds the additional capability and security of incremental delivery.

Using prototypes can reduce the size of a system by ensuring that only functions deemed necessary by users are implemented. Too often, unclear requirements result in code being delivered which is never used. Fewer specification-related and design-related errors should exist, reducing the number of changes required during evolution. Finally, because users are given an opportunity to validate prototypes, there is a higher probability that the requirements implemented are those desired. Thus, prototyping can lead to "softer" software than that produced using the waterfall model.

Figure 1.1 The Waterfall Development Paradigm

Figure 1.2 The Prototyping Development Paradigm

Concurrent systems, because of their inherent complexity, are frequently more difficult to specify and design than sequential ones. Concurrent systems are characterised by:

- concurrent execution of *tasks*¹ — *virtual concurrency* if using one processor or *real concurrency* if using multiple processors;
- intertask communication — *synchronous communication* using mechanisms such as the Ada rendezvous [ALRM83] or *asynchronous communication* using semaphores, mailboxes, etc.;
- task scheduling — the rules used to determine when tasks can execute;
- dependence on time — tasks frequently base their operation on timing parameters, such as when delaying or when performing timed intertask communications;
- interaction via interrupts — concurrent systems often interact with their environment using interrupts.

Once the requirements are established, a developer must choose a design from a potentially large set of possibilities, trading off factors such as performance and portability. For instance, a choice may have to be made between synchronous and asynchronous communication. Usually decisions are made based on previous experience with the domain in question. Although using experience is desirable and necessary, a better selection method involves evaluating as many designs as possible and then choosing the best considering observed results.

This is infeasible using the waterfall life cycle because once a design is chosen, it is coded and tested completely before any operational evaluation is done. Iterating this process would be too slow and too costly.

Prototyping, with its built-in iteration, is ideal for evaluating multiple designs. However, this is not sufficient. Developers must be provided tools and techniques which enable prototypes to be constructed and evaluated quickly and cheaply. Many benefits of prototyping would be lost if building and running prototypes was too time consuming or expensive.

¹ The term *task* refers to the unit of concurrency in the language/system in question.

1.3 Thesis Objectives

The first objective of this thesis is to survey the software development literature to determine the current state of prototyping, with particular interest in the tools and the techniques used to perform prototyping. By summarizing and analyzing prototypers' experiences with different languages and systems, features considered beneficial for and detrimental to prototyping can be identified.

A second objective is to identify the requirements for a complete prototyping system. Establishing these requirements will result in a base line from which operational systems can be developed. The requirements include those of a prototyping language in which to express program semantics, and those of a prototyping environment to allow program execution, manipulation, and observation.

The third and primary objective is to develop and analyze *ProtoTool*, an environment for prototyping concurrent systems. Its implementation is to be integrated with the MLog multiparadigm programming language [Kara89a] because of MLog's suitability for prototyping, but the underlying concepts should be implementation-independent. *ProtoTool* should give developers fast and informative feedback about potential designs using two perspectives: (1) information gathering and presentation, and (2) developer control of prototype execution.

1.3.1 Information Gathering and Presentation

Information pertinent to the execution of concurrent systems must be obtained automatically and displayed using a syntactically-shallow, semantically-deep graphical interface. Intertask communication should be the main emphasis because this is normally the most troublesome part of concurrent system design. Additional information of interest must be made available, such as the state of interrupts and the characteristics of the virtual processor (speed, task scheduling policy, etc.).

1.3.2 Developer Control of Prototype Execution

The developer must be given the opportunity to answer "What if?" questions about the design by being able to modify characteristics of the execution environment. Thus, objectives of *ProtoTool* are to identify what the developer should be able to control and

to provide the capability to do so. For instance, the developer should be able to change the task scheduling policy of the virtual processor and should be able to manipulate interrupts. In order to allow control of entities such as interrupts, models of these entities must be developed which are consistent with their target environment counterparts.

1.4 A Research Overview

In response to these thesis objectives, the following are the key developments arising from our research:

- a model for time which allows timed interaction facilities to be constructed;
- a model for interrupts which enables interrupts to be simulated in software;
- an operational prototyping environment, *ProtoTool*, which gives software developers execution control and feedback during the evaluation of concurrent system designs;
- a time line display tool, *TimeLineTool*, for graphically illustrating the temporal order of concurrent system events.

1.4.1 Time Model

Time is modeled as a system's progression of activity. It does not have to be measured in terms of the "real world" time. A clock is a means of associating a number with an event such that the number is considered the time when the event occurred. Thus, in concurrent systems which use message passing, it is possible to consider each message a clock tick.

1.4.2 Interrupt Model

Interrupts are modeled as entities which autonomously gain control of the processor. The following interrupt types are supported: *periodic* and *once*, which occur at a specified set of times, *repeat*, which occur at a specified set of times in a repeating pattern, and *ivar*, which occur when a specified MLog instance variable is read or written². When an interrupt occurs, it is processed according to its queuing mode: *immediate*, *queued*, or *timed*. Immediate interrupts are missed if they cannot be processed immediately. Queued interrupts are queued until they can be handled, up to a specified limit, after which they

²Ivar interrupts allow examination of the effects of interrupts during data reads or writes.

are discarded. Timed interrupts can remain pending for a specified period, after which they are lost.

1.4.3 *ProtoTool*

ProtoTool is integrated with MLog's execution engine and takes advantage of MLog's windowing capabilities to provide an efficient user interface.

ProtoTool gives developers the ability to modify parameters associated with MLog's virtual processor such as its speed, its task-scheduling and interrupt-scheduling policies, and the current time. Evaluation can be halted at any time to allow examination of design characteristics. Two execution modes are provided: *free running* mode, where the system time is advanced automatically to the time at which the next timeout will occur if no active objects³ are able to run but timeouts are pending, and *interactive* mode, where control is returned to the developer if no active objects are able to run but timeouts are pending.

Information pertaining to the MLog engine, such as its speed and current time are displayed continuously. When control is returned to the developer, the next active object to run or event to occur is shown along with any other pertinent data. For instance, if the next event to occur is an interrupt, an indication is given if the interrupt can be processed, and if not, why not. Details about an individual interrupt can be shown, such as the times when it was processed, the times when it was missed and why, if and how many are pending, and if any active objects are waiting for it to occur.

ProtoTool presents to the developer many interrupt manipulation functions, such as the ability to enable or disable interrupts globally or individually, the ability to generate a selected interrupt, and the ability to change an interrupt's priority. Interrupt service routines are defined by the developer.

1.4.4 *TimeLineTool*

TimeLineTool shows, either in *real time* mode as execution proceeds, or in *record* mode later as a replay, the interactions among tasks and interrupts. The developer can move through the display at will, changing the resolution if desired. Synchronous and asynchronous interactions, creation and deletion of tasks, and interrupt occurrences are

³ Mlog active objects are like Ada tasks.

the types of data displayed graphically. When running on a monochrome monitor, the width and texture of the lines contain the semantics. For example, a solid, thick line represents the task currently running, while a dashed, thin line indicates the associated task is blocked. If colour is available, the semantics are colour-dependent. For example, green indicates the task currently running, yellow indicates a task which is ready to run, and red indicates a task which is blocked.

The concept behind *TimeLineTool*'s approach to temporal behaviour display comes from Buhr [Buhr90]⁴.

1.5 Thesis Evaluation

The results of this thesis will be evaluated using a lift⁵ control system as the example. This problem contains all the elements of concurrent systems: (1) concurrently-executing tasks, (2) timed intertask communication, and (3) environment interaction via interrupts. It will be shown that the capabilities of *ProtoTool* and *TimeLineTool* aid in the development of a high-quality design. For example, *TimeLineTool* will be used to illustrate deadlock⁶, and *ProtoTool* will be used to establish a good interrupt priority scheme.

1.6 Thesis Outline

The following chapter discusses more features of prototyping and presents the results of a survey of current prototyping techniques. Chapter three gives a brief overview of MLog. Chapters four and five, respectively, describe the time and interrupt models implemented by *ProtoTool*. Included in chapter four are timing issues in distributed systems. Chapter six discusses *ProtoTool* itself, including the information gathered and presented, its user interface, and the techniques employed to enable execution control. The next chapter describes *TimeLineTool* in detail. Chapter eight describes the solution to the lift control problem, and evaluates *ProtoTool* and *TimeLineTool* based on this example. Chapter

⁴The notation used throughout this thesis for illustrating concurrent system designs is taken from the same reference.

⁵The problem definition was generated by a British firm, hence the use of the term *lift* instead of *elevator*.

⁶Deadlock occurs when two or more tasks wait for each other in a circular manner.

nine identifies suggested improvements to *ProtoTool* and *TimeLineTool*, and defines other tools which could be integrated with *ProtoTool* to create a more sophisticated prototyping package. The last chapter gives conclusions and provides the motivation for further research. The MLog code for the solution to the lift control system example is included in an appendix.

Chapter 2

Issues in Software Prototyping

2.1 Deficiencies of the Waterfall Model

The *waterfall development paradigm*, shown in figure 1.1, is employed liberally in the software industry. In most cases, it meets its primary objective of making the engineering of software orderly and manageable by delineating clearly the development phases and their deliverables. By providing a framework in which to operate, the progress of a project can be tracked and problems can be identified. The expense of this manageability is decreased flexibility as developers are required to build systems in a constrained, sequential manner. Two notable characteristics of this paradigm are:

- no operational user validation is done until the entire system has been specified, designed, coded, tested, and delivered;
- there is no feedback to what is considered the most critical phase of a project, the specification [Jado89, Boar85(pp.9)].

Changes to a system become more costly as work proceeds on it, in roughly an exponential fashion, so that coding, testing, and evolution contribute most to the overall cost⁷, with evolution alone commonly absorbing fifty to ninety percent [Hekm88(pp.5)]. Hence, to avoid futile and costly development, the outputs from the specification and design phases must be correct.

Much software developed using the waterfall model is either not accepted by users or requires many revisions once delivered. This has lead to criticism of the model [Glad82, Balz86, McCr86, Swar86, Gidd86, Fish87(pp.1-3)], and has revealed three assumptions embedded in it:

- developers, during specification, interpret the requirements as users intend;

⁷Delivery normally incurs only a small cost.

- users estimate the impact of installation and only request systems which will integrate easily;
- all requirements are established before development starts.

A shortcoming of the waterfall paradigm is that any undesirable effects of these assumptions remain hidden until after the costly life cycle phases. Thus, there exists some risk that much time and effort will be lost if the system is not satisfactory to users. It assumes that users and developers share a similar view of the system, which is frequently untrue. One cause of this discrepancy is the way the specification phase is performed. It usually consists of manually analyzing the requirements, generating a written specification statement, and obtaining informal user validation. A written specification is accessible to anybody who understands the language, but because it cannot be operationally validated, an assumption is made that it is a final, complete, correct, and consistent representation of the requirements [Boar85(pp.10)]. Often this is not the case because:

- the specification statement is normally written for the design group, so users find it difficult to assimilate;
- users cannot fully appreciate how a system will operate by reading a document.

The following quote⁸ illustrates these problems:

"In virtually every project I have done, I did not have all the information needed. I have had to go ask questions, make judgements about what I thought the users wanted, and had to go back and modify the software to do what the users really wanted (but couldn't coherently state)."

In the design phase, developers must select a design from a potentially large set of possibilities without the aid of any operational feedback⁹. Further, because the coding phase is completed before any testing is done, design errors may cause code to be discarded. This is particularly important for concurrent systems where different designs can result in very dissimilar code.

⁸ This quote is by Jerome Vollborn and was taken from an electronic news system, November, 1989.

⁹ Tools which execute designs are just beginning to appear in the research community.

2.2 Features of the Prototyping Model

The *prototyping* development paradigm [RPWS82, Agre86] is designed to quickly and inexpensively remove doubt about a system's requirements, feasibility, and design. It advocates three notions not present in the waterfall model:

- planned, desirable iteration, occurring when new, more accurate information is obtained;
- production of executable versions of systems, called *prototypes*, for intermediate verification and validation;
- user involvement early in the development process.

Prototyping provides a means of increasing the probability that the output of the specification phase are those requirements desired by users and that they define a feasible system. It also ensures that the design correctly implements the requirements and is sound, and that the output of the coding phase faithfully reproduces the system's requirements and design.

The waterfall model forces a design to be coded using production standards and tested before obtaining any operational feedback, making this process expensive and lengthy if errors occur. Prototyping allows many designs to be evaluated quickly and cheaply before submitting one for coding. Whereas the waterfall paradigm implies that the entire system is coded and tested before any delivery takes place, prototyping espouses a scheme which enables some code to be written, tested, delivered, and validated before committing to the remainder.

Prototyping reduces development risks by ensuring a high quality output from each life cycle phase. A prototype is constructed immediately after an initial understanding of the problem is acquired, and is then executed with the intent of finding errors, omissions, contradictions, and ambiguities. When such problems are found, the prototype is modified and executed again. This information acquisition and learning process continues iteratively until a satisfactory result is obtained. The prototype is then thrown away, used directly as the input to the next phase, used as the input to a transformation mechanism, or used as the initial implementation of the next phase.

The prototyping model allows greater flexibility than the waterfall model. For example, if the requirements are clear but the production environment is novel or

unfamiliar, a method using a conventional specification approach but prototyping in the design and delivery phases is appropriate. If the requirements are unclear but the system has a simple design and little code, only requirements prototyping may be necessary. A separate prototyping scheme for each major portion of the life cycle allows a hybrid development model to be constructed.

Figure 2.1 shows the areas where prototyping is most useful.

Prototyping aids requirements analysis during the specification and delivery life cycle phases. By executing and validating requirements prototypes in the specification phase, users determine their actual requirements and ascertain how well the system will fit into their environment. Developers have a chance to become more familiar with the system and can learn how users wish to interact with it. The major concern of requirements prototypes is the behaviour of the system, not its eventual software structure. Delivery prototypes are code fragments which implement partial functions and are used to find requirements errors and design errors not uncovered earlier. These prototypes allow users to work “hands on” in the production environment before the complete system is installed, and must deal with the structure of the software as well as the behaviour because they implement pieces of the final system.

Figure 2.1 Prototyping's three primary uses

Designing a system is a creative process, one in which developers are encouraged to try alternate designs and compare them. Prototyping tools and techniques which allow information about a design to be gathered and analyzed make this time consuming process easier and give developers an opportunity to experiment with more possibilities before

constraints force a decision to be made. Design prototypes focus on both the behaviour and structure of a system.

Prototyping assists project management by helping to determine if development of a system is feasible. Even if the requirements are interpreted correctly and the design is sound, the system may be infeasible for reasons such as size, development time, and complexity. Using the waterfall model, feasibility may be determined only after much effort has been expended, especially if hidden details emerge in the design and coding phases. With prototyping, feasibility is determined much earlier because of the need to get at least parts of the system functioning quickly. An experienced developer will be able to convey to management the feasibility of a project very early in its life cycle using both behavioural and structural information.

It is important to recognize that prototyping does not replace requirements analysis and design, nor does it turn system development into a trial and error process. Prototypes should be used strictly as verification and validation vehicles. Thus, requirements analysis and design should be approached with the same enthusiasm as when using any other development model.

2.3 Prototyping, Modeling, and Simulation

An accepted definition of a model, in a systems engineering context, is an abstraction and simplification of a system for the purpose of studying it. Models exist in two basic forms [Bank84(pp.10)]: (1) physical models, and (2) mathematical models. The process of modeling involves deriving a system's behavioural characteristics and mapping them into a model. For physical models, the mapping is into an entity which could be built and experimented with in a "hands on" way. For mathematical models, equations are developed which embody the behaviour, and are then solved to simulate the system's operation. A model need not resemble its associated system in appearance as only the behaviour must be the same.

Prototyping involves creating versions of systems which either do not satisfy all the functional requirements or disregard selected non-functional requirements. The result is scaled-down versions of systems rather than abstractions. As a result, prototyping and modeling are slightly different. As opposed to having a conceptual notion of the system

in mind, prototypers work from an image of the real system, never losing sight of the eventual implementation. The exception is requirements prototypes. They are used only to clarify requirements, so they can be completely implementation-independent.

Simulation is a problem solving technique involving the design of a system model and experimentation with it for the purpose of learning about the system. Mathematical models are more popular than physical ones because the computational power of computers has made solving equations an easier, faster endeavour.

At the conceptual level, simulation and prototyping are quite similar, both being methods for probing a system to gather information about it before making commitments. Fishman [Fish78(pp.2)] identified eight ways in which a model can be used within a simulation context:

- to organize theoretical beliefs and empirical observations, and to deduce the logical implications of this organization;
- to improve system understanding;
- to bring into perspective the need for detail and relevance;
- to expedite analysis;
- to provide a framework for testing system modifications;
- to enable easy manipulation of the system;
- to permit control of the system's environment;
- to inexpensively experiment with the system.

Each of these points also applies to prototyping, showing that the fundamental basis of both is the acquisition of information by building an entity which describes the operation of the system. Both allow for iterative refinement as knowledge is gained, and both provide the ability to reproduce execution scenarios. Each allows investigation of "*What if?*" questions about the system [Bank84(pp.2)].

In terms of when to apply them and how to accomplish them, simulation and prototyping are somewhat different. Prototyping is used to build scaled-down versions of systems, whereas simulation is often applied when such is not possible¹⁰. Simulation is appropriate for large systems where mathematical modeling is the only feasible approach. Simulation is also used when it is not practical or convenient to build the real system. For

¹⁰For example, when building a new manufacturing facility or examining the economy.

example, interrupts are simulated by *ProtoTool*¹¹ because such is easier than physically interrupting the machine on which MLog is running. This example illustrates how simulation and prototyping can be used together to accomplish a goal.

Simulation may be preferable if analysis is required because analyzing a formal mathematical model is generally easier than analyzing an informal representation such as a prototype. Prototyping is superior for design because of the closeness of prototypes to the actual system. Prototyping is more proactive because it starts with scaled-down versions, allowing room for construction. Simulation is more reactive, working best when starting from an existing system from which to extract an operational model.

Perhaps the factor which distinguishes prototyping from simulation the most is the degree of abstraction. Simulation models can be very abstract with respect to the system they represent. Banks' use of the term *artificial* when describing simulation implies a high degree of abstraction [Bank84(pp.2)]. Prototypes are not abstracted to the same degree. This is particularly true of design prototypes which must consider many issues associated with the eventual target implementation.

2.4 Current Prototyping Techniques

This section describes the following five techniques currently used for prototyping, and gives examples of each:

- very high level languages;
- object-oriented programming systems;
- specification languages;
- algebraic specifications;
- special prototyping systems.

2.4.1 Very High Level Languages

Very high level languages are characterised by a short, concise notation, and a high expressiveness. It is possible to construct programs in a shorter time and with a reduced

¹¹See chapter five for details about interrupts in *ProtoTool*

effort compared to lower level languages. This makes prototyping feasible by minimizing its cost as prototypes can be discarded at only a small financial loss.

Prolog is recognized as a language which can be used for prototyping [Dwig83, Davi82, Komo86, Bruy84, Lieb84]. Its declarative style and implicit control mechanism make rapid development possible. Because they are executable and formal, prototypes written in Prolog should be verifiable and transformable. In addition, most Prolog environments contain good debugging facilities which reduces development time. Prolog is incorporated into MLog [Kara89a], which is designed to be a rapid prototyping language combining the object-oriented, logic programming, and concurrent system paradigms.

Lisp has been used for prototyping, primarily because of its reputation of delivering high productivity. Some programming environments have been developed for it, for example Microscope [Ambr88], making its use even more attractive. A potential disadvantage is the need to express a large amount of detail, a factor which may retard quick coding.

SETL [Kruc84] is a programming language based on set theory which contains some powerful constructs to express algorithms succinctly and easily. The descriptive power of set theory allows complex algorithms to be developed quickly and to be specified at an abstract level if desired. Thus, SETL can be used for simulation as well as prototyping.

Snobol [Hock85], a language designed to facilitate character string manipulation, is appropriate for prototyping text-oriented systems such as language processors.

Another very high level language that has been advocated for prototyping is APL [Tavo84]. It contains some powerful operations and provides flexible filing systems and a report formatting facility which make it suitable for prototyping data processing applications.

Application-oriented languages [Brow88], ones targeted at a particular domain, can be used for prototyping because they free the developer from coding concepts which are inherent to the domain by providing guidance about appropriate solutions. Application generators can be used with application-oriented languages to automatically or semi-automatically produce application code from abstract specifications.

The Unix¹² shell language [Kern84(chps.3,5)] can be used to prototype because it contains concise, high level constructs which are useful for experimenting with ideas before committing to them. When combined with the extensive array of Unix tools, a good prototyping environment exists.

2.4.2 Object-Oriented Programming Systems

Object-oriented programming is appealing for prototyping because it promotes code reuse. If an object library grows large enough, developing prototypes may become a process of connecting existing objects in such a way that they meet a behavioural specification. In addition, many problem domains are well suited for object-oriented solutions, a fact which should help minimize ambiguous requirements.

Smalltalk [Gold84] is the most popular object-oriented system. Some feel it provides a good prototyping environment because of its simple graphical interface and its use of object-oriented concepts [Love83]. It is particularly good at developing user interfaces because of its assortment of windowing components. Even those who feel Smalltalk has some deficiencies agree that, once learned, it is a very productive tool [Niel89].

The Trellis programming environment [OBri87] is designed to support composition of systems in an object-oriented framework. Some of its tools are applicable to prototyping, including a cross reference tool which shows what code is affected when a change is made, and a *grass catcher* tool which gives the user a list of code which must be changed when a compilation fails. It contains a tool to allow browsing through the object hierarchy in a tree-like fashion, helping the developer determine how the system is constructed.

Graphtrace [Kley88] is similar to Trellis in that it assists developers in understanding a system's composition. This is done using a graphical tool which shows how the system's objects are connected.

2.4.3 Specification Languages

The basis for this technique is a formal, executable specification language which abstracts behaviour from implementation. The intent is to emulate the behaviour of a system rather than to implement it, so the structure of the specification need not resemble that of the

¹²Unix is a trademark of AT&T Bell Laboratories.

eventual implementation. Executable specifications are used for the same purpose as requirements prototypes, namely to obtain user validation early in the life cycle.

One such language is PAISley [Zave86], which merges functional programming [Hend86] and asynchronous processes. It is aimed at specifying real time, concurrent systems. PAISley and its environment contain several features desirable for prototyping:

- support for modular specifications, allowing multiple developers to build them;
- the ability to execute incomplete specifications;
- facilities to call out to existing code and have external code call in to specifications.

The ENCOMPASS environment [Terw87], which uses the PLEASE specification language [Terw86], is similar to PAISley in that it partially automates formal software development methods, but it also contains transformation mechanisms for generating implementation code from PLEASE specifications. It also:

- has facilities to store, track, manipulate, and control all objects used in the development process, including documents, specifications, code, proofs, and test data;
- contains mechanisms to support interactions among developers such as creation, decomposition, distribution, monitoring, and completion of development tasks;
- has project management and software reuse tools.

Thus, as well as addressing the technical aspects of prototyping, ENCOMPASS deals with the management of the prototyping process, a critical issue considering the flexibility of the prototyping paradigm.

ESL [DeFr88] is an event-based specification language for ECSP programs, where ECSP is an implementation language whose constructs are an extension of those defined by CSP [Hoar78]. ESL uses a constructive approach, specifying how a program must function by defining a set of possible behaviours. Despite the fact that ESL programs can be transformed into ECSP programs, ESL has two flaws which hinder its use for prototyping:

- specifications must be complete before they can be executed;
- dynamic loading is not supported.

PROT nets [Brun86] use a Petri net-based graphical language to specify concurrency and synchronization of concurrent entities. The PROT nets are specified through a graphical net editor and are then translated into a program skeleton to be used as the basis for a prototype. The advantage of using a well known formalism such as Petri nets is that the environment's learning curve will be small, making it widely accessible.

A similar tool is SPECS [Dahl87] which is based on Petri nets, but is extended with an object oriented, hierarchical structuring facility. The Petri nets specify the structure and control flow, while Smalltalk is used to describe data structures and sequential behaviour. Thus, the benefits of graphical specification are combined with the possibility of code reuse, making this an attractive prototyping environment.

STATEMATE [Hare88] is a tool which uses a graphical notation called *state charts* and an underlying event language and interpreter to allow validating the requirements of complex concurrent systems. A feature attractive to prototyping is its ability to generate skeleton implementation code from a specification.

2.4.4 Algebraic Specifications

Algebraic specifications are similar to executable specifications except that, unlike the latter, they are based on a rigorous mathematical framework which allows formal reasoning. It is normally easier to translate algebraic specifications because of their formal underpinnings.

ADTS [Jado89] and its associated environment support rapid prototyping using algebraic specification by providing automatic translation to high level language code. The following features make it useful for prototyping:

- modular specifications, allowing reuse and parallel development;
- a transformation mechanism which detects some specification errors, thus reducing the number of user validation iterations;
- the ability to embed calls to and from a specification in high level language code so that the specification can be used as part of a larger system.

Autostar [Zhu89] is a software development system which accepts algebraic specifications and produces a corresponding Ada implementation. The formal specification is automatically transformed into an intermediate representation called ADTADA which

is similar to Ada in syntax. Then, using user assistance when necessary, the ADTADA code is transformed into Ada.

2.4.5 Special Prototyping Systems

The first step in using existing code or debugging new code is obtaining an understanding of how the code is structured and how it behaves. As a result, tools which assist developers in understanding software can increase productivity and help to make prototyping feasible.

Visiprog [Hend85] allows developers to learn about a system in a dynamic context. A program executes continuously and is viewed through an input window, a program window, and an output window. When an input is changed, new outputs are generated. When the program is changed, new outputs are produced using the existing inputs. In certain cases, Visiprog can derive the input associated with an output. Using Visiprog, a developer can gain quickly an understanding of a system's behaviour.

Microscope [Ambr88], an integrated program analysis tool set, supports evolutionary software development by helping developers understand programs written in Lisp. It contains both static and dynamic analysis capabilities, and uses graphical displays to convey information. Sophisticated monitoring facilities provide program control, while a history browser allows the developer to see what calls were made during an execution and in what order.

The tools and environments discussed thus far are useful for prototyping, but were not designed with prototyping as the primary goal.

Veda [Jard88] is a prototyping tool for distributed algorithms, designed for protocol verification¹³. The Estelle programming language, whose similarity to Pascal makes it attractive to developers, is used to specify the distributed algorithms. Two features make Veda particularly good for prototyping:

- the ability to control time and the external environment;
- a sophisticated set of information gathering tools, called *observers*, which enable acquisition of program data.

¹³It can be used for non-protocol applications also.

Luqi [Luqi88a, Luqi88b, Luqi89] describes a prototyping system which uses modularity and component reuse for prototyping large real-time systems. It contains a combined textual and graphical language using nodes to represent processing functions, connections to represent call paths, hierarchical decomposition to represent modules, and text to specify execution details. The graphical approach makes it easy to see how the system is assembled, and enables large parts of it to be visible on a workstation screen. Any specified real time constraints are automatically checked. An attempt was made to automate the component retrieval function by using an artificial intelligence based tool employing various search strategies [Luqi88c]. Initial observations are promising.

TimeBench [Kara89b] is Carleton University's successor to its CAEDE [Buhr85] project. TimeBench allows concurrent system designs to be entered graphically using Buhr's machine charts notation [Buhr90], and automatically generates the code, in a special machine charts language called MCL, which specifies how the concurrent components interact with each other. The remainder, containing algorithmic details, is specified by developers using MCL. MCL contains many features for easing the coding burden on developers, such as lists which automatically expand and contract as needed. An MCL interpreter executes the resulting program, and provides visual feedback by animating the machine charts entered by the developer.

2.4.6 Weaknesses of Current Prototyping Techniques

The prototyping techniques described above are effective, but they leave room for improvement. For instance, very high level languages allow fast coding, but few are accompanied by a prototyping environment which takes advantage of their capabilities. Thus, developers are given little guidance in how to use the language properly.

Specification languages and algebraic specifications are appropriate for requirements prototyping, but they are unsuitable for design prototyping because of their abstraction from implementation. Design prototyping is essential for concurrent systems, so a single prototyping system which allows requirements prototyping and design prototyping would be beneficial.

The special prototyping systems discussed are geared towards assisting developers in understanding programs by observing static relationships and dynamic behaviour. This information gathering falls short in terms of providing developers of concurrent systems

with the detailed data they require to make competent decisions. In addition, few facilities exist for modifying prototypes and the underlying run-time system to experiment with alternative configurations.

2.5 Requirements of an Automated Prototyping System

In order to increase productivity, developers require an automated prototyping system containing all the facilities necessary to quickly and inexpensively construct and evaluate prototypes. Two components comprise such a system: (1) a prototyping language, and (2) a prototyping environment. This section discusses the requirements of each of these components, most of which have been taken from [DRAF89]. This reference represents common views of what capabilities a prototyping system should include.

2.5.1 Prototyping Language Requirements

A language is used to specify the contents and semantics of code units. The requirements are discussed in terms of the language's computation model, its support of abstractions, and miscellaneous considerations.

2.5.1.1 Computation Model

A language's computation model defines the way its programs are interpreted and executed, which in turn affects the way solutions are developed. There exist many computation models, including object-oriented, imperative, declarative, pure logic, abstract data type, equational and term rewriting, and artificial intelligence.

The object-oriented model [Booc86] is appealing because of the ability to reuse code. By distributing functions into small components with well-defined structural and behavioural interfaces, programs can be assembled using existing software, thus reducing development time and cost. A prototyping language should try incorporate the object-oriented paradigm with its other computation models.

The imperative model requires instruction sequences to be specified explicitly at compile time, as in Ada. This is undesirable for prototyping where such sequencing information may not be known until some operational feedback is acquired. In contrast, the declarative model employed by languages such as Prolog requires only relationships

to be defined at compile time, allowing run time data to determine the operational characteristics. This facilitates prototyping by reducing the amount of detailed knowledge necessary to develop working prototypes. A good prototyping language would have both declarative and imperative capabilities.

2.5.1.2 Abstraction

An abstraction represents a common pattern and provides a means of specifying which variation to use, thus assisting rapid construction by promoting reuse. It should be possible to abstract any programming entity, including modules and types, and all entities must be first class¹⁴. In addition, abstractions must be representation independent, meaning any change to an abstraction's implementation must not affect its users.

In an attempt to maintain integrity, values of abstract data types should be accessible only via legal operations. However, an override mechanism should exist so that they can be manipulated directly when necessary.

Both the traditional procedural/functional abstraction, which enables an expression to be parameterized by a value, and the newer object-oriented abstraction, which is the abstraction of the behaviour of an object over a value, should be provided and all the advantages of each must be available. The former centralizes functions while the latter distributes them.

To benefit from reusability, the language must provide a means for aggregating abstractions into larger groups. Because the resulting modules are themselves an abstraction of the contents they encapsulate, a module specification language may be required to describe modules for the purpose of integrating them.

The prototyping language should have a rich type system where a type is regarded as an abstraction of the behaviour of data. Many [DRAF89] feel strong typing is best, ensuring data is used only in ways consistent with its type. A disadvantage of strong typing is the need to know at compile time what type procedure parameters must be, making them less generic. Some, including Karam [Kara89a], feel that strong typing would prove to be too restrictive for rapid prototyping. Freeing developers from the handcuffs of type checking should make prototype development faster and easier.

¹⁴First class entities are ones which can be manipulated as if they are data.

2.5.1.3 Miscellaneous Requirements

It should be possible to express concurrency because of the increased interest in concurrent systems, and the granularity of the concurrent entities should match that of the targeted production environment. For example, if Ada is the intended production environment, the concurrent entities should be large grain. In addition, the expression of concurrency should be explicit to facilitate code comprehension.

In the same vein, non-determinism should be supported. The non-deterministic parts of code should be expressed explicitly so that their implementation becomes identifiable and tractable.

The language should have a single, well-defined concrete syntax¹⁵ for two reasons: (1) some tools deal directly with the concrete syntax and requiring them to be parameterized would make them unnecessarily complicated, and (2) a common form allows code to be exchanged and understood more easily, thereby promoting reuse.

There should be a single, well-defined abstract syntax¹⁶. By standardizing an abstract syntax, tools such as compilers will be interchangeable and portable across implementations.

To obtain the most benefit from its development, the scope of the language should be as wide as possible, remembering that a generic language may not contain structures detailed enough to correctly or efficiently implement some applications. However, a specific language may not be feasible because of its small applicability.

It is desirable for the prototyping language to be defined as formally as possible in order to guarantee correctness if prototypes are to act as executable specifications or are to evolve into the production system. However, this formality should not preclude writing incomplete requirements prototypes or design prototypes which are used only as learning vehicles.

Efficiency of prototype execution cannot be ignored completely because if running a prototype takes excessively long, the prototyping system will not be used. A prototype should not run more than one order of magnitude slower than the corresponding production system.

¹⁵The concrete syntax is the way the code appears in source files.

¹⁶The abstract syntax is the syntactic language definition.

If the computation model is significantly different than that of the production environment, developers may find it too difficult to use, forcing the use of a separate prototyping group. Hence, the prototyping language may have to be tailored to a particular production environment.

In order to make developer and code transition from the prototype environment to the production environment easier, there should be a natural mapping from the prototyping language to the production language in the structural and behavioural domains. This requirement is unnecessary if prototype to production environment transformation mechanisms exist, or if separate prototype and development groups are used.

2.5.2 Prototyping Environment Requirements

The environment acts as the glue which binds the system together, providing facilities for developers to build, execute, evaluate, and modify prototypes. The environment should allow quick code entry and modification, and should promote easy code understanding.

The environment's architecture must be open to allow adaptability. That is, it must be portable, customizable, tunable, and extensible [Ridd86]. It must present visible and coherent input and output interfaces, enabling other vendors or even users to incorporate additional tools. This is especially important for prototyping where freedom in the development model may be translated into different tools for different developers.

The environment must be integrated, that is, there must be a high degree of homogeneity among its facilities [Ridd86]. In particular, the system's data should be represented by a single view and all tools must make use of this view, resulting in the ability to move easily between tools because of a consistent underlying data structure.

The environment must provide testing tools. Developers must be able to specify a test scenario and run the test according to that scenario. Automated means should exist for determining if a test was successful, and if so, the developer should be given enough information to determine the reason for the success. Developers should be able to specify invalid situations and should be notified if such situations occur. Automated test tools are crucial to prototyping where it is vital to ascertain quickly if a prototype meets its behavioural specification.

For graphics-related applications, a tool kit which allows quick and easy assembly of complex window-based user interfaces is desirable. This kit must contain standard components and should permit the addition of user-defined ones. Building user interfaces is normally a slow process which hinders rapid development.

A screen-based, syntax-directed editor is required to expedite code entry. A graphical editor could be considered if the language is syntactically heavy, but caution should be used because graphical tools sometimes deliver less than they promise [Mart88].

Another necessity is a source level debugger incorporating state of the art functions with respect to the language. For example, a debugger for a concurrent language should allow monitoring the ready-to-run queue.

The developer needs a set of system definition tools to help construct systems from existing code by providing configuration management facilities, insight into how to best incorporate code changes into existing systems, and assistance in managing the multiple versions of modules generated during the prototyping process.

A dynamic loading facility is required which allows single units of code to be changed, compiled, and replaced in an executing system without requiring recompilation and reloading of the remainder of the system. This capability is important for user validation sessions where immediate feedback from changes is needed.

To reduce coding times, organizations are partitioning work into parallel development. Hence, a concurrency control mechanism is required which permits simultaneous access to shared resources. This is very important for prototyping where reduced coding times can result in development and evaluation of more prototypes, resulting in a better product.

A set of information presentation facilities must exist which can display all retrieved information in a pleasing form. This is necessary for prototyping where obtaining an understanding of a prototype and its behaviour are mandatory. The presentation formats must be consistent, syntactically lean and semantically rich.

An information extraction mechanism is critical to the success of a prototyping environment. It must be possible to instrument any part of a prototype, its data, or any of the control or data abstractions upon which it is based. It should be possible to attach a monitor to any entity which is executed each time the entity is accessed.

One of the ways of quickly building a prototype is to leave portions of it incomplete. The prototyping environment must handle incomplete prototypes by providing a facility for indicating what should occur if an attempt is made to access an undefined entity. The ability to specify default behaviour must exist.

Finally, new prototyping environments should use existing programming environments as the bases on which to build, for two reasons:

- an enormous amount of effort has been expended creating these environments, effort which should not be wasted;
- developers are familiar with them, so using them will ease the transition into the new development paradigm.

2.6 Prototyping Terminology

Prototyping terminology has not stabilized. Some authors recommend using the term *breadboard* to represent a software product without a user interface but almost completely functional, as is the case with initial hardware systems. In the same vein, a *mock up* contains a complete user interface but very few other functions. The term *pilot system* has also been suggested for early versions of software [Rzev84]. A proposed synonym for prototyping is *software architecture modeling* [Brow88]. However, use of the word *architecture* implies considering structure, eliminating the possibility of requirements prototyping which is structure independent. In this thesis, the term *prototype* refers to an executable version of a system which is used to explore concurrent system design issues. These *design prototypes* may contain any degree of functional or non-functional completeness, and may be retained or discarded depending on the reason for their development.

Chapter 3

An Overview of MLog

MLog [Kara89a] combines three programming paradigms: (1) *object-oriented programming* (OOP), with the associated packaging features, (2) *concurrent programming*, based on sequential tasks with synchronous and asynchronous message passing for intertask communication, and (3) *logic programming*, in the form of the Prolog computation model.

3.1 OOP Concepts in MLog

MLog supports the object oriented programming systems (OOPS) concepts of *object*, *class*, *meta class*, *inheritance*, *methods*, *message passing*, and *instance variables* in a manner similar to Smalltalk. The basic MLog execution units, *objects*, represent the encapsulation of procedures and data into a single entity. This is implemented in two stages:

- a class is used to define the procedures (methods) and data format (instance variables);
- an object is created as an instance of the class with a private copy of the instance variables and access to the class' methods.

Code reusability, important for fast prototyping, is achieved by combining the class notion with inheritance. In single inheritance OOPS such as MLog, every class is a *subclass*, or specialization, of another class, its *superclass*. The tree of classes is called the *class hierarchy*, with class `object` at the root. A class inherits all the methods and instance variable declarations of its superclass. This process is recursive since every superclass has its own superclass, with the exception of class `object` which by definition has no superclass. Thus, a class also inherits methods and instance variable declarations that are defined in all classes above it. A subclass may override the methods of a superclass, add new methods, and provide new instance variable declarations.

Computation in MLog is performed by objects sending messages. Each message includes a target object¹⁷, the method name, and any arguments. When a message is received by a target, the specified method is invoked which causes further computation and concludes with a reply to the original sender.

3.2 Concurrency in MLog

Unlike most OOPS, MLog supports two types of objects:

- passive objects, which are sequential but for which no mutual exclusion is provided;
- active objects, which are large-grain, sequential, communicating, autonomous tasks for which mutually exclusive access is inherently provided.

Message passing to a passive object is equivalent to a procedure call because the target object cannot execute autonomously of the arrival of a message, and cannot delay the message's processing. An interaction with an active object takes the form of a rendezvous, and will only occur if both the sender and the target agree to communicate.

The nature of an object, passive or active, is determined at object creation time. Thus, the instances of a class are not constrained to be of one type or the other. When an active object is created, it automatically sends itself a `begin` message. The `begin/0`¹⁸ method represents an active object's "main line" program.

Two types of intertask communication mechanisms exist: (1) *asynchronous*, and (2) *synchronous*. To perform an asynchronous interaction, the sender object sends the message to the target object but does not wait for a reply. For a synchronous interaction to occur, the sender object sends the message and then waits until it is processed by the target object, the target object decides when to accept messages from a sender, and then given that a message can be accepted, processes the message. A first in, first out queue is used to buffer messages awaiting processing.

When an object wishes to accept a message, it specifies a message pattern in one of two ways: (1) with a method name and argument(s), meaning a specific message, or (2)

¹⁷The target object is implicit if an object is sending to itself.

¹⁸The functor/arity form represents a method whose name is "functor" and whose argument count is "arity".

with a single uninstantiated variable, meaning any message. Starting with the first message in the message queue, the two steps of an MLog rendezvous are applied: *eligibility*, and *evaluation*. Both steps must pass for the rendezvous to complete successfully.

Figure 3.1 The MLog Rendezvous

To pass the eligibility step, an incoming message must unify with the message pattern specified by the target object. In the evaluation step, the method head is unified with the result of the eligibility step, and the corresponding method body is evaluated. If this is successful, both active objects resume execution with the unification of arguments resulting from the two rendezvous steps. If a failure occurs within a method, normal Prolog backtracking takes place. If no method clause can succeed, the evaluation step fails.

If either step fails, the message is left in the queue and the next one is tried. If no message is successful, the target object waits until a new message arrives, at which time the rendezvous steps are applied again. Searching for messages always starts at the beginning of the message queue. Figure 3.1 shows the steps of an MLog rendezvous.

A unique feature of MLog is the concept of an *anonymous target*. A message to an anonymous target specifies a class of active objects rather than a particular one. MLog will locate a target object in the class which is willing to accept the message. If none is available, the sender is suspended until one becomes available. When the rendezvous is completed, the sender is made aware of the identity of the target object which was the actual acceptor.

3.3 Prolog, OOPS, and MLog

The following Prolog and OOPS concepts are considered equivalent in MLog:

- Prolog procedures (sets of clauses with the same functor and arity) and OOPS methods;
- Prolog goal evaluation and OOPS message passing;
- Prolog dynamically-asserted facts and OOPS instance variable assignment.

Backtracking over messages sent to passive objects unbinds any arguments and propagates the failure to preceding messages. A target object backtracking over a rendezvous attempt unbinds the target's variables but has no effect on the sender because the rendezvous completed successfully. The analogous case is true for a sender object.

In MLog, elements such as symbols, numbers, lists, and structures are not treated as objects because it is considered easier to leave them as non-objects which must be treated as special cases. This allows importing existing Prolog code into MLog.

3.4 MLog Implementation

The current MLog environment is implemented in 'C' as a single Unix process on a SUN workstation. It includes tools such as a console, a class browser, a context sensitive debugger (DMLog) [Lien89], and the human interface facility. The latter consists of a set

of MLog classes and methods that provide convenient access to the windowing software, allowing MLog interfaces to be programmed entirely in MLog.

MLog is implemented using the same philosophy as Smalltalk and many versions of Prolog: develop a basic run-time system which provides a boot strapping facility and primitive operations, and then build the rest of the environment using the offered language, in this case MLog. The run-time system provides: (1) the basic Prolog computation engine (the virtual processor), (2) the concept of an object, (3) processor sharing for all of the active objects, (4) many primitive methods, and (5) interfaces to the input/output devices.

As with most run-time systems for concurrent applications, MLog contains ready-to-run queues. It contains two logical queues: (1) the *application ready-to-run queue* for active objects which are part of the user's application, and (2) the *system ready-to-run queue* for active objects which are part of MLog, called *system objects*. Consoles, the class browser, and DMLog active objects are examples of system objects. Two queues are necessary to allow system objects to run independently of application active objects [Lien89].

The current version of MLog supports two task scheduling policies for application active objects: (1) *run-til-block*, where an active object runs until it blocks voluntarily, and (2) *timesliced*, where active objects are swapped out and placed at the end of the application ready-to-run queue after a certain number of messages have been sent.

Chapter 4

Timing in Concurrent Systems and in *ProtoTool*

This chapter describes timing issues in concurrent systems and shows how these issues are incorporated into the implementation of *ProtoTool*. Note that the current version of *ProtoTool* does not support prototyping distributed systems, so no reference is made in section 4.2 to distributed timing concerns.

4.1 Timing in Concurrent Systems

The concept of time is fundamental to our thinking, and is derived from the more basic notion of the order in which events occur. In fact, timing is embodied in event ordering¹⁹. Time is inherent to concurrent systems because the word *concurrent* implies activities may occur simultaneously. In addition, concurrent systems deal explicitly with timing issues because of constructs such as delays and timed intertask communications.

Whenever we think of time, we consider the method by which it is measured. For concurrent systems, this is done using a clock. A clock is an entity which assigns a number to an event, where the number is thought of as the time when the event occurred. This definition of a clock makes no assumption about the relationship of these numbers to physical time, so clocks may be implemented using counters with no actual timing mechanism. If a system is to meet its specification correctly, the specification must be given in terms of events which are observable within the system. Thus, if the specification gives physical times, the system must contain real clocks.

¹⁹See [Lamp78] for a more complete treatment of event ordering and how it relates to time.

4.1.1 Uniprocessor Systems

Uniprocessor implementations of concurrent systems which use timing operations can be difficult to design. For instance, specifying timing parameters based on the speed of the processor may raise portability concerns. Also, it may take much use in the target environment before the timing parameters can be adjusted to suitable values, and even then, these values may change. Systems which employ timed intertask communications can be very complex and can be challenging to analyze. An advantage of these virtually-concurrent systems is that only one clock is needed, and each task has access to this single timing source.

4.1.2 Distributed Systems

Lamport [Lamp78] defines a truly concurrent or *distributed* system as “a collection of distinct processes, spatially separated, which communicate with each other by passing messages, where the message delay is not negligible compared to the time between events on a single processor.” Distributed systems complicate timing issues considerably because of the need to allow timed interactions between nodes.

Perhaps the most difficult timing problem in distributed systems is ensuring that each node “sees” the same time, that is, ensuring there is a single, global sense of time. Three possible solutions are identified in [Volz87]: (1) use of a network time server, (2) maintaining synchronized clocks, and (3) including timing parameters in internode communications.

Figure 4.1 A Distributed System with a Network Time Server

Figure 4.1 shows how a network time server is employed. Whenever a task requires timing information, it accesses the network time server with the appropriate request and obtains the necessary information. The advantage of this configuration is that the time is stored in one location, making inconsistency impossible. However, four significant problems exist. First, if timing parameters are to be accurate, the time required to access the server must be considered. It may be impossible to obtain accurately the access time because the task would have to know when the access was made, which requires an access, etc. An estimate may be the best that can be achieved. Second, in the case of a delay or timeout, the access time may be greater than the specified interval, rendering the time acquisition useless, and potentially blocking the task for longer than desired. Third, many simultaneous accesses may cause unacceptable delays. Fourth, if the server fails, the system is left with no way to obtain timing information. This may force the system to halt.

Figure 4.2 A Distributed System with Synchronized Clocks

The four problems associated with the network time server solution can be alleviated if each node maintains its own clock which is synchronized with the other clocks in the system using a master timing node, as shown in figure 4.2²⁰. In this solution, each node's clock is independent of the others. In fact, the clocks need not use the same method of associating timing numbers with events. Periodically, the master timing node sends the current time to each node so that none of the clocks drifts too far from the actual

²⁰This is the method used in telephone networks.

time. The update frequency depends on the quality of the clocks. This quality should be as high as possible so the update frequency can be minimized, and so the system will operate reliably for a long period should the master timing node fail.

The third potential solution involves completely independent node clocks and exchange of timing parameters, as depicted in figure 4.3. This solution is appropriate when it is not possible to share a common sense of time between nodes, such as between satellites exploring deep space. The exported timing parameters and local timing on the receiving node are used to manage the interaction. Some knowledge of the communication delays and the relative drift of the clocks is required to establish the best timing parameters. The primary disadvantage of this solution is that the clocks are allowed to drift without bound.

Figure 4.3 A Distributed System with Independent Clocks

Another potential solution involves task migration, which is shown in figure 4.4. With task migration, a task executing on one node is transferred to another node where execution is resumed. This transfer may involve passing the actual code for the task, or, if the code for all tasks is present on each node, it may only be an event specifying which task to run. Whenever a timed interaction is desired, the sending task is migrated to the receiving task's node where a single sense of time is available. The complexity and overhead of task migration may make this approach infeasible. If not, the migration time may prove too costly.

Figure 4.4 A Distributed System with Task Migration

4.1.3 Timing Constructs

Concurrent systems which use message passing for task synchronization, such as MLog, should contain four basic timing constructs: (1) a delay statement, (2) a timed receive, (3) a timed send, and (4) a conditional send. The question which needs addressing is *"What is the meaning of the specified time?"*.

When a task times out, its subsequent temporal behaviour depends on the degree of processor sharing involved. In a uniprocessor system where all tasks share the same processor, a task may have to wait in a ready-to-run queue before it can execute again. The duration of this wait depends on the processor's scheduling policy. In a completely-distributed system where each task has its own processor, a task runs as soon as it is scheduled. In a distributed system of uniprocessors, a task may be required to wait, but for a shorter period than if only a single processor is used. Thus, an application designed for a uniprocessor environment may display different behaviour when ported to a distributed configuration.

4.1.3.1 Delay Statement

The semantics of a delay statement are identical for distributed and uniprocessor systems because its effect is strictly local. The timer starts as soon as the task is blocked, and the task remains blocked *at least* the specified amount of time. As described in the previous paragraph, the task may have to wait longer than the timeout interval before it can run again.

4.1.3.2 Timed Receive

Like a delay statement, a timed receive's effects are local. The timer starts after it has been determined that no interaction can be initiated. If an interaction can be started before

the specified interval expires, the receiving task is awakened to process the interaction. Otherwise, the task is blocked until the interval is over. As with a delay statement, a task which times out will be guaranteed immediate execution only in a complete, γ -distributed system.

4.1.3.3 Timed Send

In a timed send, the sender says if and when the interaction may be cancelled, while the receiving task determines when the interaction may begin. An interaction is initiated which is abandoned if a receiver does not begin processing it within the specified interval. If a receiver responds, the sender is awakened when processing is finished. Otherwise, the sender is awakened when the timer expires and is then subject to its processor's scheduling policy as described above. A timed send poses three questions:

1. *When does the timer start?;*
2. *Where is the timer located?;*
3. *What is the effect of the timer's node failing?.*

In a uniprocessor system, there is no delay between when the send is done and when the information arrives at the receiver. Hence, the timer starts as soon as it is determined that the receiver cannot immediately accept the interaction. The timer used is the one on the node in question, and if that node fails, the entire system fails.

In a distributed system, there is a communication delay between when the send is performed and when the receiver is given the information. Thus, should the timer start when the interaction is initiated, which would represent the sender's time frame, or when the information arrives at the receiver, which would represent the receiver's time frame? Some, including Volz and Mudge [Volz87], feel the action must be accomplished within the required time in spite of network delays or failures. The opposite interpretation would ignore communication delays completely, thereby making all intervals queue waiting times only. The primary reason for these two interpretations is the difficulty in maintaining a single, global sense of time.

The location of the timer and the effect of the timer's node failing depends on the interpretation chosen. If the sender's time frame is considered appropriate, the sender keeps control over itself. However, this generates problems if the sender decides to

abandon the interaction while the receiver is starting to process it. As a result, some form of protocol would be necessary. If the sender's node was to fail, only the sender would be affected. If the other interpretation is used, it is the receiver's responsibility to determine when the interaction has timed out. Thus, the sender loses all control of itself. If the receiver's node fails, the sender will remain blocked unless some form of system-wide exception handling exists. This defeats the purpose of a timed send which is to allow the sender to opt out if desired.

4.1.3.4 Conditional Send

A conditional send is one which is cancelled if the receiver has not reached a point where it can immediately accept the interaction, or if other senders are already awaiting interactions. The reference to time in this definition is in the word *immediately*.

In a uniprocessor system, a conditional send is the same as a timed send with a delay of zero. This consistency indicates that a conditional send is an expression of the readiness of the receiver to accept the interaction. This can be determined easily because every task's state is available for analysis.

In a distributed system, the word *immediately* poses problems. If using the sender's time frame, every conditional send will fail because of communication delays. If using the receiver's time frame, conditional sends are possible, but communication delays will be ignored. To be consistent with the uniprocessor situation, it seems more appropriate to use the second interpretation in this case.

A possible solution to this interpretation problem is to specify both a communication interval and a queue interval. The former could be managed by the sender so that if the receiver's node fails, the sender would be able to recover. The latter would be managed by the receiver using only local timing information. If a timeout occurs, the sender would be able to determine if it was because of communication problems, or if it was because the receiver was unable to process the interaction in time.

4.1.3.5 Task Synchronization Without Message Passing

Message passing is not the only way to provide task synchronization. Facilities such as semaphores and mailboxes can be used also, and it is certainly feasible to interact with them under timing constraints. The primary conceptual difference is that tasks in

message passing systems interact directly with each other and wait logically at either the sender or the receiver, while tasks in semaphore-based or mailbox-based systems interact through intermediaries and wait logically at a neutral location. However, the concepts associated with the timing constructs presented above are applicable. For instance, a task could make some data available to another and then perform a timed interaction with a semaphore which is notified when the data is received. From the sender's perspective, these actions would have the same semantics as a timed send in a message passing system.

In terms of implementation, new issues arise such as whether to use the sender's time frame, the receiver's time frame, or a third one if the intermediary is not on the same processor as either the sender or the receiver. An algorithm is necessary for establishing whether to allocate a fixed number of intermediaries or to create them dynamically as required. If the latter, their location must be chosen from the sender's, the receiver's, or a neutral processor.

4.2 Timing Implementation in *ProtoTool*

4.2.1 Basic Implementation

Rather than using a physical mechanism to generate periodic clock pulses, *ProtoTool* uses a message counter to simulate a clock. Each time a message is passed to an active or passive object, the counter is incremented to represent another "clock tick". This makes the timing portions of MLog and *ProtoTool* completely portable because no interaction with the underlying hardware or operating system is required.

Currently, the smallest time resolution provided by *ProtoTool* is one second. This resolution was chosen because any attempt to be more accurate would be wasteful given the differences between MLog's prototyping environment and the eventual target environment. The timing facilities provided by *ProtoTool* are conceptually consistent with those of production systems, but they cannot be used to obtain fine-grain timing information. An assumption is made that the number of messages passed is proportional to the amount of time which would be required in the target environment. A better model would involve directly mapping target instructions to MLog operations. This would provide an improved relationship between MLog prototypes and target implementations

and would allow tailoring MLog to particular productions systems. For example, the following relationship could be specified: "an MLog unification of two integers corresponds to twenty target instructions".

Timing parameters are specified in the following form: `[Hours, Minutes, Seconds]`. This external time representation is natural and fits well into MLog because of Prolog's list processing capabilities. The current time, in seconds, is stored internally as a single integer rather than as hours, minutes, and seconds because of the ease with which an integer can be modified. In addition, should it be deemed feasible to use a resolution smaller than one second, fewer modifications would be necessary²¹. The ideal solution would allow the user to specify the resolution, and then *ProtoTool* would automatically adjust to the change.

There are two modes for timed interactions: (1) relative, and (2) absolute. Relative interactions specify a time in relation to the current time, whereas absolute interactions specify an exact time. In either case, zero or a time already passed²² means immediately.

ProtoTool provides facilities to read the current time, and to set the time forward, backward, or to an absolute value. Setting the time forward poses no significant implementation problems. This simply gives the application the impression that it was unable to run for a period of time. Setting the time backward is much more difficult. Currently, everything done between the current time and the new time is not undone. For example, objects scheduled between the current time and the new time remain scheduled. A better algorithm would continuously save the application's state to facilitate returning to a particular time. Such an algorithm is beyond the scope of this thesis.

A queue called the *delay queue* is used to hold timeout information. This is a priority queue ordered by time, with the smallest time at the head of the queue. Each entry contains the following:

- the absolute time when the timeout expires (all times on the delay queue are absolute to permit easy ordering);
- a reference to the object for which the timeout pertains;
- the reason for the timeout;

²¹The external time representation would need to be changed accordingly.

²²This includes negative times.

- the current level of nested accepts²³ for the object in question.

4.2.2 Timed Interaction Facilities

ProtoTool provides four methods which implement timed interactions: (1) **waitToAccept/6** which performs a timed receive for messages, (2) **waitToSend/6** which executes a timed send of a message, (3) **waitForInterrupt/6**²⁴ which does a timed receive for an interrupt to occur, and (4) **waitForInteraction/7** which combines the effects of **waitToAccept/6** and **waitForInterrupt/6**. In addition, method **delay/2** is *ProtoTool's* version of a delay statement. Each of these methods fails if any parameter is in an incorrect form.

4.2.2.1 waitToAccept/6

The format of the **waitToAccept/6** method is:

```
waitToAccept(Message, Mode, Timeout, Actual, Else, Result);
```

Message specifies the messages which are eligible for acceptance. An uninstantiated variable means any message is eligible. **Mode** is either **relative** or **absolute** and represents the timing mode for **Timeout**, the maximum time to wait or the time at which to awaken, respectively. The actual time waited is returned in **Actual**. Should the timeout period expire, method **Else** is performed. This is included because of the difficulty in coding an if-then-else structure in Prolog. The result of the interaction is returned in **Result**: **message** if a message was received successfully, or **timeout** if not. If **Mode** is **relative** and **Timeout** is **[0,0,0]**, or if **Mode** is **absolute** and **Timeout** has passed, a non-blocking accept for **Message** is performed. As an example, consider an active object which controls an elevator. The following piece of code implements a three second timeout on the doors closing:

²³A nested accept is an accept within an accept. Nested accepts are exited in reverse order of occurrence. The nesting level is used to associate timeouts with accepts.

²⁴See chapter five for details about interrupts in *ProtoTool*.

```

    .
    waitToAccept(closeDoors, relative, [0, 0, 3], Actual, closeDoors, Result),
    .
closeDoors :- ...

```

4.2.2.2 waitForInterrupt/6

The `waitForInterrupt/6` method has the following form:

```
waitForInterrupt(Priority, Mode, Timeout, Actual, Else, Result)
```

The priority of interrupt to be waited for is specified by `Priority`. The result is `interrupt` if the interrupt occurred, or `timeout` if not. This method fails if the specified interrupt is undefined. The purpose of this method is to allow an active object to be notified of an interrupt so processing which is inappropriate for an interrupt service routine can be completed. If `Mode` is `relative` and `Timeout` is `[0, 0, 0]`, or if `Mode` is `absolute` and `Timeout` has passed, `Else` is attempted immediately. Assuming an interrupt of priority ten is used to signal the doors closing, the elevator control example from `waitToAccept/6` becomes:

```

    .
    waitForInterrupt(10, relative, [0, 0, 3], Actual, closeDoors, Result),
    processResult(Result),
    .
processResult(timeout).
processResult(interrupt) :-
    closeDoors.
closeDoors :- ...

```

4.2.2.3 waitForInteraction/7

The format of the `waitForInteraction/7` method is the following:

```
waitForInteraction(Message, Priority, Mode, Timeout, Actual, Else, Result)
```

This method combines the features of `waitToAccept/6` and `waitForInterrupt/6`. All the parameters are as previously described, with the exception of `Result`, which is `message` if a message was received, `interrupt` if an interrupt occurred, or `timeout` if `Timeout` expires. If `Mode` is `relative` and `Timeout` is `[0, 0, 0]`, or if `Mode` is `absolute` and `Timeout` has passed, a non-blocking accept for `Message` is executed. Using the

elevator control example again, assume that an interrupt of priority ten indicates the doors should be closed and that a message to method `outOfService/0` means the elevator should be taken out of service. The following code implements these requirements:

```

        .
        waitForInteraction(outOfService, 10, relative, [0, 0, 3], Actual,
                           closeDoors, Result),
        processResult (Result),
        .
processResult (timeout) .
processResult (message) .
processResult (interrupt) :-
    closeDoors.
closeDoors :- ...
outOfService :- ...

```

4.2.2.4 waitToSend/6

The interface to the `waitToSend/6` method is:

```
waitToSend(Message, Mode, Timeout, Actual, Else, Result)
```

`Message` is the message to be sent. The destination of the message can be a specific active object or an anonymous acceptor. `Result` will be returned as `ok` if the send succeeded, or `timeout` if `Timeout` expired. The remaining parameters and their semantics are the same as `waitToAccept/6`. This method fails if the message destination is illegal. If `Mode` is `relative` and `Timeout` is `[0, 0, 0]`, or if `Mode` is `absolute` and `Timeout` has passed, `Else` is tried immediately²⁵. As an example, consider accessing a remote file server. The following code is part of an active object which informs the user every ten seconds when a write is being attempted:

²⁵A conditional send does not exist currently in *ProtoTool*.

```

    repeat,
    waitToSend(server-write(Data), relative, [0,0,10], Actual,
               tellUser, Result),
    processResult(Result),
    processResult(ok).
    processResult(timeout) :- fail.
    tellUser :- ...

```

If the timeout occurs, method `tellUser/0` is executed and `Result` is set to `timeout`. Method `processResult/1` then fails, causing backtracking into the `repeat` statement, and hence, another message to `waitToSend/6`²⁶.

4.2.2.5 delay/2

The format of the `delay/2` method is:

```
delay(Mode, Timeout)
```

`Mode` and `Timeout` have the same meaning as for `waitToAccept/6`. If `Mode` is `relative` and `Timeout` is `[0,0,0]`, or if `Mode` is `absolute` and `Timeout` has passed, `delay/2` succeeds but does not block the calling active object. The following example shows the MLog code for an active object which performs diagnostics every twenty minutes for the first hour of every day:

```

begin :-
    repeat,
    theValueOf(hours, Hours),
    delay(absolute, [Hours, 0, 0]),
    delay(relative, [0, 20, 0]),
    runDiagnostics,
    delay(relative, [0, 20, 0]),
    runDiagnostics,
    delay(relative, [0, 20, 0]),
    runDiagnostics,
    NewHours is Hours + 24,
    makeTheValueOf(hours, NewHours),
    fail.
runDiagnostics :- ...

```

²⁶This "repeat-fail" construct is a common way of implementing iteration in Prolog.

4.2.3 Selected Timing Algorithms in *ProtoTool*

4.2.3.1 Timed Interactions

To be consistent with the theory presented earlier, a timed interaction is cancelled if it is not started within the specified interval. MLog complicates the theoretical model by allowing interactions to fail²⁷. Figure 4.5 illustrates possible scenarios and their outcomes.

Figure 4.5 MLog Interactions and Timeouts

In the top example, the interaction starts and succeeds within the timeout interval, so the interaction is successful. In the next example, the first attempt fails, but the second succeeds before the timeout expires, resulting in a successful interaction. The third example shows another successful interaction because although it did not succeed until after the timeout expired, the interaction began within the timeout period. The next

²⁷An active object can begin an interaction, examine the contents of the message, and then fail the interaction if the contents are not satisfactory. The message is then put back on the message queue. These actions are transparent to the message's sender.

example shows a single, failed interaction attempt during the timeout period. As no successful interaction began before the timeout occurred, the interaction is unsuccessful. The fifth example illustrates an unsuccessful attempt because the interaction failed after the timeout expired.

The bottom example in figure 4.5 illustrates a nested accept. Care must be exercised to ensure that the timeout is applied to the correct level of acceptance. This is done by having each active object keep track of its own nesting level. When an item is placed in the delay queue, the corresponding nesting level is included. When a timeout occurs, it is ignored if its nesting level does not match the current one of the active object in question. In the example, the timeout applies to the first interaction, so with respect to timing, this example is the same as the third one described above.

The following pseudo code gives the algorithm for the `waitToAccept/6` method described in section 4.2.2.1:

```

generate delay entry, place on delay queue
wait for message
if successful interaction within timeout interval then
    remove delay entry from delay queue
    resume receiver after timed accept
else
    if receiver blocked at correct nesting level then
        resume receiver to try Else of timed accept
    else % must be inside a nested accept
        if successful interaction then
            resume receiver after timed accept
        else
            when check for another message, detect timeout
            resume receiver to try Else of timed accept
        endif
    endif
endif
endif

```

Below is the algorithm for method `waitToSend/6`, in pseudo code:

```

generate delay entry, place on delay queue
send the message
if successful interaction within timeout interval then

```

```

    remove delay entry from delay queue
    resume sender after timed send
else
    expire message so no more active objects try read it
    if no active object currently reading it then
        resume sender to try Else of timed send
    else
        wait until reader is done with it
        if successful interaction then
            resume sender after timed send
        else
            reader resumes sender to try Else of timed send
    endif
endif
endif
endif

```

For a timed accept, it is the MLog engine's responsibility to detect a timeout if no interaction is in progress, while the receiver is responsible for detecting its own timeout if an interaction fails. For a timed send, the onus is on MLog's engine to detect a timeout if no interaction is in progress. Otherwise, it is the receiver's responsibility to detect if a failed interaction contains an expired message, and if it does, to schedule the sender to attempt the `Else` of the timed send.

4.2.3.2 Scheduling Objects After Timeouts

When a timeout expires, the MLog engine uses the following algorithm to manipulate the object in question:

```

case origin of timeout
    from delay/2:
        put object on ready-to-run queue
    from waitToAccept/6:
        if (object is waiting for message and
            timeout's nesting level = current nesting level of object) then
            put object on ready-to-run queue
        else
            let object detect timeout if interaction fails
        endif
    from waitToSend/6:
        make object's message expired
        if no active object currently reading message then
            put object on ready-to-run queue

```

```

else
  let message's reader detect timeout if interaction fails
endif
from waitForInterrupt/6:
  remove object from interrupt's waiters queue
  put object on ready-to-run queue
from waitForInteraction/7:
  if (object is waiting for message and
      timeout's nesting level = current nesting level of object) then
    remove object from interrupt's waiters queue
    put object on ready-to-run queue
  else
    let object detect timeout if interaction fails
  endif
endcase

```

4.2.3.3 Incrementing the Current Time

When each second elapses, the delay queue must be checked to see if any timeouts have occurred. This is accomplished by stopping the currently-running active object, incrementing the time, checking the delay queue and scheduling any active objects which have timed out, and then resuming the stopped active object. The stopped active object is placed on the front of the interrupt ready-to-run²⁸ queue so that it will be the first to run. This ensures that scheduling does not work on a preemptive priority basis.

4.2.4 Timing Example

The following MLog code implements a small concurrent system which illustrates how *ProtoTool's* timing facilities can be used. It consists of one client and three servers, all implemented as active objects as shown in figure 4.6. The client makes service requests on a periodic basis using MLog's anonymous target feature. If a request cannot be serviced within a specified interval, the client reports a timeout. The servers perform timed accepts for requests on their `doIt/0` method, and report a timeout if a request does not arrive during the timeout period. The servers simulate processing by calling method `delay/2` after `doIt/0` has succeeded. The output has been appended to the code.

²⁸See chapter five for details about the interrupt ready-to-run queue.

Figure 4.6 The Client-Server Timing Example

```

% .....
%
%   This MLog code implements a small concurrent system which contains one client and three
%   servers. The client periodically requests service from one of the servers using MLog's
%   anonymous target feature. The servers simulate processing of requests by delaying. All
%   interactions are timed.
% .....
% .....
% class: client of object.
%
% description: - implements a client which makes requests for service from instances of class
%               "server"
%               - ivar "interval" specifies how long to wait for service before reporting a
%                 timeout
%               - does an anonymous send to class "server" and if an interaction is
%                 successful, reports the name of the server which accepted the request
% .....
class: client of object.

ivars: [interval].

begin :-
  theValueOf(interval, interval),
  repeat,
    waitToSend(server(Server)-dolt, relative, interval, duration, timeout(interval), result),
    processResult(result, server, duration),
  fail.timeout(interval) :-

```

```

mainConsole-(write("client: timed out after "),writeln(Interval)).

processResult(ok,Server,Duration) :-
mainConsole-(write("client: interaction successful with "),
write(Server),write(" after "),writeln(Duration)).

% .....
%
% class: server of object.
%
% description: - implements servers which wait for requests on their "doIt" method and then
%               do a post-rendezvous delay to simulate processing of the request
%               - ivar "interval" specifies how long the servers wait for a request before
%                 reporting a timeout
%               - ivar "delay" specifies the duration of the post-rendezvous processing
% .....

class: server of object.

ivars: [interval,delay].

begin :-
name(MyName),
theValueOf(interval,Interval),
theValueOf(delay,Delay),
repeat,
waitToAccept(doIt,relative,Interval,Duration,timeout(MyName,Interval),Result),
processResult(Result,Duration,Delay,MyName),
fail.

timeout(MyName,Interval) :-
mainConsole-(write(MyName),write(": timed out after "),writeln(Interval)).

processResult(message,Duration,Delay,MyName) :-
mainConsole-(write(MyName),write(": received request after "),writeln(Duration)),
delay(relative,Delay).

doIt.

% .....
%
% class: main of object.
%
% description: - creates the client and the servers, allows them to run for 1 minute, and
%               then removes them
% .....

class: main of object.

begin :-
mainConsole-nl,
client-new(Client,active,[interval={0,0.2}]),
server-new(server1,active,[interval={0,0.2},delay={0,0.4}]),
server-new(server2,active,[interval={0,0.2},delay={0,0.5}]),
server-new(server3,active,[interval={0,0.2},delay={0,0.6}]),
delay(relative,{0,1,0}),
client-removeInstance(Client),
server-removeInstance(server1),
server-removeInstance(server2),
server-removeInstance(server3).At {0,0,0}: No application objects able to execute

```

MLog v4.0? load(clientserver).

At [0,0,0]: No application objects able to execute

MLog v4.0? run.

```
server1: received request after [0,0,0]
client:  interaction successful with #server1 after [0,0,1]
server2: received request after [0,0,3]
server3: timed out after [0,0,2]
client:  interaction successful with #server2 after [0,0,0]
server1: received request after [0,0,3]
server3: timed out after [0,0,2]
client:  interaction successful with #server1 after [0,0,0]
server3: received request after [0,0,3]
server2: timed out after [0,0,2]
client:  interaction successful with #server3 after [0,0,0]
server1: received request after [0,0,3]
server2: timed out after [0,0,2]
client:  interaction successful with #server1 after [0,0,0]
server3: received request after [0,0,0]
server2: timed out after [0,0,2]
client:  interaction successful with #server3 after [0,0,1]
server2: received request after [0,0,3]
server1: timed out after [0,0,2]
client:  interaction successful with #server2 after [0,0,0]
server3: received request after [0,0,0]
server1: timed out after [0,0,2]
client:  interaction successful with #server3 after [0,0,1]
server2: received request after [0,0,0]
server1: timed out after [0,0,2]
client:  interaction successful with #server2 after [0,0,1]
server3: received request after [0,0,0]
server1: timed out after [0,0,2]
client:  interaction successful with #server3 after [0,0,1]
server1: received request after [0,0,3]
server2: timed out after [0,0,2]
client:  interaction successful with #server1 after [0,0,0]
server2: timed out after [0,0,2]
client:  timed out after [0,0,2]
server3: received request after [0,0,3]
server2: timed out after [0,0,2]
server1: timed out after [0,0,2]
client:  interaction successful with #server3 after [0,0,0]
server2: received request after [0,0,3]
```

At [0,1,1]: No application objects able to execute

MLog v4.0?

Chapter 5

Interrupts in Concurrent Systems and in *ProtoTool*

5.1 Interrupts in Concurrent Systems

An interrupt can be defined as an exceptional event which causes a processor to make a temporary transfer of control from its current program to another one which services the event. Interrupts can be used to signal the completion of an operation, to initiate a new one, or to indicate the occurrence of a hardware or a software error. One way to view an interrupt is as a subroutine call which is generated externally to the program. Using interrupts can improve an application's performance because no polling of potential interrupt sources is necessary. This means the processor spends a larger percentage of time running the application. In addition, the application is more responsive to external requests because the interrupt sources initiate servicing rather than waiting for the processor to query them.

The following actions occur when an interrupt is serviced:

- the processor identifies the source of the interrupt;
- the address of the interrupt service routine (ISR) is obtained;
- the execution state is saved as for a subroutine call;
- execution begins at the start of the ISR;
- when the ISR completes, control is transferred back to the interrupted program.

In most environments, software cannot control when interrupts occur, but can control their effects. Operations such as enabling and disabling interrupts globally and individually, masking interrupts, and clearing interrupts are normally provided for interrupt control.

5.1.1 Interrupt-to-Processor Connection Configurations

It is the responsibility of the processor to detect when an interrupt has occurred. This is accomplished by examining internal data structures or registers at a fixed point in the execution cycle. When an interrupt is detected, the processor must determine the source and must obtain the address of the associated ISR. How this is done depends on the interface between the processor and the interrupt sources. Three schemes are discussed here: (1) single-line, (2) multiple-line, and (3) vectored.

5.1.1.1 Single-Line Interrupts

Figure 5.1 Single-Line Interrupt Configuration

Using this method, each potential interrupt source is connected to the processor via the same physical link as shown in figure 5.1. When an interrupt is detected during examination of the interrupt request register, the processor polls each source to determine which one generated the interrupt. This polling process can be slow, especially if there are many sources, but there is no limit to the number of potential sources. The priorities of the sources are established by the polling order, where the first one polled has the highest priority. To obtain the address of the ISR, the processor can extract it when polling, or the source can place it, along with any associated data, in a location accessible by the processor for subsequent retrieval.

5.1.1.2 Multiple-Line Interrupts

Figure 5.2 Multiple-Line Interrupt Configuration

In a multiple-line configuration, that shown in figure 5.2, each interrupt source has a unique priority, so the processor can identify immediately the source. The primary disadvantage of this method is that, to be manageable, the number of sources must be kept small. The address of the ISR and any data are obtained in the same way as for the single-line solution. The processor must be provided the sources' priorities because each source is independent of the others.

5.1.1.3 Vectored Interrupts

Figure 5.3 Vectored Interrupt Configuration

Shown in figure 5.3, a vectored interrupt scheme has roughly the same physical

appearance as the multiple-line configuration of figure 5.2. The difference is the amount of processor intervention required. Vectored interrupts cause a direct transition to the associated ISR, freeing the processor from identifying the interrupt's source. Each of the unique interrupt lines is bound to a location in the processor's memory which contains the address of the ISR²⁹. When an interrupt occurs, the ISR's address is fetched and control is transferred there. Unlike multiple-line interrupts, vectored interrupts need no processor action to initiate servicing because the interrupt's identity and the address of the ISR are both known.

5.1.2 Interrupt Priority Interpretations

The relative priorities of interrupts are used to choose which one to process in the event of simultaneous occurrences. In such cases, the highest priority pending interrupt is the one selected for processing. These priorities can also be related to task priorities in the application's software in one of two interpretations. In the first one, any interrupt has a higher logical priority than any task, meaning any task can be interrupted at any time by any interrupt. In the second interpretation, only interrupts with a priority higher than that of the current task can interrupt it, resulting in a uniform treatment of task and interrupt priorities. The interpretation employed depends on the application. Usually, an ISR disables interrupts while it is running. If such is not done, the occurrence of a higher priority interrupt may temporarily suspend processing of the executing ISR.

5.1.3 Interrupt Occurrence Distributions

From an application's perspective, interrupts occur with a distribution in time, such as uniform, normal, Poisson, etc. An interrupt's distribution depends on its nature. For example, clock interrupts are uniform because they occur at evenly-spaced intervals. Conversely, interrupts from a keyboard normally occur with a non-uniform distribution. Before designing a system which uses interrupts, a developer must know the occurrence distribution of each one to correctly configure the servicing mechanism.

²⁹Hence, changing an ISR requires modifying one memory location.

5.1.4 Queuing of Interrupts

If an interrupt cannot be processed when it occurs³⁰, a decision must be made by the processor about whether it should be queued, and if so, how. Three possible queuing modes are presented here: (1) immediate, (2) queued, and (3) timed.

5.1.4.1 Immediate Interrupts

Immediate interrupts are missed if they cannot be processed as soon as they occur. Hence, these interrupts correspond to a conditional send from the interrupt source. Figure 5.4 shows an example of the handling of an immediate interrupt. Interrupt 1 is processed as soon as it occurs. Interrupt 2 is missed because interrupt 1's processing has not finished. Interrupt 3 is missed because interrupts are disabled. Interrupts 4 and 5 are handled immediately. This queuing mode is appropriate for very high priority interrupts, such as ones which would be associated with the failure of a device in a nuclear reactor.

Figure 5.4 Example Processing of an Immediate Interrupt

5.1.4.2 Queued Interrupts

Queued interrupts have an associated *pending interrupt queue* which stores occurrences awaiting servicing. Interrupts which cannot be processed immediately are queued until the queue is full, at which time subsequent interrupts are lost. These interrupts have the appearance of an asynchronous interaction initiated by the interrupt source where the message queue's size is bounded. An example of queued interrupt processing with a queue size of three is shown in figure 5.5. Interrupt 1 is serviced as soon as it occurs. Interrupts 2, 3, and 4 are queued because interrupts are disabled. Because the queue is now full, interrupt 5 is missed. When interrupts are enabled, interrupts 2, 3, and 4 are processed in succession. Interrupt 6 is queued because processing of interrupt 3

³⁰This would happen, for instance, if interrupts are disabled.

is in progress, but is then serviced when it is the only entry remaining in the pending interrupt queue.

Figure 5.5 Example Processing of a Queued Interrupt (Queue Size = 3)

5.1.4.3 Timed Interrupts

Timed interrupts can remain pending for a specified duration, after which they are missed. Only one occurrence of a timed interrupt can be pending, but it is possible to have one being processed and one pending. A timed send from the interrupt source is an appropriate model for these interrupts. In figure 5.6, interrupt 1's processing begins immediately. Interrupt 2 is made pending because interrupt 1 is still being handled. Interrupt 3 is missed because interrupt 2 is pending. Servicing of interrupt 2 starts when that of interrupt 1 ends. Interrupt 4 is first made pending and then is missed when its timeout expires after interrupts were disabled. Interrupt 5 is handled as soon as it occurs.

Figure 5.6 Example Processing of a Timed Interrupt

5.1.5 Interrupt Tasks

There are situations when an interrupt cannot be processed completely by its ISR. For example, performing input or output inside an ISR is considered poor design because it keeps the processor running at a high priority longer than necessary. One solution to this problem is to associate with an interrupt a task which is notified by the ISR when interrupt processing is to be completed. Once scheduled, this *interrupt task* is treated as a "regular" task which can interact with other tasks and perform time-consuming operations such as input and output and intertask communication.

If an ISR tries to notify an interrupt task but none is waiting, two approaches can be used. In the first, the interrupt is considered missed because it could not be processed completely. In the second, the interrupt is deemed processed because the ISR was able to run, despite the fact that it was not processed entirely. The first is the more intuitive method. A solution to the problems of the second interpretation would involve dynamically creating interrupt tasks, but this may prove too costly in terms of execution time.

5.2 Interrupts in *ProtoTool*

5.2.1 *ProtoTool's* Interrupt Models

ProtoTool models interrupts as entities which autonomously gain control of MLog's virtual processor. These entities can be considered abstract events which take the form of the interface between the type of system being prototyped and its execution platform. Because *ProtoTool* is designed to help prototype concurrent software systems, these entities are called *interrupts* as that is the name given to the interface between such software systems and the hardware on which they execute. If the system being prototyped is part of a hardware system, these interrupts would correspond to an event such as the voltage on a line being changed.

Interrupts are simulated using software rather than generated physically because the latter would require interaction with the supporting hardware and operating system. By being architecture-independent, MLog's interrupt mechanism needs no modifications when ported to a different run-time system.

Each interrupt is represented within *ProtoTool* as a passive object whose instance variables contain information describing the interrupt. A passive object is used for two reasons, one conceptual and one pragmatic. Conceptually, information about interrupts such as priority and current status is persistent, while interrupt processing is transient. Thus, interrupt information should be stored in a data structure which maintains a constant, stable state. This is an ideal situation for a passive object. Pragmatically, it may be necessary to interact with an interrupt at any time, including during processing. If active objects are employed, the resulting rendezvous could alter the behaviour of the application, thus introducing undesirable side effects.

ISRs are executed in the context of active objects called *interrupt handlers*. When an interrupt is defined, an interrupt handler is created automatically and is associated with the interrupt. The interrupt handler is transparent to the user as all interactions are with the passive interrupt object. Using active objects this way is a feasible approach because:

- an interrupt is “active” when it is being processed, therefore it is consistent conceptually to use a dedicated active object to execute its ISR;
- dynamically associating interrupt handlers with interrupts adds unnecessary overhead and raises the issue of the number of interrupt handlers to create;
- an interrupt cannot interrupt itself, so there is no concern of needing more than one interrupt handler for any interrupt.

ProtoTool allows “regular” active objects to take the form of interrupt tasks by binding dynamically to interrupts. Interrupt tasks need not have the same priority as the interrupt for which they are waiting. More than one interrupt task may exist for an interrupt, in which case they are notified on a first-waited-first-notified basis, independent of their relative priorities. In addition, an interrupt task may wait for different interrupts at different times. An active object waits for an interrupt using methods `waitForInterrupt/6` and `waitForInteraction/7`, both of which are described in chapter four, and methods `waitForInterrupt/1` and `waitForInteraction/3`. The latter two have the same semantics as `waitForInterrupt/6` and `waitForInteraction/7` respectively, but are not timed interactions. An interrupt is considered missed if it attempts to notify an interrupt task but none is waiting.

Because ISRs should be run as indivisible units, MLog's timesliced scheduling policy is not used when executing ISRs. However, interrupt processing can be suspended temporarily by an interrupt of higher priority. It is considered poor design for an ISR to perform any operations which would cause it to become suspended, such as calls to *delay/2* or rendezvous attempts. *ProtoTool* allows these activities in order to give the user maximum feedback about potential design flaws³¹.

The interrupt model used by *ProtoTool* implements a vectored interrupt scheme. Because each interrupt has its own priority, each effectively has a unique interrupt line. Further, because interrupt handlers execute the ISRs, running the appropriate interrupt handler causes a direct transition to the ISR in question.

ProtoTool supports both of the interrupt priority interpretations described previously. When running in *software has priority* mode, only interrupts with a higher priority than that of the currently-running active object can interrupt it. In *interrupts have priority* mode, any interrupt has a higher logical priority than any active object. Regardless of which mode is in effect, interrupt A can suspend processing of interrupt B if interrupt A has the higher priority.

ProtoTool supports the immediate, queued, and timed interrupt queuing modes exactly as described earlier.

5.2.2 *ProtoTool's* Interrupt Types

ProtoTool currently supports five interrupt types: (1) *periodic*, (2) *once*, (3) *repeat*, (4) instance variable read (*ivarread*), and (5) instance variable write (*ivarwrite*). The interrupt class hierarchy is illustrated in figure 5.7.

As figure 5.7 shows, class *interrupt* is a superclass of the individual interrupt classes. The class methods in class *interrupt* implement functions which deal with all interrupt types, such as:

- removing an interrupt;
- globally enabling and disabling interrupts;
- checking the format of instance variables which are common to all interrupt types, such as priority and queuing mode;

³¹This does not mean that such activities are condoned.

- obtaining and outputting information such as which interrupts are enabled and which are disabled;
- obtaining status information such as whether interrupts are enabled.

Figure 5.7 The Interrupt Class Hierarchy

The instance methods in class `interrupt` address concerns of individual interrupts which are common to all interrupt types, some of which are:

- obtaining status information such as whether an interrupt is enabled and whether processing is in progress;
- individually enabling and disabling;
- displaying an interrupt's description;
- changing an interrupt's priority or queuing mode.

Each of the individual interrupt classes contains class methods which are used when an interrupt is defined. Because each class has a different description, some of the methods which create a new interrupt and check its parameters are specific to the individual classes. Instance methods exist within each interrupt class for changing an interrupt's definition³² and for obtaining information peculiar to individual interrupts.

³²For example, changing the period of a periodic interrupt.

5.2.2.1 Periodic Interrupts

Periodic interrupts are instances of class `periodicInterrupt` and represent interrupts which occur at an evenly-spaced interval specified by the user. Applications do not directly trigger periodic interrupts³³. It is the responsibility of MLog's virtual processor to generate these interrupts at the appropriate times. If MLog is being used to prototype an operating system, these interrupts could be used as clock pulses.

5.2.2.2 Once Interrupts

When a once interrupt is defined, a list of times is given, each member of which is a time when an interrupt should occur, relative to the time at definition. For example, if the time at definition is `[1, 5, 40]` and the time list is `[[1, 3, 10], [1, 6, 18], [2, 43, 36]]`, interrupts will occur at `[2, 8, 50]`, `[2, 11, 58]`, and `[3, 49, 16]`. When the last interrupt has occurred, once interrupts can no longer affect the application. These interrupts are useful for determining how well a design handles non-uniform or bursts of interrupts. Once interrupts are instances of class `onceInterrupt`.

5.2.2.3 Repeat Interrupts

Repeat interrupts, instances of class `repeatInterrupt`, are similar to once interrupts except that when the last interrupt in the time list has occurred, each interrupt is generated again using the last interrupt's time as the new time base. This has the effect of creating a repeating interrupt pattern. This interrupt type is a combination of periodic and once interrupts, allowing the user to define interrupts which are periodic on a large scale, but not necessarily periodic on a small scale.

5.2.2.4 Ivarread Interrupts

An `ivarread` interrupt is specified with the following data structure: `[Object, Ivar]`. These interrupts occur when instance variable `Ivar` is read by active object `Object` using method `theValueOf/2`. If `Ivar` is left uninstantiated, all instance variables read by `Object` are applicable. If `Object` is uninstantiated, any object reading `Ivar` will generate the interrupt. If both are uninstantiated, any object reading any instance variable will

³³This is also true of once and repeat interrupts.

generate the interrupt. Occurrences of `ivarread` interrupts are controlled by the application, unlike the timed interrupts discussed previously which are generated automatically by MLog's engine. Thus, users have the ability to generate interrupts anywhere in the application by reading an instance variable which is associated with an `ivarread` interrupt. Also, the effects of an interrupt occurring when data is read can be investigated.

5.2.2.5 Ivarwrite Interrupts

Ivarwrite interrupts are similar to `ivarread` interrupts except they are triggered when an instance variable is written using method `makeTheValueOf/2`.

5.2.3 ProtoTool's Interrupt Handlers

ProtoTool's interrupt handlers are active objects which are instances of an ISR class whose `processInterrupt/1` method implements the ISR. The interrupt handler class hierarchy is shown in figure 5.8.

Figure 5.8 The Interrupt Handler Class Hierarchy

Class `interruptHandler` contains the `begin/0` method which represents the mainline of each interrupt handler. This method is implemented as follows:

```

begin :-
    repeat,
        theValueOf(interrupt, Interrupt),
        waitToAccept($interrupt),
        getIsrData(Data),
        runInterrupt(Interrupt, Data) .

```

```

runInterrupt (Interrupt, Data) :-
    processInterrupt (Data), !,
    interruptDone (Interrupt, true) .
runInterrupt (Interrupt, _) :-
    interruptDone (Interrupt, false) .

```

Interrupt handlers are sent the special message `$interrupt` from MLog's virtual processor when they are to begin executing. The `getIsrData/1` method is used to retrieve any data associated with the interrupt. This data is then passed to method `runInterrupt/2` for processing along with the object identifier of the interrupt's passive object. Method `runInterrupt/2` attempts to execute the interrupt's ISR by sending a message to method `processInterrupt/1`. If this succeeds, method `interruptDone/2` is called with `true` as the status. If method `processInterrupt/1` fails, the status given to method `interruptDone/2` is `false`. Method `interruptDone/2` fails unless the specified interrupt was removed during servicing. Thus, the `begin/0` method continues until the interrupt is deleted because the result of method `runInterrupt/2` is the same as that for method `interruptDone/2`.

To implement a specific ISR, a subclass of class `interruptHandler` must be created which may override the default `getIsrData/1` method and should override the default `processInterrupt/1` method. The default `getIsrData/1` method returns an uninstantiated variable to indicate no data was retrieved. The default `processInterrupt/1` method sends a message to MLog's main console indicating that the interrupt in question has no ISR, and then fails to reflect that servicing was not successful.

ISR class `printIsr` is provided as part of MLog and prints the time at which the interrupt occurred, preceded by a user-defined header³⁴. Its code is:

³⁴This ISR class is intended primarily for debugging as it performs output.


```

class: printIsr of interruptHandler.

ivars: [terminal=mainConsole,header=""].

processInterrupt(_) :-
    getTime(Time),theValueOf(terminal,Terminal),
    theValueOf(header,Header),
    performAsync(Terminal-(write(Header),write("interrupt at "),
        writeln(Time))),!.

```

Class `notifySoftwareIsr` attempts to notify an interrupt task by using the following code:

```

class: notifySoftwareIsr of interruptHandler.

processInterrupt(_) :-
    % get object id of associated passive interrupt object
    theValueOf(interrupt,Interrupt),
    notifySoftwareOfInterrupt(Interrupt),!.

```

Method `notifySoftwareOfInterrupt/1` fails if no interrupt task is waiting for the interrupt.

The values of an interrupt handler's instance variables can be specified at run-time using the `isr` instance variable of the associated interrupt³⁵. For example, if the following call to method `new/3` is made at `[0,0,0]`, a once interrupt of priority 10 and time list `[[0,0,5]]` will be created whose interrupt handler is an instance of class `printIsr` which will print "Periodic interrupt at `[0,0,5]`" to output terminal `OutputTerminal` when the interrupt occurs:

```

onceInterrupt-new(_,passive,[priority=10,times=[[0,0,5]],
    isr=[printIsr,
        [header="Periodic ",
        terminal=OutputTerminal]]]),

```

5.2.4 *ProtoTool's* Interrupt Information

Each instance of class `interrupt` stores, in instance variables, information describing

³⁵See section "*ProtoTool's* Interrupt Information" for details.

the corresponding interrupt. The declaration of these instance variables is shown here, including their default values:

```
class: interrupt of object.

ivar: {running=false,enabled=false,suspended=false,
      pending=[],processed=[],missed=[],waiters=[],
      howqueued=[immed,_],new=true,isr=nil,
      handler=nil,definition=nil}.
```

The following instance variables assume a boolean value of **true** or **false**:

- **running**: **true** when the interrupt is being processed;
- **enabled**: **true** if the interrupt is to be seen by the application;
- **suspended**: **true** if the interrupt is defined but cannot affect the application³⁶;
- **new**: **true** before the interrupt has been enabled for the first time.

The queuing mode is contained in instance variable **howqueued** and has one of the following forms: (1) **[immed,_]**, (2) **[queued,QueueSize]**, or (3) **[timed,TimeoutInterval]** for immediate, queued, and timed interrupts respectively.

Interrupt occurrences whose processing has been completed are stored as a list in instance variable **processed**. Each element is of the form **[TimeOccurred,TimeStarted,TimeFinished]** where **TimeOccurred** is the time when the interrupt occurred, **TimeStarted** is the time when processing began, and **TimeFinished** is the time when processing ended. When processing is in progress, the **TimeFinished** field is omitted. Pending interrupts, stored as a list in instance variable **pending**, contain the time when the interrupt occurred and the reason why it could not be processed immediately in the following form: **[TimeOccurred,Reason]**. Missed interrupts are represented by elements of a list maintained by instance variable **missed**, where each element has the form **[TimeOccurred,TimeLost,Reason]**. **TimeOccurred** is the time when the interrupt occurred, **TimeLost** is the time when the interrupt was deemed missed, and **Reason** is why the interrupt was missed. The following list outlines the missed interrupt reasons supported by *ProtoTool*:

- interrupts disabled — interrupts were disabled globally;

³⁶The interrupt can be resumed later, which effectively recreates it.

- this interrupt disabled — the interrupt in question was disabled individually;
- priority too low — the interrupt was not of sufficient priority to suspend the currently-running active object or interrupt handler;
- processing not done — a previous occurrence of the interrupt was still being processed;
- others pending — the interrupt could not be processed immediately and could not be made pending because other pending occurrences existed;
- no software object waiting — no interrupt task was awaiting notification;
- cleared by user — the user cleared the interrupt while it was pending.

Interrupt tasks which are waiting for an interrupt are kept as a list in instance variable `waiters`, each element of which contains the time when the interrupt task began waiting, and its object identifier. The format is `[TimeWaited, ObjectId]`.

Instance variable `isr` holds the class of the associated ISR and any run-time instance variables settings for it, in the following form: `[IsrClass, [IvarSettings]]`. When an interrupt is defined, this information is used to create its interrupt handler, the object identifier of which is stored in instance variable `handler`.

The information which defines an interrupt, such as the period of a periodic interrupt or the `[Object, Ivar]` structure for `ivarread` and `ivarwrite` interrupts, is stored in instance variable `definition`. This is set by the individual interrupt classes when the interrupt is defined.

5.2.5 Interrupt Implementation in *ProtoTool*

5.2.5.1 Queuing in *ProtoTool*

Figure 5.9 shows MLog's and *ProtoTool*'s queuing structure.

The *system queue* holds system objects which are ready to run. If the object at the front of the system queue has the same priority as the application object chosen to run³⁷, the system object will be selected to execute.

³⁷This is the object at the front of the interrupt ready-to-run queue if such exists. If not, it is the first object in the ready-to-run queue.

The *interrupt ready-to-run queue* is used to queue active objects which have been interrupted and interrupt handlers which are waiting to execute³⁸. When an active object is interrupted, it is put at the front of the interrupt ready-to-run queue so it will be chosen first when the interrupt's servicing is completed. Every active object in the interrupt ready-to-run queue has a higher logical priority than all active objects in the ready-to-run queue, regardless of their respective physical priorities.

Figure 5.9 MLog's and *ProtoTool*'s Queuing Structure

Application active objects which are ready to run but were not interrupted are put on the *ready-to-run queue* according to their priority, with the highest priority ones at the beginning. For instance, when an object is scheduled after receiving a message, it is put here.

As mentioned before, *ProtoTool* allows interrupt handlers to execute actions which cause them to become suspended, such as calling method `delay/2`. When an interrupt handler is suspended, MLog's virtual processor is considered to be running with *at least* the suspended handler's priority. A record of this priority must be kept to ensure that

³⁸System objects cannot be interrupted.

only higher priority active objects or higher priority interrupt handlers run while the handler in question is suspended. This is done using the *current application queue*. The currently-running active object or interrupt handler is always at the front of this queue. A “regular” active object remains on the current application queue until it suspends itself. If an interrupt handler suspends itself, it remains on the queue to establish the minimum execution priority until it is ready to run again. When the interrupt handler completes its processing, it is removed. When selecting an object to run, the MLog scheduler considers the priorities of the objects on the current application queue. Figure 5.10 shows an example and explanations of how the current application queue is managed.

Figure 5.10 Example Management of the Current Application Queue

Pending interrupts are stored in a queue called the *interrupt queue*. This queue is ordered by interrupt priority with the highest priority pending interrupt at the beginning. Thus, when determining if an interrupt should occur, searching it from the beginning automatically implements the highest-priority-first resolution scheme. The queue's entries are the object identifiers of the associated passive interrupt objects. If an interrupt has more than one occurrence pending, it appears only once in the interrupt queue as the list in instance variable `pending` reflects the number of pending occurrences. When

an interrupt's last pending occurrence is removed, the interrupt is dequeued from the interrupt queue.

Periodic, once, and repeat interrupts are recorded by MLog's virtual processor as appropriate entries on the delay queue as this is where all time-dependent events are stored. Thus, occurrences of timed interrupts are treated as timeouts of the associated passive interrupt object. When the delay entry is taken from the delay queue, a check is made to establish if the interrupt can be processed. If an interrupt's queuing mode is *timed* and it cannot be processed immediately, an entry is added to both the interrupt queue and the delay queue. If the interrupt is processed before the timeout expires, the delay queue entry is removed. If the timeout expires, the interrupt is no longer pending, so the interrupt queue entry is removed.

Ivarread and ivarwrite interrupts are recorded in the *ivarread queue* and the *ivarwrite queue* respectively. Two queues are used because although both can be termed accesses, reads and writes are logically different. Also, this avoids parameterizing each entry to include the access mode. Each queue contains entries which hold the object identifier of passive interrupt objects which are of that type. Both queues are priority queues with the highest priority at the beginning. Hence, if two or more interrupts apply to the same instance variable access, the highest priority one will occur. When an instance variable access occurs, the appropriate queue is searched. If a match is found, the associated interrupt is generated. If not, the application is not affected.

5.2.5.2 ProtoTool's Interrupt Occurrence Algorithm

The algorithm on the next page is used to determine if an interrupt handler should be scheduled to process an interrupt.

Lines 2 to 7 determine if previous occurrences of the interrupt will prevent this one from being processed immediately. Lines 10 and 11 determine if the interrupt is masked globally or individually. If there is no active object running currently, the interrupt is allowed to run by default as line 12 indicates.

```

1  can_occur = false

2  if interrupt being processed then reason = processing not done
3  else
4      if other occurrences pending then reason = others pending
5      else can_occur = true
6      endif
7  endif

8  if can_occur then
9      can_occur = false
10     if interrupts are enabled then
11         if this interrupt is enabled then
12             if virtual processor is idle then can_occur = true;
13         else
14             curr_prio = priority of current active object
15             intr_prio = priority of potential interrupt
16             if at least one interrupt handler on
17                 current application queue then
18                 if intr_prio > highest interrupt handler priority then
19                     can_occur = true;
20                 else can_occur = false;
21                 endif
22             else can_occur = true;
23             endif
24             if can_occur then
25                 can_occur = (intr_prio > curr_prio)
26                 if current active object is not an interrupt handler then
27                     can_occur = can_occur or
28                         in "interrupts have priority" mode
29                 endif
30                 if not can_occur then reason = priority too low
31                 endif
32             else reason = priority too low
33             endif
34         endif
35     else reason = this interrupt disabled
36     endif
37 else reason = interrupts disabled
38 endif
39 endif

40 if can_occur then schedule interrupt handler
41 else 'reason' contains why the interrupt cannot be processed
42 endif

```

Lines 16 to 23 check the current application queue to see if a suspended interrupt handler

has established a minimum execution priority. If so, the priority of the current interrupt is compared against it. Line 25 shows that the interrupt's priority must be greater than the priority of the currently-running active object if the interrupt is to be processed immediately. Lines 26 to 29 implement the check of the current interrupt priority interpretation. If this is *interrupts have priority*, the interrupt can occur as long as the currently-running active object is not an interrupt handler. If it is, only their relative priorities are considered.

5.2.5.3 Interrupt Example

Figure 5.11 The Temperature Gauge Interrupt Example

The following MLog application implements a concurrent system which interacts with its environment using interrupts. It prototypes a temperature monitor with three gauges. Each gauge is checked periodically, and if the temperature is greater than an upper limit or less than a lower limit, a flag is raised indicating an error. The developer

can increase or decrease the temperature by five degrees using the *Up* and *Down* buttons provided in the user interface. The design is shown above in figure 5.11.

Passive object **TempBuffer** stores, in instance variables, the current temperature and limits for each gauge. *Up* and *Down* button pushes update the corresponding values. The three temperature monitors, *Itask1*, *Itask2*, and *Itask3*, act as interrupt tasks by waiting for notification from the associated periodic interrupt. When notified, they retrieve the current temperature and update the user interface accordingly.

```
% *****
%
%   This Mlog code implements a small concurrent system which monitors three temperature
%   "gauges". Each gauge's temperature and status is output to a dedicated portion of a window-
%   based interface. Two buttons are included for each gauge, one which raises the temperature by
%   five degrees, and one which lowers it by the same amount. Each gauge has a hot and a cold
%   limit. When one of the limits is passed, an appropriate status is displayed. The status of
%   each gauge is checked periodically by a dedicated interrupt task which is notified when a
%   status-check interrupt occurs. This code can only be run in MLog's window mode.
%
% *****

% *****
%
% class: temperatureBuffer of object.
%
% description: - holds the current temperature and limits for each gauge
%
% *****

class: temperatureBuffer of object.

ivars: [temp1,temp2,temp3,
        hotLimit1,hotLimit2,hotLimit3,
        coldLimit1,coldLimit2,coldLimit3].

get(1,Value,HotLimit,ColdLimit) :-
    theValueOf(temp1,Value),
    theValueOf(hotLimit1,HotLimit),
    theValueOf(coldLimit1,ColdLimit).
get(2,Value,HotLimit,ColdLimit) :-
    theValueOf(temp2,Value),
    theValueOf(hotLimit2,HotLimit),
    theValueOf(coldLimit2,ColdLimit).
get(3,Value,HotLimit,ColdLimit) :-
    theValueOf(temp3,Value),
    theValueOf(hotLimit3,HotLimit),
    theValueOf(coldLimit3,ColdLimit).

up(1) :-
    theValueOf(temp1,Current),
    New is Current + 5,
    makeTheValueOf(temp1,New).
up(2) :-
    theValueOf(temp2,Current),
    New is Current + 5,
    makeTheValueOf(temp2,New).
up(3) :-
    theValueOf(temp3,Current),
    New is Current + 5,
    makeTheValueOf(temp3,New).
```

```

down(1) :-
    theValueOf(temp1,Current),
    New is Current - 5,
    makeTheValueOf(temp1,New).
down(2) :-
    theValueOf(temp2,Current),
    New is Current - 5,
    makeTheValueOf(temp2,New).
down(3) :-
    theValueOf(temp3,Current),
    New is Current - 5,
    makeTheValueOf(temp3,New).

% .....
%
% class: temperatureMonitor of object.
%
% description: - implements the interrupt tasks which are responsible for updating the
%               gauges' temperatures and status'
%               - waits for notification of a status-check interrupt, gets the necessary
%               data from the temperature buffer, and updates the display accordingly
% .....

class: temperatureMonitor of object.

ivars: [temp,intPriority,headerMsg,statusMsg,tempBuffer].

begin :-
    theValueOf(temp,Temp),
    theValueOf(intPriority,IntPriority),
    theValueOf(headerMsg,HeaderMsg),
    theValueOf(statusMsg,StatusMsg),
    theValueOf(tempBuffer,TempBuffer),
    repeat,
    waitForInterrupt(IntPriority),
    TempBuffer-get(Temp,Value,HotLimit,ColdLimit),
    updateTemp(Temp,Value,HeaderMsg),
    updateStatus(Value,HotLimit,ColdLimit,StatusMsg),
    fail.

updateTemp(Temp,Value,HeaderMsg) :-
    convert(Temp,string,TempString),
    convert(Value,string,ValueString),
    concatenate("Temperature ",TempString,String1),
    concatenate(String1," = ",String2),
    concatenate(String2,ValueString,String),
    HeaderMsg-redisplayMessage(String),!.

updateStatus(Value,HotLimit,_,StatusMsg) :-
    Value > HotLimit,
    StatusMsg-redisplayMessage("HOT"),!.
updateStatus(Value,_,ColdLimit,StatusMsg) :-
    Value < ColdLimit,
    StatusMsg-redisplayMessage("COLD"),!.
updateStatus(_,_,_,StatusMsg) :-
    StatusMsg-redisplayMessage("Ok").

% .....
%
% class: main of object.
%
% description: - creates the interrupts, the temperature buffer, and the interrupt tasks
%               - creates and displays the user interface
%               - acts as an idle task after creating things by continuously preempting
%               itself
% .....

```

```

class: main of object.

quit.

begin :-
    temperatureBuffer-new(TempBuffer, passive, {temp1=10, temp2=20, temp3=15,
        hotLimit1=40, hotLimit2=30, hotLimit3=20,
        coldLimit1=5, coldLimit2=10, coldLimit3=15}),
    createTemperatureScreen(TempBuffer, Frame, Header1Msg, Status1Msg, Header2Msg, Status2Msg,
        Header3Msg, Status3Msg),
    periodicInterrupt-new(Int1, passive, {priority=10, period=[0,0,2],
        isr={notifySoftwareIsr, {}}}),
    periodicInterrupt-new(Int2, passive, {priority=11, period=[0,0,3],
        isr={notifySoftwareIsr, {}}}),
    periodicInterrupt-new(Int3, passive, {priority=12, period=[0,0,4],
        isr={notifySoftwareIsr, {}}}),
    temperatureMonitor-new(ITask1, active, {temp=1, intPriority=10, headerMsg=Header1Msg,
        statusMsg=Status1Msg, tempBuffer=TempBuffer}),
    temperatureMonitor-new(ITask2, active, {temp=2, intPriority=11, headerMsg=Header2Msg,
        statusMsg=Status2Msg, tempBuffer=TempBuffer}),
    temperatureMonitor-new(ITask3, active, {temp=3, intPriority=12, headerMsg=Header3Msg,
        statusMsg=Status3Msg, tempBuffer=TempBuffer}),
    Frame-displayWindow,
    interrupt-enableInterrupts,
    Int1-enable,
    Int2-enable,
    Int3-enable,
    repeat,
    preempt,
    accept(quit),
    temperatureMonitor-removeInstance(ITask1),
    temperatureMonitor-removeInstance(ITask2),
    temperatureMonitor-removeInstance(ITask3),
    periodicInterrupt-removeInstance(Int1),
    periodicInterrupt-removeInstance(Int2),
    periodicInterrupt-removeInstance(Int3),
    temperatureBuffer-removeInstance(TempBuffer),
    frame-deleteWindow(Frame),
    frame-removeInstance(Frame), !.

createTemperatureScreen(TempBuffer, Frame, Header1Msg, Status1Msg, Header2Msg, Status2Msg,
    Header3Msg, Status3Msg) :-
    myself(MyId),
    frame-new(Frame, passive, {textFrame=true, height=1},
        width=420, xLocation=0, yLocation=0, label="Temperature Monitor"),
    panel-new(Temp1Panel, passive, {owner=Frame, percentHeight=33}),
    message-new(Header1Msg, passive, {owner=Temp1Panel, xLocation=5, yLocation=5,
        labelString="Temperature 1 ="}),
    message-new(Status1Msg, passive, {owner=Temp1Panel, xLocation=180, yLocation=5,
        labelString=""}),
    button-new(_, passive, {owner=Temp1Panel, xLocation=230, yLocation=5,
        labelImage="Up", buttonwidth=4, eventMessage=(TempBuffer-up(1)S ())}),
    button-new(_, passive, {owner=Temp1Panel, xLocation=275, yLocation=5,
        labelImage="Down", buttonwidth=4, eventMessage=(TempBuffer-down(1))}),
    button-new(_, passive, {owner=Temp1Panel, xLocation=345, yLocation=5,
        labelImage="Quit", buttonwidth=4, eventMessage=(MyId-quit)}),
    panel-new(Temp2Panel, passive, {owner=Frame, percentHeight=33}),
    message-new(Header2Msg, passive, {owner=Temp2Panel, xLocation=5, yLocation=5,
        labelString="Temperature 2 ="}),
    message-new(Status2Msg, passive, {owner=Temp2Panel, xLocation=180, yLocation=5,
        labelString=""}),
    button-new(_, passive, {owner=Temp2Panel, xLocation=230, yLocation=5,
        labelImage="Up", buttonwidth=4, eventMessage=(TempBuffer-up(2)S ())}),
    button-new(_, passive, {owner=Temp2Panel, xLocation=275, yLocation=5,
        labelImage="Down", buttonwidth=4, eventMessage=(TempBuffer-down(2))}),
    button-new(_, passive, {owner=Temp2Panel, xLocation=345, yLocation=5,
        labelImage="Quit", buttonwidth=4, eventMessage=(MyId-quit)}),
    panel-new(Temp3Panel, passive, {owner=Frame, percentHeight=34}),
    message-new(Header3Msg, passive, {owner=Temp3Panel, xLocation=5, yLocation=5,
        labelString="Temperature 3 ="}),
    message-new(Status3Msg, passive, {owner=Temp3Panel, xLocation=180, yLocation=5,
        labelString=""}),
    button-new(_, passive, {owner=Temp3Panel, xLocation=230, yLocation=5,
        labelImage="Up", buttonwidth=4, eventMessage=(TempBuffer-up(3)S ())}),
    button-new(_, passive, {owner=Temp3Panel, xLocation=275, yLocation=5,
        labelImage="Down", buttonwidth=4, eventMessage=(TempBuffer-down(3))}),
    button-new(_, passive, {owner=Temp3Panel, xLocation=345, yLocation=5,
        labelImage="Quit", buttonwidth=4, eventMessage=(MyId-quit)}}).

```

Chapter 6

ProtoTool

Having discussed *ProtoTool's* models and implementation of timing constructs and interrupts in the preceding two chapters, we now turn our attention to *ProtoTool* itself. *ProtoTool* provides the facilities for controlling prototype execution and for gathering and presenting information about concurrent system designs.

6.1 Software Structure

As figure 6.1 shows, *ProtoTool's* capabilities are provided through *prototool entries* in the MLog engine. Entries either perform an action which controls a prototype's execution, or obtain some information for presentation³⁹.

If MLog is being run in terminal mode⁴⁰, developers, through MLog consoles, or components of applications make requests of *ProtoTool* by calling the corresponding *prototool entries*. In this mode, data is output to MLog consoles when appropriate.

If MLog's window mode is employed, the *prototool entries* are augmented with a system object, `ProtoTool`, and a window-based user interface⁴¹. When an event occurs for which the developer should receive notification, the MLog engine puts a message containing the event in `ProtoTool's` message queue and schedules it to run. When `ProtoTool` executes, the *prototool entry* associated with the event is called, the necessary data is obtained, and the user interface is updated. `ProtoTool` has a very high priority

³⁹Figure 6.1 shows the two types of *prototool entries*, *execution control* and *data retrieval*. There are actually numerous methods of each type.

⁴⁰In terminal mode, no windowing capabilities exist. The user interface to MLog is a line-by-line, character-based console.

⁴¹Application components can still make requests of *prototool entries* if MLog is running in windowing mode.

to ensure the effects of events are seen as soon as possible after they occur. When a developer makes a request of ProtoTool via the window-based interface, the request is sent to **ProtoTool** which then references the appropriate prototool entry of the MLog engine. **ProtoTool** does not initiate interactions with any active objects. It obtains data from and modifies prototype execution through passive objects within the MLog engine. Most of the prototool entries reside in class **object**. Any interrupt-related calls are made to either class **interrupt** for global interrupt information or to the passive interrupt objects for individual interrupt information.

Figure 6.1 *ProtoTool's* Structure

An example of *ProtoTool's* window-based user interface is shown in figure 6.2. The functions of the components will be described in the following sections.

<input type="button" value="Speed"/> <input type="button" value="Timeslice"/>		<input type="button" value="Absolute"/> <input type="button" value="Forward"/> <input type="button" value="Backward"/>		Simulation Mode: <input type="text"/>	
Speed: 100 Timeslice: 0 Speedwindow		Time: [0,0,0] Time window		interactive	
Prampwindow Simulationmodewindow					
Status System: stopped window Application: released		<input type="button" value="Gonow"/> <input type="button" value="Gonext"/> <input type="button" value="Hold"/>		<input type="button" value="display time lines"/> <input type="button" value="record"/> <input type="button" value="TimeInnewwindow"/>	
Interrupts: enabled <input type="button" value="disabled"/> Interruptwindow Interrupts have priority: <input type="text"/> false		Interrupt Selection: <input type="text"/> -1 -1 -1 -1 -1 -1 -1 -1 -1			
<input type="button" value="Enabled"/>	<input type="button" value="Suspended"/>	<input type="button" value="ClearAll"/>	<input type="button" value="Enable"/>	<input type="button" value="Suspend"/>	<input type="button" value="Clear"/>
<input type="button" value="Disabled"/>	<input type="button" value="Resumed"/>	<input type="button" value="Pending"/>	<input type="button" value="Disable"/>	<input type="button" value="Resume"/>	<input type="button" value="Display"/>
		<input type="button" value="Generate"/>		<input type="button" value="Priority"/>	
Next-objectwindow					
Information-display window					

Figure 6.2 The ProtoTool Display

6.2 Execution Control

6.2.1 Simulation Modes

ProtoTool supports two simulation modes: (1) *free-running* and (2) *interactive*. The current mode is highlighted by the 'Simulation Mode' choice⁴² in the simulation mode window.

In free-running mode, *ProtoTool* keeps the application running until no active objects are ready to execute and no timeouts are pending. If no application active objects are able to run but the delay queue is not empty, *ProtoTool* automatically advances the system time to that of the first timeout and processes it. Thus, control is returned to the developer only when the application terminates or when some form of input is required. This execution mode is necessary because MLog does not contain an idle task⁴³. Without such, the time must be advanced explicitly because no messaging is done unless an active object is running⁴⁴. Under this mode, prototypes can take less time to run because no unnecessary waiting is done. Free-running mode is useful for making an initial analysis of a design.

Interactive mode relies more on the developer. *ProtoTool* returns control to the developer whenever there are no application active objects ready to run, regardless of any timeouts on the delay queue. If pending timeouts exist, information about the first one is provided to assist the developer in deciding what action to take. For instance, the time could be modified or interrupts could be manipulated. Interactive mode permits probing of and experimentation with a design and therefore is suitable for detailed analyses. This mode allows developers to step through an application based on time in a way similar to instruction single-stepping which is provided by some debuggers.

6.2.2 Execution Suspension and Resumption

Running in interactive mode allows developers to modify a prototype's execution. However, this must be done at times dictated by the application. To permit developers

⁴²A choice is a windowing construct where only one element of a set can be selected at a time. In *ProtoTool*, the selected element is highlighted in reverse video.

⁴³An idle task is used to consume processor time. It is the lowest priority task in a system and runs only when no other tasks are able.

⁴⁴Recall that MLog simulates clock ticks with messages.

to suspend an application at any time, the *Stop/Hold* button is available in the status window⁴⁵. When pushed with an active object executing, control is returned immediately to the developer and the application is frozen in its current state. The identity of the application active object running at the time is displayed to give the developer an indication of the application's progress⁴⁶. When the application is not running, the *Stop/Hold* button's label is set to *Hold* to reflect that pushing it simply suspends the application in its current state. MLog's debugger, DMLog [Lien89], is equipped with a *Stop* button which has the same effect as *ProtoTool's Stop* button, and has different facilities for resuming execution. Each tool can suspend and resume execution independently of the other, even when both are being used at the same time.

A suspended application can be resumed in two ways. Using the *Gonow* button, scheduling is started at the current time. If no application active objects are ready to run and no timeouts are to expire at the current time, nothing happens. This function is most useful after an application has been suspended with the *Stop/Hold* button. If the *Gonext* button is pushed and at least one application active object is ready to run, scheduling begins at the current time. If no such object exists, the time is advanced to that of the first pending timeout if the delay queue is not empty. If it is, nothing happens. When running in interactive mode, the *Gonext* button can be used to advance the time, if required, to resume the application.

6.2.3 Virtual Processor Configuration

The speed of MLog's virtual processor, expressed in messages per second, can be modified by pushing the *Speed* button in the speed window and entering the new value in the prompt window. There is no upper limit on the speed. One message per second is the theoretical lower limit, but a practical lower limit exists because of the overhead associated with message evaluation. A lower limit of fifty is suggested. The *Timeslice* button is available to change the time slice duration, specified as a message count, where zero means MLog's run-til-block scheduling policy is to be used. A non-zero value represents each time slice's duration.

⁴⁵If running in MLog's terminal mode, the application cannot be stopped this way.

⁴⁶If a system object is running when the *Stop* button is pushed, the interrupt ready-to-run queue and the ready-to-run queue are examined to determine the "currently-running" application active object.

The *Forward* and *Backward* buttons in the time window permit altering the system time by a relative amount in the indicated direction. The *Absolute* button sets the time to the value entered in the prompt window.

6.2.4 Time Lines

The time line window contains two choices⁴⁷. The left choice indicates if time lines are being displayed in real time, while the right one shows if the concurrent system events are being stored in a file. If the latter's value is 'record', the name of the file appears after the 'File:' prompt.

6.2.5 Interrupts

The 'Interrupts' choice in the interrupt window displays 'enabled' when interrupts are enabled globally and 'disabled' when they are disabled globally. The current interrupt priority interpretation is shown in the 'Interrupts have priority' choice. When 'true' is selected, the *interrupts have priority* interpretation is in use. The interpretation is *software has priority* if 'false' is highlighted. The currently-defined interrupt priorities are displayed in increasing numerical order in the 'Interrupt Selection' choice, where a priority of '-1' represents an undefined interrupt. *ProtoTool* allows a maximum of ten interrupts to be defined. The following buttons in the interrupt window can be used to control a prototype's execution:

- Enable — enables the selected interrupt;
- Disable — disables the selected interrupt;
- Suspend — suspends the selected interrupt;
- Resume — resumes the selected interrupt;
- Clear — removes any pending occurrences of the selected interrupt;
- Definition⁴⁸ — prompts the developer to enter, for the selected interrupt, a new value for the `definition` instance variable⁴⁹;

⁴⁷See chapter seven for details about *TimeLineTool* and time lines.

⁴⁸This button's label changes to reflect the type of interrupt selected. For instance, if a periodic interrupt is selected, the label is "Period". If the selected interrupt is undefined, the label is as shown in figure 6.2.

⁴⁹For instance, if an `ivarread` interrupt is selected, the developer is prompted for a new `{Object,Ivar}` structure.

- **Priority** — prompts the developer for a new priority for the selected interrupt;
- **Queuing** — prompts the developer for a new queuing mode for the selected interrupt;
- **Generate** — creates an occurrence of the selected interrupt and transfers control to the MLog engine to attempt to service it;
- **ClearAll** — removes any pending occurrences of all defined interrupts.

6.3 Information Presentation

When running in window mode, much of the information maintained by *ProtoTool* is displayed continuously in the window-based user interface. The speed and time slice of MLog's virtual processor are shown in the speed window, while the time window contains the current time. The choices in the simulation mode, time line, and interrupt windows highlight their selected values in reverse video and are therefore easy to assimilate.

In terminal mode, the information which is continuously displayed in the window-based user interface must be obtained by calling the corresponding data retrieval protocol entries. The data is then output to the MLog console which requested it.

6.3.1 System and Application Status

The status window shows the current status of the application after the 'Application:' prompt and displays the status of the MLog engine following the 'System:' prompt. An application status of 'released' indicates that the application is able to run. If this status is 'held', the application has been frozen by a push of the *Stop/Hold* button. This results in the object selection process shown in figure 5.9 ignoring the interrupt ready-to-run queue and the ready-to-run queue. When the MLog engine is running either a system object or an application active object, the system status is 'running'. When no active objects of any type are ready to execute, this status is set to 'stopped'.

In terminal mode, the status of MLog's virtual processor is conveyed by the state of the main console. When it is prompting the user for input, the system is stopped. When it is processing a command, either a system object or an application active object is running.

2

of/de

2

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

6.3.2 Display of the Next Object

In either window or terminal mode, when the MLog engine stops, *ProtoTool* displays the identity of the next application object to execute. In terminal mode, this information is output to the main MLog console, while in window mode, the next object window shows the data. The current time is displayed along with the identity of the object and any associated data. The following examples show and explain the possible situations:

- **At [0,0,4]: No application objects able to execute**
at time [0,0,4], nothing in the application is ready to run;
- **At [0,0,5]: Object ready to run: `server1(#server)`**
at time [0,0,5], object `server1` of class `server` is the next application active object to execute;
- **At [0,0,8]: Object ready to run: ISR for interrupt priority 10**
at time [0,0,8], the next application active object to run is the interrupt handler for the interrupt whose priority is ten;
- **At [1,2,3]: Next application event: Will happen at [1,2,8]**
Delay timeout of: `server1(#server)`
at time [1,2,3], no application active objects are ready to run — at time [1,2,8], object `server1` of class `server` will return from a call to method `delay/2`;
- **At [2,15,18]: Next application event: Will happen at [2,17,22]**
Send timeout of: `#123(#client)`
at time [2,15,18], no application active objects are ready to run — at time [2,17,22], a timed send by object `#123` of class `client` will timeout;
- **At [0,1,2]: Next application event: Will happen at [0,1,3]**
Accept timeout of: `#123(#client)`
at time [0,1,2], no application active objects are ready to run — at time [0,1,3], a timed accept by object `#123` of class `client` will timeout;
- **At [4,5,6]: Next application event: Will happen at [5,10,20]**
Wait interrupt timeout of: `#110(#itask)`
at time [4,5,6], no application active objects are ready to run — at time [5,10,20], a timed wait for an interrupt by object `#110` of class `itask` will timeout;

- **At [10,20,30]: Next application event: Will happen at [10,25,35]**
Interaction timeout of: bob(#person)
 at time [10,20,30], no application active objects are ready to run — at time [10,25,35], a timed wait for an interaction by object bob of class person will timeout;
- **At [0,1,0]: Next application event: Will happen at [0,1,5]**
Interrupt of priority 10, Can interrupt
 at time [0,1,0], no application active objects are ready to run — at time [0,1,5], an interrupt of priority ten will occur, and given the current situation⁵⁰, it will be serviced;
- **At [1,1,1]: Next application event: Will happen at [2,2,2]**
Interrupt of priority 10, Can't interrupt: Interrupts disabled
 at time [1,1,1], no application active objects are ready to run — at time [2,2,2], an interrupt of priority ten will occur, and given the current situation, it cannot be serviced because interrupts are disabled globally;
- **At [3,4,5]: Next application event: Will happen at [3,4,59]**
Interrupt timeout priority 15
 at time [3,4,5], no application active objects are ready to run — at time [3,4,59], the timeout will occur of a pending occurrence of an interrupt whose queuing mode is timed and whose priority is fifteen;

If the virtual processor is being held by an interrupt handler⁵¹, the following is an example of the line which will appear in addition to the ones described above:

Processor held by: ISR for interrupt priority 10

In window mode, the information in the next object window can be saved to a file as a record of how execution progressed.

6.3.3 Other Information

Interrupt information, requested by pushing buttons in the interrupt window, is displayed in the information display window if *ProtoTool's* window-based user interface is in use. Also, any errors *ProtoTool* detects while running are output to the information display

⁵⁰Because the developer can manipulate interrupts, this situation may change.

⁵¹See chapter five for details.

window. For instance, if the process running *TimeLineTool* terminates prematurely, an appropriate error message is displayed there. In terminal mode, any requested information is output to the MLog console from which the request originated.

Chapter 7

TimeLineTool

7.1 Motivation and Overview

The most critical aspect of many concurrent systems is the way their components interact. These systems are sometimes referred to as *behaviour-intensive* because when and how interactions occur can be as important as the functions of the components involved. In addition, many concurrent systems interact with their environment using interrupts. This can have a profound effect on the temporal characteristics of the application.

When evaluating a concurrent system design, a developer needs to know the relative order in which events occur. For instance, to determine the cause of a deadlock, it may be beneficial to know the order in which messages were processed by a particular task. This can be done by inserting print statements at appropriate spots in the application's code, such as "Task A received message 'start' from task B". However, textual output of this nature tends to be difficult to understand. In particular, it is hard to visualize how a single event fits into the overall behaviour of the application. One solution is to take advantage of developers' pattern recognition skills as it has been said often that a picture is worth a thousand words.

TimeLineTool takes the events generated during the execution of a concurrent system and illustrates them using a syntactically-shallow, semantically-deep graphical interface. The events pertaining to each component in an application are displayed in the component's *time line* or *execution thread*. Combining the time lines of all components into one display makes it very easy to see and analyze the temporal behaviour of the system. Because it provides *pan* and *zoom* capabilities, *TimeLineTool* allows developers to examine timing information over a large time frame, or to focus on fine-grain details. If *TimeLineTool* is running in *real-time mode*, time lines are displayed as events occur so developers get a graphical view of their application as it executes. In *file mode*, events

previously recorded in a file are extracted and displayed to form a replay. This mode permits detailed, after-the-fact analyses.

The current version of *TimeLineTool* is targetted at systems which use message passing for task synchronization. It does not support events associated with mailbox-based or semaphore-based run-time systems⁵². Further, only uniprocessor applications are applicable as all tasks are assumed to be on the same node, and message delays are not considered.

7.2 Event Generation

It was suggested above that print statements could be used to generate textual execution traces. This is undesirable because should output require an interaction with another task, the application's behaviour would be altered. Regardless of whether such an interaction occurs, performing output consumes processing time. Thus, developers would not see the true temporal behaviour. Using the underlying run-time system to generate trace data alleviates this problem provided doing so is completely transparent to applications. This is difficult in production systems because the system time is normally bound to the "real world" time. *ProtoTool*, however, can do this because it maintains a virtual clock which is distinct from the "real world" clock. Hence, *ProtoTool* can generate trace data without affecting applications in any way. *TimeLineTool* makes no assumptions about how events are generated. It simply displays them as is without doing any semantics verification.

7.3 The Display

Figure 7.1 shows the *TimeLineTool* display. The top portion is the control panel containing the facilities which allow developers to alter the display. The bottom section is a window showing a part of the canvas on which the time lines are drawn. To permit modification of the display resolution, the horizontal time scale is managed by *TimeLineTool*. Otherwise, the displayable time range would be restricted by the maximum width of a canvas. Thus, the window's width is the same as that of the canvas. The canvas's height is greater than

⁵²However, it can be used with mailbox-based and semaphore-based run-time systems if the mailboxes and semaphores are built from tasks.

Figure 7.1 The *TimeLineTool* Display

the window's so the number of tasks which can be shown is not restricted by the size of the overall display. The scroll bar at the left of the window permits vertical movement.

Region A is where the time lines are drawn. The vertical lines serve as *time guides* to assist establishing when an event occurred. The time intervals currently displayed appear in region B. New times are shown whenever a request is made to move outside the current time range. Region C contains the names of the tasks. Task names are restricted to twenty characters, the composition of which is determined by the run-time system generating the events.

This display is divided vertically into *slots*, where each task occupies one slot. The slots have been made large enough so that the events of adjacent slots can be differentiated. The number of slots, and hence the height of the canvas, is a configuration parameter which cannot be modified at run time. The first slot is reserved for events which are global to the system, such as enabling and disabling interrupts. Once a slot has been allocated to a task, the slot cannot be reused, even if the task terminates. An integer is used to identify tasks. It is the responsibility of the run-time system generating the events to establish these unique task identifiers.

7.4 Display Modes

7.4.1 Real-Time Mode

In real-time mode, *TimeLineTool* executes as a separate process, independent of the event-generating run-time system, which receives its data from a Unix pipe as shown in figure 7.2.

Figure 7.2 *TimeLineTool* in Real-Time Mode

This mode provides instant feedback to developers. Using *ProtoTool*, the application can be stopped, its time lines can be examined and any necessary program or environment

changes can be made. This should decrease significantly the amount of time required to achieve a good design because learning is done "on the fly".

When a new event is received, *TimeLineTool* stores it in the appropriate slot and then displays it. The event is stored so that it can be viewed later if desired. For example, it may be necessary to determine how a sequence of events was triggered, which requires looking back in time. This can be done with the application stopped.

7.4.2 File Mode

File mode is designed to allow the behaviour of an application to be shown after it has finished executing. For example, the time lines could be shown at a design review or during a debugging session. Figure 7.3 illustrates the configuration.

Figure 7.3 *TimeLineTool* in File Mode

An interesting use of this mode is as a verification vehicle during evolution. When a system is installed, its events can be stored to represent the initial, presumably-correct, behaviour. Later, after changes have been made, a new event file can be generated. The *TimeLineTool* displays can then be compared visually to reveal any differences.

7.5 Display Resolution

The smallest resolution currently supported is one microsecond. All times are specified as a number of microseconds, and are stored as a floating point value. The resolution is reflected by the number of microseconds between adjacent time guides⁵³. In the display of figure 7.1, the resolution is one millisecond.

⁵³The time between two adjacent time guides represents one *time unit*.

Pushing one of the resolution buttons in the control panel changes the resolution to that value⁵⁴. For more precise control, the magnification slider can be used to pan or zoom. Selecting a positive value increases the resolution by a factor equal to the selected value, thus performing a zoom. Negative values effect a pan operation. Successive pans and zooms are cumulative, so that, for example, two consecutive zooms by a factor of two with an initial resolution of twelve milliseconds would result in a final resolution of three milliseconds. Slider values of minus one, zero, and one have no effect on the resolution. The basis for an adjustment is the left-most time guide. The time there remains the same, while the others change appropriately.

7.6 Display Movement

The *Start* and *End* buttons set the current time⁵⁵ to that of the first and last recorded event respectively. The *Step >>* button shifts the display ahead one time unit, while the *Step <<* button moves the display back one time unit. The *Forward* and *Backward* buttons work in conjunction with the *Step Size* choice, adjusting the display ahead or back, respectively, by the number of time units chosen.

For any forward movement, if the last event would be shifted off the display, the current time is set to the last event's time. If the first event would be moved off the canvas by a backward movement, the first event's time becomes the current time.

7.7 Colour and Monochrome Displays

TimeLineTool detects automatically the capabilities of its display device. If the monitor supports colour, all lines have the same thickness and texture, while the colour contains the semantics. For example, red is used when a task is waiting, yellow indicates a task is ready to run, and green identifies the running task. For monochrome monitors, different line thicknesses and textures are used. For instance, a dashed line represents waiting, a thin solid line indicates a task is ready to run, and a thick solid line is reserved for the running task.

⁵⁴For example, pushing the *1msec* button makes the resolution 60,000,000 microseconds.

⁵⁵The current time is the time of the left-most time guide.

7.8 Supported Events

This section gives a brief outline of each of the concurrent system events currently supported by *TimeLineTool*. Each figure contains events which are logically related. Although not given, each event has a unique integer identifier which *TimeLineTool* uses internally.

Figure 7.4 shows the events associated with creating and deleting components. The component at the tail of the arrow is the one performing the creation or deletion. If the action is not performed by the application, the arrow's tail is left unattached⁵⁶.

Symbolic Name	Meaning	Monochrome Display	Colour Display
TASK_CREATED and INTERRUPT_DEFINED	The specified task has been created, or the given interrupt is now valid		colour = green
TASK_REMOVED and INTERRUPT_UNDEFINED	The specified task has terminated, or the given interrupt is no longer valid		colour = red

Figure 7.4 Creation and Deletion Events

Figure 7.5 illustrates the events which reflect the activity status of tasks. For *TimeLineTool*, the execution of an ISR is considered the same as that of a task.

The events pertaining to synchronous interactions are displayed in figure 7.6. When a task initiates a synchronous interaction, its time line is shifted to be close to the receiver to indicate it is waiting for the receiver to process the message. If more than one sender is waiting for the same receiver, they all wait along the same horizontal line. When the interaction completes, the time line is shifted to its original position, and the task is then ready to run.

⁵⁶For example, in MLog, the initial active object is created by MLog's virtual processor, not by the application.

Symbolic Name	Meaning	Monochrome Display	Colour Display
TASK_RUNNING and ISR_RUNNING	The specified task or ISR is running		—— colour = green
TASK_READY and ISR_READY	The specified task or ISR is ready to run		—— colour = yellow
TASK_WAITING and ISR_WAITING	The specified task or ISR is waiting for an interaction		—— colour = red

Figure 7.5 Activity Status Events

Symbolic Name	Meaning	Monochrome Display	Colour Display
SYNC_INTERACTION_INITIATED	A rendezvous has been initiated
SYNC_INTERACTION_ESTABLISHED	The receiver of the message has begun to process it
SYNC_INTERACTION_FAILED	The receiver of the message put it back on the message queue
SYNC_INTERACTION_COMPLETED	The rendezvous has completed successfully

Figure 7.6 Synchronous Interaction Events

Although very few run-time systems implement them, *TimeLineTool* supports the anonymous synchronous interaction events shown in figure 7.7. When such an interaction is initiated, a vertical line is drawn to indicate that an interaction has been started. However, because the identity of the receiver is unknown, only some descriptive text is given⁵⁷, and the task is shown waiting “on its own ground” rather than at a receiver. When the interaction is completed, an alternative approach would involve removing the anonymous interaction events and replacing them with the corresponding “regular” synchronous events because the identity of the receiver is now known. This is not done because it would be impossible to determine if a synchronous interaction was initially anonymous, thus hiding useful information.

Symbolic Name	Meaning	Monochrome Display	Colour Display
ANON_SYNC_INTERACTION_INITIATED	A rendezvous has been initiated with an as yet undetermined receiver
ANON_SYNC_INTERACTION_ESTABLISHED	A task has begun to process a message which was sent anonymously
ANON_SYNC_INTERACTION_FAILED	The receiver of an anonymous message put it back on the message queue
ANON_SYNC_INTERACTION_COMPLETED	A rendezvous involving an anonymous message has been completed successfully

Figure 7.7 Anonymous Synchronous Interaction Events

⁵⁷In MLog, this is the class to which the message was sent.

Symbolic Name	Meaning	Monochrome Display	Colour Display
ASYNC_INTERACTION	An asynchronous message has been sent
ANON_ASYNC_INTERACTION_INITIATED	An anonymous asynchronous message has been sent

Figure 7.8 Asynchronous Interaction Events

The asynchronous interaction events illustrated in figure 7.8 are similar to the synchronous events described in figure 7.6 except that the sender does not become blocked. It simply drops off the message and continues executing.

The icons used to display the enabling and disabling of interrupts are shown in figure 7.9. If the action is performed by a task in the *TimeLineTool* display, an arrow is drawn from the task's time line to the associated icon. If not, such as when interrupts are enabled or disabled using *ProtoTool*, the icon appears alone. If interrupts are being modified globally, the icon appears in the slot reserved for system events.

Symbolic Name	Meaning	Monochrome Display	Colour Display
INTERRUPT_ENABLED and INTERRUPTS_NOW_ENABLED	An individual interrupt was enabled or all interrupts were enabled
INTERRUPT_DISABLED and INTERRUPTS_NOW_DISABLED	An individual interrupt was disabled or all interrupts were disabled

Figure 7.9 Interrupt-Specific Events

The remainder of the supported events are shown in figure 7.10. As with the interrupt-specific icons above, an arrow is drawn to the icon from the time line of the task

performing the operation, if appropriate. Events which are global to the system, such as **PROCESSOR_INFORMATION_CHANGE**, appear in the slot reserved for system events.

Symbolic Name	Meaning	Monochrome Display	Colour Display
TASK_INFORMATION_CHANGE	A task's details, such as priority, have been changed		brown
INTERRUPT_MISSED	An interrupt has been missed		red
TASK_SCHEDULING_CHANGE	The run-time system's scheduling policy has been changed		purple
INTERRUPT_SCHEDULING_CHANGE	The way interrupts interact with the software has been changed		purple
PROCESSOR_INFORMATION_CHANGE	Details of the run-time system, such as its speed, have changed		purple
NEW_TIME	The time has advanced without an event occurring	none	none
QUIT_DISPLAY	The display is to be terminated	none	none

Figure 7.10 Miscellaneous Events

7.9 Implementation

7.9.1 Software Structure

The software structure of *TimeLineTool* is shown in figure 7.11. Module **TLCMD** is the program interface. It is called by command if in file mode or is created as a process if in real-time mode. **TLCOMMON** contains routines which are common to all operation modes and display devices. Module **TLDRAW** is responsible for actually drawing events on the

display device. Modules **TLFILE** and **TLRT** contain, respectively, code which is specific to the file and real-time modes.

TLCMD parses any command arguments, determines which operation mode is in use, and passes control to either **TLFILE** or **TLRT**. These modules then read events and access **TLCOMMON** to process them. When events are to be drawn, **TLCOMMON** calls **TLDRAW**.

Figure 7.11 *TimeLineTool's* Software Structure

7.9.2 Event Data Format

Events are stored one per physical line in a file and are passed one per logical line through the pipe. The individual components of events, such as the event identifier and the time, are separated by commas. Events have a variable number of components, so only those which are required are stored or passed. Using a fixed data format would make parsing events simpler, but the number of components of the largest event would have to be present in each line which would be wasteful. The following example illustrates some of the events generated by the client-server program given in chapter four⁵⁸.

⁵⁸The comments on the right are not included in the actual output.

```

0,0,#60{#main} ,52,60      % at time 0, object 52 created object 60
                             %   with name #60{#main}
3,0,60,3                   % at time 0, object 60 was ready to run
2,0,60,-1                  % at time 0, object 60 started running
0,80000,#61{#client} ,60,61 % at time 80000, object 60 created
                             %   object 61 with name #61{#client}
3,80000,61,3               % at time 80000, object 61 was ready to
                             %   run

```

7.9.3 Event Data Reading Algorithms

The following algorithm is used when reading events from a file when in file mode:

```

while there are more events to read
  read next event from file and obtain event type
  if data format is correct for this event type then
    ok = false
    if task has been allocated a slot already then
      store this event in linked list of events
      ok = true
    else
=>   if there is room for another task then
=>     allocate a slot for the new task
=>     store this event in linked list of events
=>     ok = true
=>   endif
    endif
    if ok then
      if this is the first event then
        first_event = last_event = this event
      else
        if this event's time > last_event's time then
          last_event = this event
        else
          if this event's time < first_event's time then
            first_event = this event
          endif
        endif
      endif
    endif
  endif
endwhile

begin displaying time lines starting at first_event's time

```

When in real-time mode, the following algorithm is used to read events:

```

forever
  wait for event from pipe and obtain event type
  if data format is correct for this event type then
    ok = false
    if task has been allocated a slot already then
      store this event in linked list of events
      ok = true
    else
=>   if there is room for another task then
=>     allocate a slot for the new task
=>     store this event in linked list of events
=>     ok = true
=>   endif
  endif
  if ok then
    if this is the first event then
      first_event = last_event = this event
    else
      if this event's time > last_event's time then
        last_event = this event
      else
        if this event's time < first_event's time then
          first_event = this event
        endif
      endif
    endif
    if this event's time > last time currently displayed then
      make the left-most time guide's time this event's time
      redraw time lines with new current time
    else
      redraw time lines showing this event
    endif
  endif
endif
endforever

```

In both algorithms, the code indicated by the =>s shows that *TimeLineTool* can add tasks "on the fly" without receiving a `TASK_CREATED` event. If this occurs, the task is given the name 'task_#' where '#' is replaced by the task identifier in the event in question.

Chapter 8

Evaluation of *ProtoTool* and *TimeLineTool* Using The Lift Control Problem

8.1 The Lift Control Problem

The problem chosen to evaluate *ProtoTool* and *TimeLineTool* is that of a lift control system containing three lifts in a building with five floors. The following specifications must be met:

- each lift has a set of buttons, one for each floor, which illuminate when pushed and cause the associated lift to stop at the corresponding floor: the illumination is cancelled when the floor is visited;
- each floor has two buttons⁵⁹, one to request upward movement and one to request downward movement, which illuminate when pushed and are turned off when a lift stops at the floor and is either travelling in the desired direction or has no outstanding requests;
- when a lift has no requests to service, it should remain at its final destination floor and await further requests;
- all requests from floors for lifts, called *floor requests*, must be serviced eventually, with all floors given equal priority;
- all requests within lifts for floors, called *lift requests*, must be serviced eventually, with floors being serviced sequentially in the direction of travel.

The lift control problem contains all the elements of concurrent systems, including: (1) concurrent execution of tasks⁶⁰ — all lifts can be moving simultaneously, meaning each should be modeled as a distinct task, (2) intertask communication — the lifts may

⁵⁹The top and bottom floors have one only.

⁶⁰This would probably be real concurrency as the lifts are separate physically as well as logically.

be required to exchange information such as their current location and destination, and (3) environment interaction using interrupts — the most natural way to model all button pushes is as interrupts.

8.2 The Lift Control System Design

Figure 8.1 shows the design of the selected solution.

Active object **main** is created by the MLog engine and is used to: (1) create and destroy the remaining objects, and (2) process simulation control button pushes. When termination is desired, **main** removes all objects and returns control to MLog. A **reset** option has been included so the developer can start over at any time. The **reset** method of each object returns the object to its initial state.

Active object **liftCont** is the system's central controller. It receives floor request interrupts at its **floorButton/1** method and assigns them to one of the lifts by calling the associated lift manager's **floorRequest/1** method after determining the best lift for the request. If no lift is able to handle the request, the request is added to a queue of unallocated requests. When a lift manager reports its status to the central controller, each unallocated request is checked to see if it can now be processed by the reporting lift.

The lift managers are responsible for managing the activities of the associated lift. Lift requests are received at the **liftButton/1** method from interrupt **liftInt**, and new floor requests from the central controller enter at method **floorRequest/1**. Each time a lift's status changes, such as when it reaches the next floor when moving, it calls the **report/2** method of **liftCont** in order to keep the central controller up to date. This results in each lift's status existing in both the associated lift manager and the central controller. If the status' are maintained only by the lift managers, then the central controller would have to obtain each one every time a floor request is processed. This would generate considerable message traffic. By making the message to **report/2** synchronous, lifts are unable to take any action until the central controller has their current status. Thus, there is no possibility of data inconsistency.

Figure 8.1 The Lift Control System Design

The lift managers contain agendas implemented as the state machine shown in figure 8.2. The lift is in state *idle* when it has no floor requests or lift requests to process. When moving to handle floor requests or lift requests, the lift is in state *moving*. When a floor is reached which has an outstanding request, state *servicing* is entered. If only lift requests exist there, the next transition will be either to state *moving* or to state *idle*. If a floor request exists, state *waiting* is entered. When a valid lift button is pushed, a

transition is made to state *moving*. State *idle* is entered whenever the lift manager is reset, regardless of the current state.

Figure 8.2 The State Machine Agenda of Lift Managers

The floor and lift interrupts have the same form. Each is an *ivarread* interrupt⁶¹ because these interrupts are generated under user control⁶². However, no instance variable is read to generate an interrupt. Instead, the *generate/0* method of the passive interrupt object is used to force an interrupt occur. When a lift button or a floor button is pushed, the *write/2* method of the associated data buffer is called. The data buffer stores the button push's information in a first-in-first-out queue and sends a *generate* message to the corresponding passive interrupt object. When the interrupt's ISR is executed, it calls the *read/1* method of its data buffer to obtain the data for the interrupt being processed, and then passes the data to the appropriate active object.

The interactions between the interrupts and the software and between the central controller and the lift managers are shown as uncommitted so experimentation can be done. For instance, if all are synchronous, deadlock may occur with *liftCont* waiting at the *floorRequest/1* method of a lift manager, *liftInt* blocked at the *liftButton/1*

⁶¹*Ivarwrite* interrupts could have been used also.

⁶²The other interrupt types are generated by MLog's virtual processor.

method of the same lift manager, the lift manager waiting at the `report/2` method of the central controller, and `floorInt` blocked at the `floorButton/1` method of `liftCont`. It will be shown that *ProtoTool* and *TimeLineTool* can help illustrate such problems.

8.3 Design Examples and Evaluation

8.3.1 An Error-Free Design

The first design uses a configuration where all the uncommitted interactions are asynchronous. Figure 8.3 shows the lift control system's user interface and *TimeLineTool* for a portion of an execution. To arrive at this situation, lift three received a *down* request at floor five and took passengers to the first floor. Lift two processed an *up* request at the third floor and delivered people to the fourth floor. Lift one left passengers at the third floor after receiving an *up* request at the first floor.

The *TimeLineTool* display shows the sequence of actions which occurred between when lift one's third floor lift button was pushed and when lift one stopped at the third floor:

- 1 — the lift button interrupt occurred, interrupting `lift1`;
- 2 — the asynchronous message pass from `liftInt` to `lift1` took place;
- 3 — `lift1` resumed execution and processed the lift button request for the third floor;
- 4 — `lift1`, after determining it had to move up, interacted with `liftCont` to report its new status of *moving*;
- 5 — `lift1` did some internal processing and then waited for notification that it had reached the second floor⁶³;
- 6 — `lift1` reported its new status to `liftCont`;
- 7 — `lift1` did some internal processing and then waited for notification that it had reached the third floor;
- 8 — after reaching the third floor, `lift1` informed `liftCont` of its new position;
- 9 — with the passengers let off, `lift1` reported its new status of *idle* to `liftCont`;
- 10 — `lift1` resumed waiting for floor requests or lift requests.

⁶³A timed accept is used to simulate movement between floors.

<input type="button" value="Speed"/> <input type="button" value="Timeslice"/>		<input type="button" value="Absolute"/> <input type="button" value="Forward"/> <input type="button" value="Backward"/>		Simulation Mode: <input type="text" value="interactive"/>
Speed: 100 Timeslice: 0		Time: [0,1,14]		
<hr/>				
System: stopped		<input type="button" value="Gonow"/> <input type="button" value="Gonext"/>		<input type="button" value="no time line display"/> <input type="button" value="record"/>
Application: released		<input type="button" value="Quit"/> <input type="button" value="Hold"/>		File:
<hr/>				
Interrupts: ☒ disabled Interrupts have priority: ☒ false				
Interrupt Selection: -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 <input type="button" value="Clear"/> <input type="button" value="Display"/>				
<input type="button" value="Enabled"/> <input type="button" value="Suspended"/> <input type="button" value="ClearAll"/> <input type="button" value="Enable"/>		<input type="button" value="Suspend"/> <input type="button" value="Queueing"/>		<input type="button" value="Generate"/>
<input type="button" value="Disabled"/> <input type="button" value="Resumed"/> <input type="button" value="Pending"/> <input type="button" value="Disable"/>		<input type="button" value="Resume"/> <input type="button" value="Ivar"/>		<input type="button" value="Priority"/>
<hr/>				
<pre> At [0,0,15]: No application objects able to execute At [0,0,20]: No application objects able to execute At [0,0,53]: No application objects able to execute At [0,0,58]: No application objects able to execute At [0,1,14]: No application objects able to execute </pre>				
<hr/>				
<pre> Processed: [[[0,0,15],[0,0,15],[0,0,15],[0,0,21],[0,0,21],[0,0,22]],[[0,0,53],[0,0,53],[0,0,53]]] Pending: [] Missed: [] Waiting objects: [] Ivarread: object=_32898,ivar=liftButton,Priority=10,Queued: size=5 Isr class=liftIsr,Not running,Enabled,Resumed Processed: [[[0,0,20],[0,0,20],[0,0,21],[0,0,32],[0,0,32],[0,0,32]],[[0,0,58],[0,0,58],[0,0,58]]] Pending: [] </pre>				

Figure 8.4 ProtoTool In The Error-Free Design

TimeLineTool's magnification slider was used to ensure that all the desired information was visible on one screen. This example illustrates how the temporal behaviour of all entities involved can be displayed in detail.

ProtoTool's display of this situation is shown in figure 8.4. It shows the lift button interrupt of priority ten and the floor button interrupt of priority fifteen in the interrupt-control window. Because each lift is idle, the system is waiting for either a floor request or a lift request. Therefore, there are no application objects to execute as is depicted in the next-object window. The top half of the information-display window contains information about floor button interrupts. The first processed interrupt is the one which was assigned to lift three, while the second was the *up* request given to lift two. The third floor interrupt was processed by lift one and initiated the sequence of actions described above. The data about lift interrupts, shown in the bottom half of the information-display window, contains the three lift button pushes. The last one corresponds to action 3 above.

8.3.2 A Design With Interrupt Priority Allocation Errors

The second design is in error because of its allocation of floor and lift interrupt priorities. In this design, each of the uncommitted interactions is synchronous. All of the active objects in the system use the default priority of fifty, while the floor and lift interrupts are priority fifteen and ten respectively. Thus, when either of the interrupts attempts to interact with the corresponding active object, the interrupt will become suspended because its rendezvous partner has a lower priority⁶⁴.

The *TimeLineTool* display of this situation is shown in figure 8.5. The following sequence of actions took place:

- 1 — an *up* request from the first floor generated a floor interrupt, so `floorInt` began execution;
- 2 — `floorInt` initiated a synchronous interaction with `liftCont` to pass it the floor request: `liftCont` was placed on the ready-to-run queue;
- 3 — lift one's third floor lift button was pushed, so `liftInt` began execution;
- 4 — `liftInt` was about to initiate a synchronous interaction with `lift1`.

⁶⁴Recall that when an interrupt becomes suspended, only higher priority active objects can run until the interrupt resumes execution.

Figure 8.5 TimeLineTool In The Design With Interrupt Priority Allocation Errors

The fact that each active object appears to have come to an abrupt end is a good indication that some form of error has occurred. An examination of the display reveals that an interrupt has attempted a synchronous interaction with an active object, an event which demands further investigation. This investigation would be simpler if the priority of each active object was included with its name, allowing priority inversions such as this to be detected easily.

Figure 8.6 illustrates how *ProtoTool* viewed this situation. The next-object window shows that at [0,0,15], when the floor interrupt occurred, *liftCont* was scheduled and made ready to run, but *floorInt* had control of the virtual processor. Then, when the lift button interrupt occurred, *liftCont* was still waiting to execute, but now *liftInt* was preventing it from doing so. Thus, the developer is made aware of why the system is not running. The state of the floor interrupt, displayed in the top half of the information-display window, shows that the interrupt which occurred at [0,0,15] is still being processed because it shows only its time of occurrence and the time when processing began, while the two subsequent occurrences are pending. The lift interrupt's situation is similar, as shown in the bottom half of the information-display window.

8.3.3 A Design With Software Deadlock

In this design, the interrupts interact asynchronously with their associated active objects, while the central controller uses a synchronous interaction to pass floor requests to the lift managers. Should a lift manager be reporting its status to the central controller at a time when the central controller elects to give the lift manager a new floor request, deadlock will occur. Figure 8.7 shows the *TimeLineTool* display of such a situation.

The labeled actions in figure 8.7 are described as follows:

- 1 — processing of the interrupt for the last valid lift button push of lift one was completed;
- 2 — the *up* request from the second floor, which was pending during the processing of lift one's lift button interrupts, could be processed: *floorInt* began execution;
- 3 — *floorInt* passed the floor request to *liftCont*;

Figure 8.7 TimeLineTool In The Design With Software Deadlock

114

Figure 8.8 ProtoTool In The Design With Software Deadlock

- 4 — `lift1`, which had been interrupted by its own lift button requests, continued processing and initiated a synchronous interaction with `liftCont` to report its new status of *moving*;
- 5 — `liftCont` began execution and accepted the new floor request from `floorInt`;
- 6 — after deciding that lift one was able to handle the new floor request, `liftCont` initiated a rendezvous with `lift1`, creating the deadlock;
- 7 — `floorInt` began execution because another floor request interrupt was generated;
- 8 and 9 — additional floor requests were passed to `liftCont`.

The *TimeLineTool* display shows how `liftCont` accepted the new floor request before accepting the status update from `lift1`. Further, by showing exactly where an active object is waiting, it is easy to establish that `liftCont` and `lift1` are waiting for each other.

At the time of deadlock, the *ProtoTool* display shown in figure 8.8 was generated. The next-object window hints at deadlock by saying that no application objects are able to execute. The information-display window shows the status of the floor interrupt. Because no interrupts are pending or were missed⁶⁵, the developer would be able to focus the debugging effort on the software.

8.3.4 A Design With Interrupt Deadlock

In the design with interrupt priority allocation errors, the interrupts became suspended because their priorities were higher than those of the active objects with which they interacted. If the interrupts' priorities are lower than their rendezvous partners' priorities, deadlock can occur involving the interrupts. The interrupt deadlock design uses interrupt priorities of sixty and sixty-five instead of ten and fifteen, and all interactions are synchronous.

A situation showing deadlock using this design is illustrated in the *TimeLineTool* display of figure 8.9, the significant actions of which are:

- 1 — `lift2` finished processing and became *idle* at the first floor;
- 2 — an *up* request from the first floor caused `floorInt` to initiate a synchronous interaction with `liftCont`;

⁶⁵The same was true of the lift interrupt, but both interrupts could not be displayed at once.

Figure 8.9 TimeLineTool In The Design With Interrupt Deadlock

Speed 100	Timeslice 0	<input type="button" value="Speed"/> <input type="button" value="Timeslice"/>	<input type="button" value="Absolute"/> <input type="button" value="Forward"/> <input type="button" value="Backward"/>	Simulation Mode: interactive
Time: [0,1,14]				
System: stopped <input type="button" value="Gonow"/> <input type="button" value="Gonext"/> <input type="button" value="Hold"/>				
Application: released		no time line display <input type="button" value="record"/>		
File:				
Interrupts: disabled Interrupts have priority: false				
Interrupt Selection: -1 -1 -1 -1 -1 -1 -1 -1 65 <input type="button" value="Clear"/> <input type="button" value="Display"/>				
<input type="button" value="Enabled"/> <input type="button" value="Suspended"/> <input type="button" value="ClearAll"/> <input type="button" value="Enable"/> <input type="button" value="Suspend"/> <input type="button" value="Queueing"/> <input type="button" value="Generate"/>				
<input type="button" value="Disabled"/> <input type="button" value="Resumed"/> <input type="button" value="Pending"/> <input type="button" value="Disable"/> <input type="button" value="Resume"/> <input type="button" value="Ivar"/> <input type="button" value="Priority"/>				
At [0,1,10]: No application objects able to execute At [0,1,14]: No application objects able to execute Processor held by: ISR for interrupt priority 65 At [0,1,14]: No application objects able to execute Processor held by: ISR for interrupt priority 60				
Pending: [[[0,1,14],Processing in progress],[[0,1,14],Processing in progress],[[0,1,14],Processing in progress],[[0,1,14],Processing in progress],[[0,1,14],Processing in progress]] Missed: [[[0,1,14],[0,1,14],Others pending],[[0,1,14],[0,1,14],Others pending]] Waiting objects: []				
Ivarread: object=_47627,ivar=liftButton,Priority=60,Queued: size=5 Isr class=liftIsr,Running,Enabled,Resumed Processed: [[[0,0,20],[0,0,20],[0,0,21],[[0,0,54],[0,0,54],[0,0,55],[[0,1,14],[0,1,14]]]] Pending: [[[0,1,14],Processing in progress]]				

Figure 8.10 ProtoTool In The Design With Interrupt Deadlock

- 3 — `liftCont` determined that lift one should handle the request, so a rendezvous with `lift1` was started;
- 4 — `lift1` received the floor request, completed the rendezvous with `liftCont`, and then initiated a synchronous interaction with `liftCont` to pass its new status of *servicing*;
- 5 — `liftCont` completed the rendezvous with `floorInt`;
- 6 — `floorInt` was made ready to run to complete processing of the interrupt, but it could not run because its priority was lower than that of `liftCont`;
- 7 — `liftCont` completed the rendezvous with `lift1`, so `lift1` was made ready to run;
- 8 — an *up* request from the second floor was generated while `floorInt` was waiting to resume processing, so `floorInt` continued running and passed the new request to `liftCont`, which was made ready to run;
- 9 and 10 — `lift1` resumed execution and initiated a synchronous interaction with `liftCont` to report its new status of *waiting*;
- 11 — `liftCont` resumed running and began processing the rendezvous initiated by `floorInt`: believing that lift one was able to handle the request, a synchronous interaction with `lift1` was started, generating deadlock;
- 12 — after the floor interrupt's pending interrupt queue became full, subsequent floor requests resulted in missed interrupts;
- 13 — a lift button push for lift one resulted in `liftInt` beginning a synchronous interaction with `lift1`, but because `lift1` was waiting for `liftCont`, `liftInt` was forced to wait.

The *TimeLineTool* display around action 11 reveals a deficiency with the current version. It is difficult to determine if the rendezvous initiated by `liftCont` is with `floorInt` or `lift1`. A method of highlighting the rendezvous partner would make it easier to establish the order in which messages are processed. Also, the floor and lift interrupts' priorities were changed after they were defined, but this was not reflected in their names in the *TimeLineTool* display. It should be possible to alter a task's name.

Figure 8.10 shows *ProtoTool*'s display of the system when the deadlock occurred. The next-object window indicates that `liftInt` has become suspended and that no

active objects are able to run. This tells the developer that an error exists which is preventing the interrupt from interacting correctly with the software. The top half of the information-display window shows that five floor interrupts are pending while two have been missed. Note that all the times are the same, a further illustration that the application has deadlocked. The lift interrupt status, shown in the bottom half of the information-display window, indicates that the last occurrence is still being processed because the interrupt is *running*. In addition, one occurrence is pending.

Chapter 9

Suggested Improvements to *ProtoTool* and to *TimeLineTool*

In their current versions, *ProtoTool* and *TimeLineTool* give developers assistance in evaluating concurrent system designs. This chapter lists possible improvements to both tools and outlines briefly other tools which, if combined with *ProtoTool* and *TimeLineTool*, would create a more sophisticated prototyping package.

9.1 *ProtoTool* Improvements

9.1.1 Interrupts

Currently, methods `waitForInterrupt/6` and `waitForInteraction/7` allow an active object to wait for only one interrupt. An interrupt task should be able to specify a list of interrupts for which it can receive notification.

When an interrupt task is notified, it should be possible to pass any data associated with the interrupt directly to the task. This would require modifications to methods `waitForInterrupt/6`, `waitForInteraction/7`, `waitForInterrupt/1`, and `waitForInteraction/3`. For instance, the priority specification could be given as

```
[PriorityList, InterruptData]
```

where `PriorityList` is a list of interrupt priorities for which to wait, and `InterruptData` is where any data would be deposited.

Currently, applications must buffer interrupt data explicitly as was done in the lift control problem. Data associated with any pending interrupts should be maintained by *ProtoTool* to ease developers' responsibilities.

In the current version of *ProtoTool*, an interrupt's ISR is fixed once it is defined. In order to allow experimentation, ISR's should be changeable.

When an interrupt is missed, details about the occurrence are added to the *missed* list of the associated passive interrupt object, but no further action is taken. A facility should exist for binding a method to an interrupt which is executed when an occurrence of the interrupt is missed. This would allow developers to specify failure recovery actions. For example, in a controller for a nuclear reactor, it may be necessary to alert the operators by raising an alarm if a critical interrupt is missed.

A developer should be able to specify that an interrupt will occur when a specified message either succeeds or fails. For instance, in this code:

```
a :- b,c.
b :- d.
c :- e,f.
d.
e.
```

an interrupt could be defined which is to occur when the message to method *b/0* from method *a/0* succeeds, or one which should be generated when the message to method *f/0* fails when called from method *c/0*. This way of forcing interrupts anywhere in the application would be better than using *ivarread* or *ivarwrite* interrupts which was the method used in the lift control example. Other interrupt types should be added, such as ones which have normal or Poisson occurrence distributions. In addition, it should be possible to have more than one pending occurrence of interrupts whose queuing mode is timed.

To facilitate building prototypes which use button pushes to simulate interrupt occurrences, a "button interrupt" should be added which would be generated automatically when the corresponding button is pushed. In the lift control example, this would eliminate the need for the *ivarread* interrupts, and would result in a more consistent conceptual model.

Presently, when an interrupt's priority is changed, any interrupt tasks waiting for the interrupt remain on the interrupt's *waiters* queue. This was done because, from a physical standpoint, an interrupt does not change when its priority is modified. However,

there may be cases when an interrupt task does not want to remain waiting after the priority has been updated. Thus, it should be possible to inform interrupt tasks when an interrupt's priority is modified.

Once and repeat interrupts should be able to specify if the times in their time lists are absolute or relative to the time at definition. Only the latter is supported now. For repeat interrupts, only the first pattern can be absolute. The remainder must be relative to the time when the previous pattern ended.

When `waitForInterrupt/6` and `waitForInteraction/7` are called, pending occurrences of the interrupt in question are not able to notify the calling active object. Only interrupts which occur *after* the interrupt task is waiting are eligible. It should be possible for an interrupt task to specify if pending occurrences are to be considered.

Currently, if an interrupt handler becomes suspended, only active objects whose priorities are higher than that of the interrupt handler in question are allowed to execute. Ideally, developers should be able to specify which priorities are eligible for execution when an interrupt handler is suspended. This would give developers more control over prototype execution.

When an interrupt's details, such as queuing mode, are altered, any pending occurrences are discarded. This should be done only in certain circumstances. For instance, if an interrupt's queuing mode is queued, changing the size of the queue should have no effect on the pending occurrences.

9.1.2 Gathered Information

In addition to the information gathered currently, the following data could be obtained and presented to developers:

- the maximum and average number of waiting active objects at a specified set of methods;
- the maximum and average waiting time at a specified set of methods;
- the number of context switches;
- the number of times a specified method is called;
- the number of priority inversions⁶⁶;

⁶⁶A priority inversion occurs when a task waits for one of lower priority.

- the processor utilization.

These would require counters to be maintained within the MLog virtual processor.

9.1.3 Miscellaneous

A conditional send should be implemented. A timed send with a relative delay of zero or a timed send with an absolute time which has already passed is the best way to do this.

All queues in MLog and *ProtoTool*, such as message queues and the *waiters* queue for interrupts, operate independently of the priorities of the queued objects. To permit experimentation, developers should be able to select the operation mode for queues. The current method, first-in-first-out, and priority-based are two possible modes.

Developers may want to alter an object's message queue. For instance, it may be necessary to determine the effects of ignoring a particular message pattern when searching for valid messages to accept. Facilities should be provided to ask and answer questions such as "What would happen if object A ignored all messages from object B?"

Because one second is the smallest timing resolution, any events which occur between time $[x, y, z]$ and time $[x, y, (z+1)]$ are considered to have taken place at time $[x, y, z]$. Thus, if an object delays at time $[x, y, z]$ for s seconds, it may be scheduled before s seconds elapse. Figure 9.1 shows an example for a case where five message passes constitutes one second. An improved algorithm is needed for determining the time when an object should be scheduled after a timed operation.

Figure 9.1 An Example of *ProtoTool's* Timeout Scheduling

9.2 *TimeLineTool* Improvements

A provision for changing the name of a task is necessary. Also, each task's priority should be displayed beside its name so that situations such as priority inversions can be detected easily.

The last evaluation example in chapter eight illustrated the need for an "interrupt pending" event which would show that an interrupt occurred but could not be processed immediately.

As developers may be concerned with only a small portion of the displayed execution, it should be possible to move to a specific time. Currently, the display must be traversed "manually" using successively smaller steps to arrive at the desired time. This can be time-consuming and annoying.

Many times, developers are concerned with the events of a particular task. Thus, it should be possible to highlight the execution thread of a task. On a colour screen, events pertaining to the task in question could be displayed in colour while all other tasks' events would be in black and white. On a monochrome monitor, the task's events could be shown in thick lines with all other events in thin lines. In conjunction with this feature, it should be possible to select a task and step through the display based on that task's events rather than based on time.

Presently, it is difficult to determine the sender of a synchronous message which is chosen by a receiver because the execution threads for all the waiting senders are merged into a single waiting line. This was done to allow an unlimited number of senders to be displayed simultaneously. Ideally, the chosen sender should be highlighted. One potential solution is shown in figure 9.2. Each sender is given its own waiting line, so it is easy to see the source of a message. The primary disadvantage of this solution is that as the number of waiting senders increases, the amount of room required to show their waiting lines increases. This would necessitate dynamically reconfiguring the display to generate space for more waiting lines.

In the second approach, no events are discarded, but a special event is recorded which indicates that the time was set backward. By doing this, developers can see everything that happened around the time when the time was modified.

9.3 Other Tools

Prolog's declarative semantics can lead to faster coding compared to an imperative language like Ada. MLog and *ProtoTool* take advantage of this. However, the code must still be written. Two tools which can assist this process are a *graphical design editor* and a *syntax-directed code editor*. A graphical design editor allows developers to enter the structural characteristics of a design using diagrams such as the one in figure 8.1, and then automatically generates a corresponding code skeleton. Algorithms must be filled in manually. An example of such a tool is CAEDE [Buhr85]. This tool not only generates code skeletons, it also performs some design analyses which is beneficial for prototyping. The best candidates for graphical editors are languages which are syntactically heavy. A syntax-directed editor can expedite code entry by performing operations such as automatically checking for syntax errors and formatting as code is entered.

With the development of high-resolution monitors, user interfaces have become quite sophisticated, and MLog's windowing capabilities have kept up with this trend. However, these interfaces can be difficult to construct, requiring many iterations before everything is exactly where it should be. A *user interface building tool* would allow developers to graphically specify a user interface by selecting components and moving them around the screen with a pointing device such as a mouse. When desired, the tool would automatically generate the code required to build the existing interface.

Often, only a piece of an existing system needs prototyping. Thus, it should be possible to use prototypes in concert with production systems. A *callin-callout tool* would provide the necessary interfaces to allow prototypes to call out to production systems and to allow production systems to call in to prototypes.

Because MLog employs an object-oriented programming paradigm, large component libraries can be constructed. A *component-composition tool* would take a behavioural specification and would provide assistance in searching the library for components to

meet the specification. This would require associating behavioural information with each component, and may require a language and an interpreter to be effective. The result would be a behaviour-based class browser which would eliminate classes whose behaviour is not appropriate.

Chapter 10

Conclusions

Developing software is a difficult process. Rarely is there a single correct solution, so developers rely on intuition and experience when generating potential designs. This is particularly true of concurrent systems where intertask communication, timing parameters, and environment interaction using interrupts help to make building such software very challenging. The development paradigm most frequently used, the waterfall paradigm, is poorly suited for experimenting with different designs because of its sequential nature. Conversely, the prototyping life cycle model, with its built-in iteration, is ideal for constructing concurrent systems. By observing the behaviour of many potential designs before making a commitment, the best one can be selected for production-quality coding, thereby reducing the number of problems encountered during maintenance.

In order to make experimenting with designs an inexpensive and effective endeavour, developers must be provided tools to assist this process. In this thesis, we have addressed this issue by developing two tools, *ProtoTool* and *TimeLineTool*, which together form a prototyping environment for concurrent systems. The capabilities included were influenced by the following goals:

- a prototyping environment must allow pertinent information to be gathered and must present it in an effective manner;
- developers must be able to control the execution of prototypes to perform probing and experimentation.

These two tools and the theoretical models on which they are based are summarized below.

10.1 A Research Summary

10.1.1 Time

Many concurrent systems use timed constructs. The basic timed interactions were discussed in chapter four: (1) delays, (2) timed accepts, and (3) timed sends. It was shown that, in a uniprocessor environment, a conditional send is an expression of a potential receiver's readiness to accept a message and should not involve time. Many timing issues not present in a uniprocessor configuration exist in a distributed one. In particular, communication delays must be considered and can alter the semantics of timed interactions. A further timing difficulty in distributed systems involves maintaining a common sense of time among all nodes. Four possible solutions were explored: (1) a network time server, (2) synchronized clocks, (3) independent clocks, and (4) task migration. Using synchronized clocks is the best solution when communication between the nodes and a central time server is possible. If not, independent clocks must be used.

10.1.2 Interrupts

The interrupt model employed by *ProtoTool* was described in chapter five. In *ProtoTool*, interrupts are modeled as entities which autonomously gain control of the processor. Three interrupt configurations were illustrated: (1) single-line, (2) multiple-line, and (3) vectored. *ProtoTool* implements a vectored interrupt scheme. When an interrupt occurs, its servicing may have to be postponed. The three queuing modes supported by *ProtoTool*, immediate, queued, and timed, were outlined and examples were given.

10.1.3 *ProtoTool*

ProtoTool provides facilities for controlling the execution of prototypes and for presenting developers with relevant concurrent system information. It is integrated with the MLog programming language, but its concepts are portable. If run in MLog's window mode, the user interface is window-based to allow easy input and quickly-assimilated, continuously-displayed output. If only a line-by-line console is available, output must be requested explicitly while input is textual-based rather than window-based.

By incorporating the free-running and interactive simulation modes, developers can either passively observe a prototype's execution or actively take control of it. In addition,

by being able to stop execution at any time, developers have the ability to probe designs and analyze the effects of their intervention. Characteristics of MLog's virtual processor, such as its speed and task scheduling policy, can be modified to help determine the best platform on which to run an application. Because many concurrent systems interact with their environment using interrupts, *ProtoTool* provides a conceptually-consistent model of interrupts and simulates them in software. Developers can control how interrupts affect an application by, among others, enabling and disabling them, changing their relative priorities, and altering how they are serviced.

Information about a design can be extracted and can be displayed either in the window-based user interface or in an MLog console if that is where a request originates. For example, details about processed, pending, and missed interrupts can be obtained. Some information, such as the system time, is always available in the window-based interface, and can be obtained easily if only line-by-line access is possible.

Some of *ProtoTool*'s capabilities were illustrated during the evaluation of the lift control example.

10.1.4 *TimeLineTool*

Often, the most difficult part of a concurrent system design is determining when tasks should interact with each other and with interrupts. Thus, this should be the information-presentation focus. *TimeLineTool* displays the temporal ordering of task and interrupt interactions, allowing developers to view graphically a prototype's behaviour. Its real-time mode gives developers a chance to examine an application's characteristics as it runs, permitting feedback to be incorporated as design changes "on the fly". The replay mode allows detailed after-the-fact analyses, and can be used later to illustrate an initial or reference behaviour. The *pan* and *zoom* features enable developers to analyze a prototype's behaviour on a large scale or with a concern for detail. The concurrent system events which can be displayed are generic, and the input data format is simple. Thus, *TimeLineTool* can be used with any run-time system which is equipped with an execution trace facility.

It was shown that *TimeLineTool* is useful for illustrating potential problems of concurrent systems, such as deadlock and priority inversions. In the evaluation example,

it was very easy to see where tasks were waiting and why the application was unable to continue.

The small examples given at the end of chapters four and five were used to verify that the time and interrupt models employed by *ProtoTool* are consistent and can be applied to actual design problems. The lift control problem evaluated in chapter eight had more ambitious goals:

- to demonstrate the operation of *ProtoTool* and *TimeLineTool*;
- to show how these tools work in concert to identify problems with concurrent system designs;
- to evaluate both tools, uncover any deficiencies with them, and suggest desirable improvements.

It is difficult to get an appreciation for how *ProtoTool* and *TimeLineTool* operate by looking at static illustrations because both provide feedback on a dynamic basis. We feel that they deliver their advertised capabilities and are transparent to applications except that, because they require processor resources, they make applications run slower. However, this was expected and the knowledge which is gained by experimenting with design decisions offsets any performance drawbacks.

TimeLineTool correctly identified the deadlock situation in the lift control problem by showing that the components in question were waiting for each other. *ProtoTool* assisted the analysis by stating that the application was unable to continue. In the design which contained interrupt deadlock, *ProtoTool's* display of the interrupts' details indicated clearly that a problem existed in the processing of both the floor interrupt and the lift interrupt. Thus, *ProtoTool* helped to direct the focus of the analysis effort.

The evaluation of the lift control example revealed no significant deficiencies with either tool. Some of the small problems uncovered include the following:

- in the *TimeLineTool* display, a task's priority should be shown as well as its name to give developers as much information as possible about each component;
- the information display window of *ProtoTool* should be enlarged to allow more information to be illustrated;

- *TimeLineTool's* should indicate the sender of a message which is being used in an interaction.

The current versions of both *ProtoTool* and *TimeLineTool* are prototypes themselves because many improvements and additions can be made. The feedback from evaluations performed up to now has lead to the suggested improvements discussed in chapter nine. Time constraints prevented implementing these suggestions, but all are feasible and should be included in the next versions.

The examples used in this thesis to evaluate *ProtoTool* and *TimeLineTool* are "real world" problems, but they are not large in size. *ProtoTool* is independent of the size of the system it is prototyping except with respect to interrupts. Because only ten interrupts can be defined, systems which require more than ten would have to be modified some way. *TimeLineTool's* restriction with large systems is its inability to display every task's time line simultaneously.

Because both *ProtoTool* and *TimeLineTool* are designed with a uniprocessor system in mind, it would be difficult to prototype hardware systems⁶⁸. Any time delays associated with waiting in a ready-to-run queue would result in a flawed temporal representation. However, when a distributed computing model is added, hardware systems could be handled in a much more consistent manner.

It is very difficult to prove that hard real-time deadlines can be met. This is especially true of *ProtoTool* because the timing model it employs is designed to give approximations of the eventual target timing characteristics. A mapping such as the one mentioned in section 4.2.1 would make the timing model more realistic, but still not completely accurate. Only the target environment can give true timing indications.

ProtoTool and *TimeLineTool* can assist developers in achieving high-quality designs. However, this does not guarantee that the behaviour embedded in the resulting design prototype will appear in the target implementation. If the target code is generated manually, it is the developer's responsibility to extract the necessary information from the design prototyped and include it in the implementation. Automated translation techniques

⁶⁸A computer board with many chips is logically equivalent to distributed system where each chip corresponds to a node.

and tools can lessen the burden on developers, but normally require some degree of manual intervention.

10.2 Further Research

This section briefly describes three areas of possible future research: (1) program visualization, (2) programming language paradigms, and (3) event trace reasoning.

TimeLineTool's graphical event display provides developers with useful feedback about concurrent system designs. However, time lines are not the only way to give developers feedback. Different program visualization methods should be researched. In addition, desirable view manipulation techniques, some of which were outlined in the previous chapter, should be identified.

MLog combines three programming language paradigms to create a good language for prototyping. Research is required to determine how other paradigms can be used to assist in creating prototypes. For instance, there are cases when imperative programming is desirable.

The concurrent system events which are the input to *TimeLineTool* give an event trace which could be used to reason about the design. A design-reasoning tool which uses event traces as its input could provide useful information about a design.

References

- [ALRM83] *Reference Manual for the Ada Programming Language*, MIL-STD-1815a, U.S. Department of Defence, 1983.
- [DRAF89] "Draft Report on Requirements for a Common Prototyping System", *ACM SIGPLAN Notices*, vol. 24, no. 3, March 1989, pp. 93–165.
- [RPWS82] ACM SIGSOFT, *Rapid Prototyping Workshop*, Columbia, Maryland, 1982, (Appears as ACM SIGSOFT Software Engineering Notes Vol. 7, No. 5, December 1982).
- [Agre86] Agresti, W. W., "What Are The New Paradigms?", In *New Paradigms for Software Development*, W. W. Agresti, Ed., IEEE Computer Society Press, 1986, pp. 6–10.
- [Ambr88] Ambras, J., and O'Day, V., "Microscope: A Program Analysis System", *IEEE Software*, vol. 5, no. 3, May 1988, pp. 50–58.
- [Balz86] Balzer, R., Cheatham, Jr., T. E., and Green, C., "Software Technology in the 1990's: Using a New Paradigm", In *New Paradigms for Software Development*, W. W. Agresti, Ed., IEEE Computer Society Press, 1986, pp. 16–22.
- [Bank84] Banks, J., and Carson, II, J. S., *Discrete-Event System Simulation*, International Series in Industrial and Systems Engineering, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1984.
- [Boar85] Boar, B. H., *Application Prototyping, A Project Management Perspective*, AMA Management Briefing, AMA Membership Publications Division, American Management Association, New York, 1985.
- [Boeh76] Boehm, B. W., "Software Engineering", *IEEE Transactions on Computers*, vol. C-25, no. 12, December 1976, pp. 1226–1241.
- [Booc86] Booch, G., "Object-Oriented Development", *IEEE Transactions on Software Engineering*, vol. SE-12, no. 2, February 1986, pp. 211–221.

- [Brow88] Brown, D. W., Carson, C. D., Montgomery, W. A., and Zislis, P. M., "Software Specification and Prototyping Technologies", *AT&T Technical Journal*, vol. 67, no. 4, July & August 1988, pp. 33–45.
- [Brun86] Bruno, G., and Marchetto, G., "Process-Translatable Petri Nets for the Rapid Prototyping of Process Control Systems", *IEEE Transactions on Software Engineering*, vol. SE-12, no. 2, February 1986, pp. 346–357.
- [Bruy84] Bruynooghe, M., and Venken, R., "Prolog as a Language for Prototyping of Information Systems", In *Approaches to Prototyping*, R. Budde, K. Kuhlenkamp, L. Mathiassen, and H. Zullighoven, Eds., Springer-Verlag, 1984, pp. 447–458, (Proceedings of the Working Conference on Prototyping, Namur, October 1983).
- [Buhr90] Buhr, R. J. A., *Practical Visual Techniques in System Design With Applications to Ada*, Prentice Hall Software Series, Prentice Hall, 1990.
- [Buhr85] Buhr, R. J. A., Woodside, C. M., Karam, G. M., van der Loo, K., and Lewis, D. G., "Experiments with PROLOG Design Descriptions and Tools in CAEDE: An Iconic Design Environment for Multitasking Embedded Systems", In *8th International Conference on Software Engineering*, 1985, pp. 62–67.
- [Dahl87] Dahler, J., Gerber, P., Gisiger, H.-P., and Kundig, A., "A Graphical Tool for the Design and Prototyping of Distributed Systems", *ACM SIGSOFT Software Engineering Notes*, vol. 12, no. 3, July 1987, pp. 25–34.
- [Davi82] Davis, R. E., "Runnable Specifications as a Design Tool", In *Logic Programming*, K. L. Clark and S. A. Tarnlund, Eds., Academic Press, London, England, 1982, pp. 141–149.
- [Dwig83] Dwiggins, D., "Prolog as a System Design Tool", In *Proceedings of the 16th Annual Hawaii International Conference on System Sciences*, 1983, pp. 14–23.
- [Fish87] Fisher, G. E., *Application Software Prototyping and Fourth Generation Languages*, Special Publication 500-148, United States National Bureau of Standards, May 1987.
- [Fish78] Fishman, G. S., *Principles of Discrete Event Simulation*, Wiley Series on Systems Engineering and Analysis, John Wiley and Sons, Inc., 1978.
- [DeFr88] Francesco, N. D., and Vaglini, G., "Description of a Tool for Specifying and Prototyping Concurrent Programs", *IEEE Transactions on Software Engineering*, vol. 14, no. 11, November 1988, pp. 1554–1564.

- [Gidd86] Giddings, R. V., "Accommodating Uncertainty in Software Design", In *New Paradigms for Software Development*, W. W. Agresti, Ed., IEEE Computer Society Press, 1986, pp. 29–35.
- [Glad82] Gladden, G. R., "Stop the Life Cycle, I Want to Get Off", *ACM SIGSOFT Software Engineering Notes*, vol. 7, no. 2, April 1982, pp. 35–39.
- [Gold84] Goldberg, A., *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, 1985.
- [Hare88] Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., and Shtul-Trauring, A., "STATEMATE: A Working Environment for the Development of Complex Reactive Systems", In *10th International Conference on Software Engineering*, 1988, pp. 396–406.
- [Hekm88] Hekmatpour, S., and Ince, D., *Software Prototyping, Formal Methods and VDM*, International Computer Science Series, Addison-Wesley, 1988.
- [Hend86] Henderson, P., "Functional Programming, Formal Specification, and Rapid Prototyping", *IEEE Transactions on Software Engineering*, vol. SE-12, no. 2, February 1986, pp. 241–250.
- [Hend85] Henderson, P., and Weiser, M., "Continuous Execution: The VISIPROG Environment", In *8th International Conference on Software Engineering*, 1985, pp. 68–74.
- [Hoar78] Hoare, C. A. R., "Communicating Sequential Processes", *Communications of the ACM*, vol. 21, no. 8, August 1978, pp. 666–667.
- [Hock85] Hockey, S., *SNOBOL Programming for the Humanities*, Oxford University Press, New York, 1985.
- [Jado89] Jadoul, L., Duponcheel, L., and Van Puymbroeck, W., "An Algebraic Data Type Specification Language and its Rapid Prototyping Environment", In *11th International Conference on Software Engineering*, 1989, pp. 74–84.
- [Jard88] Jard, C., Monin, J.-F., and Groz, R., "Development of Veda, a Prototyping Tool for Distributed Algorithms", *IEEE Transactions on Software Engineering*, vol. 14, no. 3, March 1988, pp. 339–352.
- [Kara89a] Karam, G. M., *MLog: A Language for Prototyping Concurrent Systems*, Tech. Rep. CU-SCE-88-3 (Revised), Department of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, Canada, September 1989.
- [Kara89b] Karam, G. M., Buhr, R. J. A., Woodside, C. M., and Casselman, R.,

TimeBench: A Next Generation CASE Tool, Tech. Rep. CU-SCE-89-18, Department of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, Canada, 1989.

- [Kern84] Kernighan, B. W., and Pike, R., *The UNIX Programming Environment*, Prentice-Hall Software Series, Prentice-Hall, 1984.
- [Kley88] Kleyn, M. F., and Gingrich, P. C., "GraphTrace: Understanding Object Oriented Systems Using Concurrently Animated Views", In *OOPSLA 88*, 1988, pp. 191-205.
- [Komo86] Komorowski, H. J., and Maluszynski, J., *Logic Programming and Rapid Prototyping*, Tech. Rep. TR-01-86, Center for Research in Computing Technology, Harvard University, Cambridge, MA, 1986.
- [Kruc84] Kruchten, P., Schonberg, E., and Schwartz, J., "Software Prototyping Using the SETL Programming Language", *IEEE Software*, vol. 1, no. 4, October 1984, pp. 66-75.
- [Lamp78] Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, vol. 21, no. 7, July 1978, pp. 558-565.
- [Lieb84] Liebrandt, U., and Schnupp, P., "An Evaluation of Prolog as a Prototyping System", In *Approaches to Prototyping*, R. Budde, K. Kuhlenkamp, L. Mathiasen, and H. Zullighoven, Eds., Springer-Verlag, 1984, pp. 424-433, (Proceedings of the Working Conference on Prototyping, Namur, October 1983).
- [Lien89] Lienhard, C., *Three-Dimensional Debugging*, Tech. Rep. CU-SCE-89-23, Department of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, Canada, August 1989.
- [Love83] Love, T., "Experiences with Smalltalk-80 for Application Development", In *1983 SOFTFAIR: Software Development Tools, Techniques and Alternatives*, 1983, pp. 61-65.
- [Luqi89] Luqi, "Software Evolution Through Rapid Prototyping", *IEEE Computer*, vol. 22, no. 5, May 1989, pp. 13-25.
- [Luqi88c] Luqi, "Knowledge-based Support for Rapid Software Prototyping", *IEEE Expert*, vol. 3, no. 4, Winter edition 1988, pp. 9-18.
- [Luqi88b] Luqi, and Berzins, V., "Rapidly Prototyping Real-Time Systems", *IEEE Software*, vol. 5, no. 5, September 1988, pp. 25-36.
- [Luqi88a] Luqi, Berzins, V., and Yeh, R. T., "A Prototyping Language for Real-Time

- Software", *IEEE Transactions on Software Engineering*, vol. 14, no. 10, October 1988, pp. 1409–1423.
- [Mart88] Martin, C. F., "Second-Generation CASE Tools: A Challenge to Vendors", *IEEE Software*, vol. 5, no. 2, March 1988, pp. 46–49.
- [McCr86] McCracken, D. D., and Jackson, M. A., "A Minority Dissenting Position", In *New Paradigms for Software Development*, W. W. Agresti, Ed., IEEE Computer Society Press, 1986, pp. 23–25.
- [Niel89] Nielsen, J., and Richards, J. T., "The Experience of Learning and Using Smalltalk", *IEEE Software*, vol. 6, no. 3, May 1989, pp. 73–77.
- [OBri87] O'Brien, P. D., Halbert, D. C., and Kilian, M. F., "The TRELIS Programming Environment", In *OOPSLA 87*, 1987, pp. 91–102.
- [Ridd86] Riddle, W. E., and Williams, L. G., "Software Environments Workshop Report", *ACM SIGSOFT Software Engineering Notes*, vol. 11, no. 1, January 1986, pp. 73–102.
- [Royc70] Royce, W. W., "Managing the Development of Large Software Systems: Concepts and Techniques", In *IEEE WESCON*, 1970, pp. 1–9.
- [Rzev84] Rzevski, G., "Prototypes Versus Pilot Systems: Strategies for Evolutionary Information System Development", In *Approaches to Prototyping*, R. Budde, K. Kuhlenkamp, L. Mathiassen, and H. Zullighoven, Eds., Springer-Verlag, 1984, pp. 356–367, (Proceedings of the Working Conference on Prototyping, Namur, October 1983).
- [Swar86] Swartout, W., and Balzer, R., "On the Inevitable Intertwining of Specification and Implementation", In *New Paradigms for Software Development*, W. W. Agresti, Ed., IEEE Computer Society Press, 1986, pp. 26–28.
- [Tavo84] Tavolato, P., and Vincena, K., "A Prototyping Methodology and its Tool", In *Approaches to Prototyping*, R. Budde, K. Kuhlenkamp, L. Mathiassen, and H. Zullighoven, Eds., Springer-Verlag, 1984, pp. 434–446, (Proceedings of the Working Conference on Prototyping, Namur, October 1983).
- [Terw87] Terwilliger, R. B., *ENCOMPASS: An Environment for Incremental Software Development Using Executable, Logic-Based Specifications*, Tech. Rep. UIUCDCS-R-87-1356, Department of Computer Science, University of Illinois at Urbana-Champaign, July 1987.
- [Terw86] Terwilliger, R. B., and Campbell, R. H., *PLEASE: Executable Specifications for Incremental Software Development*, Tech. Rep. UIUCDCS-R-86-1295,

Department of Computer Science, University of Illinois at Urbana-Champaign,
October 1986.

- [Volz87] Volz, R. A., and Mudge, T. N., "Timing Issues in the Distributed Execution of Ada Programs", *IEEE Transactions on Computers*, vol. C-36, no. 4, April 1987, pp. 449-459.
- [Zave86] Zave, P., and Schell, W., "Salient Features of an Executable Specification Language and Its Environment", *IEEE Transactions on Software Engineering*, vol. SE-12, no. 2, February 1986, pp. 312-325.
- [Zhu89] Zhu, M. Y., "AUTOSTAR A Software Development System", *ACM SIGPLAN Notices*, vol. 24, no. 3, March 1989, pp. 31-45.

Appendix A

An MLog Solution to the Lift Control Problem

The following MLog code implements the solution to the lift control problem presented in Chapter eight. It prototypes the configuration where both the ISRs and the central controller communicate asynchronously with the lift managers.

```
% *****
%
%   This MLog code implements a prototype of a lift control system
%   containing 3 lifts and 5 floors. All button pushes are modeled as
%   interrupts. The following points should be considered when running it:
%
% 1) when letting people on, a lift will wait until a valid button is
%    pushed before doing anything else;
% 2) if a lift is letting people on and the floor button request was to
%    go up, all buttons below and including the current floor are ignored (the
%    symmetrical case is true when the floor button request was to go down):
%    the lift MUST travel in the direction specified by the floor button
%    request.
%
%   This code only runs with version 4.0 of the Sun operating system. It
%   can be run in "regular" mode or with timeslicing.
%
%   This version does asynchronous sends from the ISRs to the lift managers
%   and asynchronous sends from the central controller to the lift managers.
% *****
% -----
% -----
% *****
%
% class: object of nil.
%
% description: - holds methods which are used by all other classes
%
% *****
%
% class: object of nil.
%
% *****
%
% method: append(List1:<list>:R,
%               List2:<list>:R,
%               Result:<list>:U)
%
% description: - appends the contents of List2 to the end of List1 and places
%               the resulting list in Result
%               - can be used to check if Result is List2 appended to List1
%
% *****
```

```

append([],L,L).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).

% *****
%
% method: member(Element:<any>:U,
%           List:<list>:U)
%
% description: - succeeds if Element is an element of list List
%              - can be used to generate elements or lists or both
%
% *****

member(X,[X|_]).
member(X,[_|T]) :- member(X,T).

% *****
%
% method: efface(Element:<any>:U,
%              List1:<list>:U,
%              Result:<list>:U)
%
% description: - List1 is Result with the first occurrence of Element removed
%              - can be used to generate such lists
%
% *****

efface(A,[A|L],L).
efface(A,[B|L],[B|M]) :- efface(A,L,M).

% *****
%
% method: restoreButtons(ButtonIdList:<list of button object pointers>:R)
%
% description: - makes visible each button in ButtonIdList
%
% *****

restoreButtons([]).
restoreButtons([ButtonId|Tail]) :-
    ButtonId-makeTheValueOf(show,true),
    restoreButtons(Tail).

% -----
% -----
% *****
%
% class: floorIsr of interruptHandler.
%
% description: - implements the ISR for floor requests
%
% *****

class: floorIsr of interruptHandler.

% *****
%
% method: getIsrData(Data:<floor button request>:W)
%
% description: - reads the next floor button request from the floor data
%              buffer
%
% *****

getIsrData(Data) :- floorDataBuffer-read(Data).

% *****
%
% method: processInterrupt(Data:<floor button request>:K)
%
% description: - passes the next floor button request to the central
%              controller
%
% *****

processInterrupt(Data) :- performAsync(liftCont-floorButton(Data)).

% -----
% -----

```

```

% .....
%
% class: liftIsr of interruptHandler.
% description: - implements the ISR for lift requests
% .....

class: liftIsr of interruptHandler.

% .....
%
% method: getIsrData(Data:<floor button request>:W)
%
% description: - reads the next lift button request from the lift data buffer
% .....

getIsrData(Data) :- liftDataBuffer-read(Data).

% .....
%
% method: processInterrupt(Data:<floor button request>:R)
%
% description: - passes the next lift button request to the appropriate lift
%               manager
% .....

processInterrupt([ButtonId,1,Req]) :-
    performAsync(lift1-liftButton([ButtonId,Req])).
processInterrupt([ButtonId,2,Req]) :-
    performAsync(lift2-liftButton([ButtonId,Req])).
processInterrupt([ButtonId,3,Req]) :-
    performAsync(lift3-liftButton([ButtonId,Req])).

% -----
% -----
% .....
%
% class: interruptDataBuffer of object.
%
% description: - holds the data for pending interrupts in a FIFO queue until
%               the associated ISR reads it
% .....

class: interruptDataBuffer of object.

ivars: [data=[],interrupt].

% .....
%
% method: reset
%
% description: - resets the buffer
% .....

reset :- makeTheValueOf(data,[]).

% .....
%
% method: write(Data:<any>:W)
%
% description: - adds Data to the end of the data queue and generates the
%               appropriate interrupt
% .....

```

```

write(ButtonId,Data) :- ButtonId=makeTheValueOf(show,false),
                        theValueOf(data,OldData),
                        append(OldData,[ButtonId|Data],NewData),
                        makeTheValueOf(data,NewData),
                        theValueOf(interrupt,int),
                        int=generate,!.

% *****
%
% method: read(Data:<any>:R)
%
% description: - reads Data from the front of the queue
%              - fails if there is no data to read
%
% *****

read(Data) :- theValueOf(data,[Data|NewData]),
              makeTheValueOf(data,NewData).

% -----
% -----
%
% *****
%
% class: main of object
%
% description: - the highest level in the lift class hierarchy
%              - creates the graphics objects, the interrupts, the central
%              controller and the lift managers
%              - runs until the "END" button is pushed
%
% *****

class: main of object.

% *****
%
% method: begin
%
% description: - creates all the necessary objects and then waits for either
%              a 'write(_endsim)' or a 'write(_reset)' message
%              - terminates when a 'write(endsim)' message is received
%
% *****

begin :- myself(MainId),
        createLiftButtons(LiftFrame,LiftPanellist),
        createFloorButtons(FloorFrame,FloorPanellist),
        createSimControlButtons(MainId,SimContFrame,SimContPanellist),
        createLift1OutputWindow(Lift1Frame,Lift1Text),
        createLift2OutputWindow(Lift2Frame,Lift2Text),
        createLift3OutputWindow(Lift3Frame,Lift3Text),
        createIoStreams([Lift1Stream,Lift1Text],
                        [Lift2Stream,Lift2Text],
                        [Lift3Stream,Lift3Text]),
        displayWindows([LiftFrame,FloorFrame,SimContFrame,
                        Lift1Frame,Lift2Frame,Lift3Frame]),
        createLiftObjects([Lift1Stream,Lift2Stream,Lift3Stream]),
        interrupt-enableInterrupts,
        processButtons([Lift1Stream,Lift2Stream,Lift3Stream]),
        removeLiftObjects(end),
        removeIoStreams([Lift1Text,Lift1Frame,Lift1Stream],
                        [Lift2Text,Lift2Frame,Lift2Stream],
                        [Lift3Text,Lift3Frame,Lift3Stream]),
        removeAllPanels([LiftPanellist,FloorPanellist,SimContPanellist]),
        deleteWindows([LiftFrame,FloorFrame,SimContFrame]),
        interrupt-disableInterrupts,!.

% *****
%
% method: displayWindows(WindowList:<list of frame object pointers>:R)
%
% description: - displays each window pointed to by the frame object pointers
%              of WindowList
%
% *****

```

```

displayWindows([]).
displayWindows([Window|Tail]) :-
    Window-displayWindow,
    displayWindows(Tail).

% .....
%
% method: deleteWindows(WindowList:<list of frame object pointers>:R)
%
% description: - deletes each window pointed to by the frame object pointers
%               of WindowList
%
% .....

deleteWindows([]).
deleteWindows([Window|Tail]) :-
    frame-deleteWindow(Window),
    deleteWindows(Tail).

% .....
%
% method: removeAllPanels(Panellist:<list of panel object pointers>:R)
%
% description: - removes each panel in Panellist
%               - Panellist must be a list of panel lists
%
% .....

removeAllPanels([]).
removeAllPanels([Panel|Tail]) :-
    removePanels(Panel),
    removeAllPanels(Tail).

% .....
%
% method: removePanels(Panellist:<list of panel object pointers>:R)
%
% description: - removes each panel in Panellist
%
% .....

removePanels([]).
removePanels([H|T]) :-
    panel-removeInstance(H),
    removePanels(T).

% .....
%
% method: removeLiftObjects(Reason:<string>:R)
%
% description: - deletes all of the lift active objects and interrupts if
%               Reason is "end", resets them if Reason is "reset"
%
% .....

removeLiftObjects(end) :-
    liftController-removeInstance(liftCont),
    liftMgr-removeInstance(lift1),
    liftMgr-removeInstance(lift2),
    liftMgr-removeInstance(lift3),
    ivarreadInterrupt-removeInstance(floorInt),
    ivarreadInterrupt-removeInstance(liftInt),
    interruptDataBuffer-removeInstance(floorDataBuffer),
    interruptDataBuffer-removeInstance(liftDataBuffer).

removeLiftObjects(reset) :-
    interrupt-disableInterrupts,
    floorInt-reset,
    liftInt-reset,
    floorDataBuffer-reset,
    liftDataBuffer-reset,
    performAsync(liftCont-reset),
    performAsync(lift1-reset(1)),
    performAsync(lift2-reset(3)),
    performAsync(lift3-reset(5)),
    interrupt-enableInterrupts.

```

```

% .....
%
% method: write(ButtonId:<object>:R,
%           Data:<string>:R)
%
% description: - method called when the "END" and "RESET" buttons are pushed
% .....

write(_,_).

% .....
%
% method: processButtons(StreamList:<list of IO streams for the lifts>:R)
%
% description: - the wait loop of main which waits for button pushes
% .....

processButtons(StreamList) :-
    repeat,
    waitToAccept(Message),
    process(Message).

% .....
%
% method: process(Message:<any>:R)
%
% description: - handles the "END" and "RESET" button pushes
% .....

process(write(_,_endsim)).
process(write(_,_reset)) :-
    removeLiftObjects(reset),
    button-allInstances(ButtonList),
    restoreButtons(ButtonList),
    !,fail.

% .....
%
% method: createIoStreams(StreamList:<list>:U)
%
% description: - creates the IO streams for the individual lift windows
%              - StreamList has the form
%
%              [(Stream,Text)|Tail]
%
%              where Text is instantiated to the name of the 'textsw'
%              previously created and Stream is the name of the IO stream
%              created here (Text must be instantiated before the call,
%              Stream is instantiated by the call)
% .....

createIoStreams([]).
createIoStreams([(Stream,Text)|Tail]) :-
    terminalIoManager-openTerminalStream(Stream,passive,[]),
    Text=theValueOf(outStream,Temp),
    Stream=makeTheValueOf(outStream,Temp),
    createIoStreams(Tail).

% .....
%
% method: removeIoStreams(StreamList:<list>:R)
%
% description: - removes the individual lift IO streams
%              - StreamList has the following form
%
%              [(Text,Frame,Stream)|Tail]
%
%              where Frame is the name of the frame holding the 'textsw'
%              window designated by Text and Stream is the name of the IO
%              stream
% .....

```



```

removeIOStreams([]).
removeIOStreams([({Text,Frame,Stream}|Tail)) :-
    Text~resetTextsw,
    textsw~removeInstance(Text),
    frame~removeInstance(Frame),
    terminalIOManager~removeInstance(Stream),
    removeIOStreams(Tail).

% =====
%
% method: createLift1OutputWindow(Lift1Frame:<frame object pointer>:W,
%                               Lift1Text:<textsw object pointer>:W)
%
% description: - creates the output frame and textsw for lift1
%
% =====

createLift1OutputWindow(Lift1Frame,Lift1Text) :-
    frame~new(Lift1Frame,passive,[noConfirm=true,
        label="Lift 1 Status",xLocation=0,yLocation=415,
        height=100,width=300,boldLabel=true,textFrame=true]),
    textsw~new(Lift1Text,passive,[owner=Lift1Frame,textFile="lift1.txt"]).

% =====
%
% method: createLift2OutputWindow(Lift1Frame:<frame object pointer>:W,
%                               Lift1Text:<textsw object pointer>:W)
%
% description: - creates the output frame and textsw for lift2
%
% =====

createLift2OutputWindow(Lift2Frame,Lift2Text) :-
    frame~new(Lift2Frame,passive,[noConfirm=true,
        label="Lift 2 Status",xLocation=0,yLocation=515,
        height=100,width=300,boldLabel=true,textFrame=true]),
    textsw~new(Lift2Text,passive,[owner=Lift2Frame,textFile="lift2.txt"]).

% =====
%
% method: createLift3OutputWindow(Lift1Frame:<frame object pointer>:W,
%                               Lift1Text:<textsw object pointer>:W)
%
% description: - creates the output frame and textsw for lift3
%
% =====

createLift3OutputWindow(Lift3Frame,Lift3Text) :-
    frame~new(Lift3Frame,passive,[noConfirm=true,
        label="Lift 3 Status",xLocation=0,yLocation=615,
        height=100,width=300,boldLabel=true,textFrame=true]),
    textsw~new(Lift3Text,passive,[owner=Lift3Frame,textFile="lift3.txt"]).

% =====
%
% method: createLiftObjects(StreamList:<IO stream list>:R)
%
% description: - creates the lift controller, the interrupt data buffer,
%              the floor and button interrupts, and the lift managers
%
% =====

createLiftObjects([Lift1Stream,Lift2Stream,Lift3Stream]) :-
    liftController~new(liftCont,active,[liftdata1=[1,idle,-1,idle],
        liftdata2=[3,idle,-1,idle],
        liftdata3=[5,idle,-1,idle]]),
    liftMgr~new(lift1,active,[lift=1,data=[1,idle,-1,idle],
        state=idle,screen=Lift1Stream]),
    liftMgr~new(lift2,active,[lift=2,data=[3,idle,-1,idle],
        state=idle,screen=Lift2Stream]),
    liftMgr~new(lift3,active,[lift=3,data=[5,idle,-1,idle],
        state=idle,screen=Lift3Stream]),
    interruptDataBuffer~new(floorDataBuffer,passive,[interrupt=floorInt]),
    interruptDataBuffer~new(liftDataBuffer,passive,[interrupt=liftInt])S {},
    ivarreadInterrupt~new(floorInt,passive,[priority=15,ivar=[_,floorButton],
        enabled=true,howqueued=[queued,5],
        isr=[floorIsr,[]]]),
    ivarreadInterrupt~new(liftInt,passive,[priority=10,ivar=[_,liftButton],
        enabled=true,howqueued=[queued,5],
        isr=[liftIsr,[]]]).

```

```

% *****
%
% method: createLiftButtons(LiftFrame:<object pointer>:M,
%                           LiftPanelList:<panel object pointers>:M)
%
% description: - creates the lift buttons for each lift
%
% *****

createLiftButtons(LiftFrame, [Lift1Panel, Lift2Panel, Lift3Panel]) :-
    frame-new(LiftFrame, passive, [noConfirm=true, label="Lift Buttons",
    xLocation=0, yLocation=0, height=160, width=300,
    boldLabel=true, textFrame=false]),
    panel-new(Lift1Panel, passive, [owner=LiftFrame, percentWidth=33]),
    createNumberButtons(Lift1Panel, 5, 5, 1),
    panel-new(Lift2Panel, passive, [owner=LiftFrame, percentWidth=33]),
    createNumberButtons(Lift2Panel, 5, 5, 2),
    panel-new(Lift3Panel, passive, [owner=LiftFrame, percentWidth=34]),
    createNumberButtons(Lift3Panel, 5, 5, 3).

% *****
%
% method: createAButton(Owner:<name>:R,
%                       XLocation:<integer>:R,
%                       YLocation:<integer>:R,
%                       Label:<string>:R,
%                       Object:<object>:R,
%                       Data:<list>:R)
%
% description: - creates a button owned by Owner at (XLocation,YLocation)
%               called Label which executes Object-write(ButtonId,Data)
%               it is pushed
%
% *****

createAButton(Owner, XLocation, YLocation, Label, Object, Data) :-
    button-new(ButtonId, passive, [owner=Owner, xLocation=XLocation,
    yLocation=YLocation, labelImage=Label, buttonWidth=8,
    ventMessage=(Object-write(ButtonId,Data))]).

% *****
%
% method: createNumberButtons(Owner:<name>:R,
%                             Count:<button number (integer)>:R,
%                             XLocation:<integer>:R,
%                             YLocation:<integer>:R,
%                             LiftNumber:<integer>:R)
%
% description: - creates the numbered lift buttons for lift LiftNumber
%
% *****

createNumberButtons(_, 0, _, _, _).
createNumberButtons(Owner, Count, XLocation, YLocation, Lift) :-
    convert(Count, string, CountString),
    createAButton(Owner, XLocation, YLocation, CountString,
    liftDataBuffer, [Lift, Count]),
    NewCount is Count - 1,
    NewYLocation is YLocation + 25,
    createNumberButtons(Owner, NewCount, XLocation, NewYLocation, Lift).

% *****
%
% method: createFloorButtons(FloorFrame:<frame object pointer>:M,
%                             FloorPanelList:<panel object pointers>:M)
%
% description: - creates the Floor buttons by instantiating a frame for
%               them (FloorFrame) and then by creating 1 panel for each
%               Floor (FloorPanelList)
%
% *****

```

```

createFloorButtons(FloorFrame,
(Floor1Panel,Floor2Panel,Floor3Panel,Floor4Panel,Floor5Panel)) :-
frame-new(FloorFrame,passive,[noConfirm=true,label="Floor Buttons",
xlocation=0,ylocation=155,height=200,width=300,
boldLabel=true,textFrame=false]),
panel-new(Floor5Panel,passive,[owner=FloorFrame,percentHeight=18]),
createDownButton(Floor5Panel,105,7,5),
panel-new(Floor4Panel,passive,[owner=FloorFrame,percentHeight=18]),
createUpButton(Floor4Panel,27,7,4),
createDownButton(Floor4Panel,177,7,4),
panel-new(Floor3Panel,passive,[owner=FloorFrame,percentHeight=18]),
createUpButton(Floor3Panel,27,7,3),
createDownButton(Floor3Panel,177,7,3),
panel-new(Floor2Panel,passive,[owner=FloorFrame,percentHeight=18]),
createUpButton(Floor2Panel,27,7,2),
createDownButton(Floor2Panel,177,7,2),
panel-new(Floor1Panel,passive,[owner=FloorFrame,percentHeight=22]),
createUpButton(Floor1Panel,105,9,1).

% *****
%
% method: createUpButton(Owner:<name>:R,
%                       Xlocation:<integer>:R,
%                       Ylocation:<integer>:R,
%                       Floor:<integer>:R)
%
% description: - create an UP button at the specified location for the
%              specified Floor
% *****

createUpButton(Owner,Xlocation,Ylocation,Floor) :-
createAButton(Owner,Xlocation,Ylocation,"UP",
floorDataBuffer,[Floor,up]).

% *****
%
% method: createDownButton(Owner:<name>:R,
%                          Xlocation:<integer>:R,
%                          Ylocation:<integer>:R,
%                          Floor:<integer>:R)
%
% description: - create a DOWN button at the specified location for the
%              specified Floor
% *****

createDownButton(Owner,Xlocation,Ylocation,Floor) :-
createAButton(Owner,Xlocation,Ylocation,"DOWN",
floorDataBuffer,[Floor,down]).

% *****
%
% method: createSimControlButtons(MainId:<name>:R,
%                                 Frame:<object pointer>:M,
%                                 PanelList:<object pointers>:M)
%
% description: - creates the simulation control buttons, instantiating the
%              frame into Frame and the list of panels into PanelList
% *****

createSimControlButtons(MainId,SimContFrame,[SimContPanel]) :-
frame-new(SimContFrame,passive,[noConfirm=true,
label="Simulation Control Buttons",
xlocation=0,ylocation=355,height=60,width=300,
boldLabel=true,textFrame=false]),
panel-new(SimContPanel,passive,[owner=SimContFrame]),
createAButton(SimContPanel,27,10,"END",MainId,endsim),
createAButton(SimContPanel,177,10,"RESET",MainId,reset).

% -----
%
% *****
%
% class: liftController of main
%
% description: - this class implement the central controller of the system
%
% *****

```

```

class: liftController of object.
% .....
%
% The instance variables for the central controller:
%
% unallocated: the unallocated requests queue (requests which haven't been
%               assigned to a lift)
% liftdata1: the lift status of lift 1, in the following form:
%
%               [CurrentFloor,CurrentDirection,
%               DestinationFloor,DestinationDirection]
%
% liftdata2: same as liftdata1 but for lift 2
% liftdata3: same as liftdata1 but for lift 3
% .....

ivars: [unallocated={},liftdata1,liftdata2,liftdata3].

% .....
%
% method: floorButton(Req:<floor button request>:R)
%
% description: - called when the user pushes a floor button
%
% .....

floorButton(Req) :-
    findLift(Req,Lift),
    sendToLift(Req,Lift).
floorButton(Req) :-
    addToUnallocated(Req).

% .....
%
% method: reset
%
% description: - called when the system is being reset
%
% .....

reset :- makeTheValueOf(unallocated,[]),
        makeTheValueOf(liftdata1,[1,idle,-1,idle]),
        makeTheValueOf(liftdata2,[3,idle,-1,idle]),
        makeTheValueOf(liftdata3,[5,idle,-1,idle]).

% .....
%
% method: findLift(Req:<floor button request>:R,
%               Lift:<integer>:M)
%
% description: - succeeds if a lift can process Req: the number of the lift
%               is returned in Lift
%               - fails if no lift can currently process the request
%
% .....

findLift(Req,Lift) :-
    theValueOf(liftdata1,Ld1),
    theValueOf(liftdata2,Ld2),
    theValueOf(liftdata3,Ld3),
    getDistance(Req,Ld1,D1),
    getDistance(Req,Ld2,D2),
    getDistance(Req,Ld3,D3),
    findMin([Ld1],[Ld2],[Ld3],Lift).

% .....
%
% method: findMin([Lift1,Distance1]:<lift and distance>:R,
%               [Lift2,Distance2]:<lift and distance>:R,
%               [Lift3,Distance3]:<lift and distance>:R,
%               Lift:<integer>:M)
%
% description: - finds the lift with the minimum distance with respect to the
%               request being processed: succeeds and returns the lift
%               number in Lift if a lift can process the request
%               - fails if no lift can process the request (Distance1,
%               Distance2, and Distance3 are all -1)
%
% .....

```

```

findMin([L1,D1],[L2,D2],[L3,D3],Lift) :-
    min([L1,D1],[L2,D2],[Min1,Mind]),
    min([Min1,Mind],[L3,D3],[Lift,Dist]),!,
    Dist > -1.

% *****
%
% method: min([Lift1,Distance1]:<lift and distance>:R,
%           [Lift2,Distance2]:<lift and distance>:R,
%           [ClosestLift,ItsDistance]:<lift and distance>:W)
%
% description: - returns the [Lift,Distance] structure which has the smallest
%               Distance: if either distance is -1 (the lift can't process the
%               request), the distance of the other lift is returned because
%               in the worst case it will be -1 also
%
% *****

min([_,-1],[L2,D2],[L2,D2]).
min([L1,D1],[_,-1],[L1,D1]).
min([L1,D1],[L2,D2],[L1,D1]) :- D1 <= D2.
min([L1,D1],[L2,D2],[L2,D2]).

% *****
%
% method: getDistance(Req:<floor button request>:R,
%                   LiftData:<lift status>:R,
%                   Distance:<integer>:W)
%
% description: - succeeds and returns the distance away from the request if
%               the lift with the given LiftData can process the request
%               - succeeds but returns -1 in Distance if the lift cannot
%               process the request
%               - the first 6 clauses implement the central scheduling
%               algorithm:
%
%               1) the request is to go up from a floor above the current
%                  floor and the lift is moving up with no destination floor
%               2) the request is to go up from a floor above the current
%                  floor and the lift is moving up to process an on
%                  request to go up
%               3) the request is to go down from a floor below the current
%                  floor and the lift is moving down with no destination
%                  floor
%               4) the request is to go down from a floor below the current
%                  floor and the lift is moving down to process an on
%                  request to go down
%               5) the lift is idle and the floor is above the current
%                  floor
%               6) the lift is idle and the floor is below the current
%                  floor
%
% *****

getDistance([_,Floor,up],[Cfloor,up,-1,_],Dist) :-
    Floor > Cfloor,
    Dist is Floor - Cfloor.
getDistance([_,Floor,up],[Cfloor,up,_,up],Dist) :-
    Floor > Cfloor,
    Dist is Floor - Cfloor.
getDistance([_,Floor,down],[Cfloor,down,-1,_],Dist) :-
    Floor < Cfloor,
    Dist is Cfloor - Floor.
getDistance([_,Floor,down],[Cfloor,down,_,down],Dist) :-
    Floor < Cfloor,
    Dist is Cfloor - Floor.
getDistance([_,Floor,_,_],[Cfloor,idle,_,_],Dist) :-
    Floor >= Cfloor,
    Dist is Floor - Cfloor.
getDistance([_,Floor,_,_],[Cfloor,idle,_,_],Dist) :-
    Dist is Cfloor - Floor.
getDistance([_,_,-1]).

% *****
%
% method: addToUnallocated(Req:<floor button request>:R)
%
% description: - adds Req to the unallocated requests queue
%
% *****

```

```

addToUnallocated(Req) :-
    theValueOf(unallocated,Unallocated),
    append(Unallocated,[Req],NewUnallocated),
    makeTheValueOf(unallocated,NewUnallocated).

% .....
%
% method:  sendToLift(Req:<floor button request>:R,
%                Lift:<integer>:R)
%
% description: - sends Req to the lift whose lift number is Lift
%              - if the lift is currently idle, its status will be changed
%                to moving to prevent opposite direction floor requests from
%                being given to it
%
% .....

sendToLift(Req,Lift) :-
    liftName(Lift,String),
    performAsync(String-floorRequest(Req)),
    checkToSeeIfIdle(Lift,Req).

% .....
%
% method:  checkToSeeIfIdle(Lift:<integer>:R,
%                          Req:<floor button request>:R)
%
% description: - if the lift with lift number Lift is currently idle,
%               change its status so that it appears as if it is moving
%
% .....

checkToSeeIfIdle(Lift,[_,Floor,Direction]) :-
    liftData(Lift,String),
    theValueOf(String,[Cfloor,idle,_,_]),
    changeIdle(String,Floor,Cfloor,Direction).
checkToSeeIfIdle(_,_).

% .....
%
% method:  changeIdle(LiftDataName:<string>:R,
%                    Floor:<integer>:R,
%                    CurrentFloor:<integer>:R,
%                    NewDirection:<up or down>:R)
%
% description: - change a lift's status from idle to moving, setting its
%               destination floor and direction
%
% .....

changeIdle(String,Dfloor,Cfloor,Ddir) :-
    Dfloor > Cfloor,
    makeTheValueOf(String,[Cfloor,up,Dfloor,Ddir]).
changeIdle(String,Dfloor,Cfloor,Ddir) :-
    Dfloor < Cfloor,
    makeTheValueOf(String,[Cfloor,down,Dfloor,Ddir]).
changeIdle(String,Dfloor,Cfloor,Ddir) :-
    makeTheValueOf(String,[Cfloor,servicing,Dfloor,Ddir]).

% .....
%
% method:  checkUnallocated
%
% description: - checks each element of the unallocated requests queue and
%               sees if a lift can now process it: if not, it is returned
%               to the unallocated requests queue
%
% .....

checkUnallocated :-
    theValueOf(unallocated,Unallocated),
    makeTheValueOf(unallocated,[]),
    checkEachUnallocated(Unallocated).

```

```

% .....
%
% method: checkEachUnallocated(ReqList:<list of floor button requests>:R)
%
% description: - recursively checks elements of the unallocated requests queue
%               to see if a lift can process them
%               - calls floorButton to do the work: unallocated requests are
%               treated as if they just occurred
%
% .....

checkEachUnallocated([]).
checkEachUnallocated([Req|Tail]) :-
    floorButton(Req),
    checkEachUnallocated(Tail).

% .....
%
% method: report(Lift:<integer>:R,
%               LiftData:<lift status>:R,
%
% description: - called by lifts when they are reporting their new status
%
% .....

report(Lift,LiftData) :- liftData(Lift,String1),
    makeTheValueOf(String1,LiftData),
    checkUnallocated.

% .....
%
% method: liftData(Lift:<integer>:R,
%                 LiftDataName:<string>:M)
%
% description: - gets a lift's lift data instance variable name given its
%               lift number
%
% .....

liftData(1,liftdata1).
liftData(2,liftdata2).
liftData(3,liftdata3).

% .....
%
% method: liftName(Lift:<integer>:R,
%                 LiftName:<string>:M)
%
% description: - gets a lift's name given its lift number
%
% .....

liftName(1,lift1).
liftName(2,lift2).
liftName(3,lift3).

% -----
% -----

% .....
%
% class: liftMgr of main
%
% description: - this class implements the individual lift managers
%
% .....

class: liftMgr of main.

% .....
%
% The instance variables for the lift manager active objects:
%
% lift: the lift number
% screen: object pointer to the IO stream dedicated to this lift
% data: the current status of the lift in the following form:
%
%       [CurrentFloor,CurrentDirection,
%         DestinationFloor,DestinationDirection]
%
% onreqs: a list of outstanding on requests

```

```

% offreqs: a list of outstanding off requests
% state: the current state of the lift manager
%
% *****
ivars: (lift,screen,data,onreqs=[],offreqs=[],state).
%
% *****
%
% method: floorRequest(OnReq:<floor button request>:R)
%
% description: - called by the central controller when a floor request has
%              been assigned to this lift
%
% *****

floorRequest(OnReq) :-
    theValueOf(onreqs,OnReqs),
    append([OnReq],OnReqs,NewOnReqs),
    makeTheValueOf(onreqs,NewOnReqs),
    theValueOf(data,[Cfloor,Cdir,Dfloor,Ddir]),
    setData(OnReq,Cfloor,Cdir),
    setState(OnReq,Cfloor).

%
% *****
%
% method: liftButton(LiftButton:<list>:R)
%
% description: - the rendezvous entry to the lift manager called when one of
%              the lift's lift buttons has been pushed
%              - LiftButton has the following form:
%
%              [ButtonId:<integer>,Request:<floor #>]
%
% *****

liftButton([ButtonId,_]) :-
    theValueOf(state,idle),
    ButtonId-makeTheValueOf(show,true).
liftButton([ButtonId,Floor]) :-
    theValueOf(state,moving),
    processMovingEntry([ButtonId,Floor]).
liftButton(ButtonPush) :-
    theValueOf(state,servicing),
    processServiceEntry(ButtonPush).
liftButton(ButtonPush) :-
    theValueOf(state,waiting),
    theValueOf(offreqs,OffReqs),
    addLiftButton(ButtonPush,OffReqs,NewOffReqs),
    makeTheValueOf(offreqs,NewOffReqs),
    makeTheValueOf(state,moving).
liftButton([ButtonId,_]) :-
    theValueOf(state,waiting),
    ButtonId-makeTheValueOf(show,true).

%
% *****
%
% method: reset
%
% description: - resets the lift manager
%
% *****

reset(Floor) :- makeTheValueOf(state,idle),
               makeTheValueOf(onreqs,[]),
               makeTheValueOf(offreqs,[]),
               makeTheValueOf(data,[Floor,idle,-1,idle]).

%
% *****
%
% method: begin
%
% description: - the entry to the lift manager active object
%              - implements a state machine loop which prints the current
%              status and processes button pushes by the user
%
% *****

```



```

begin :- print_status,
        repeat,
        do_next_action,
        theValueOf(lift, Lift),
        theValueOf(data, LiftData),
        liftCont-report(Lift, LiftData),
        print_status,
        fail.

% *****
%
% method: print_status
%
% description: - outputs to the lifts's own output window the current status
%               of the lift
%
% *****

print_status :-
    theValueOf(state, idle),
    theValueOf(data, [Cfloor, _, _]),
    theValueOf(screen, Screen),
    Screen- (write("Idle at floor "), writeLn(Cfloor)), !.
print_status :-
    theValueOf(state, moving),
    theValueOf(data, [Cfloor, Cdir, _]),
    theValueOf(screen, Screen),
    Screen- (write("Moving "), write(Cdir), write(" from floor "),
            writeLn(Cfloor)), !.
print_status :-
    theValueOf(state, waiting),
    theValueOf(data, [Cfloor, _, _]),
    theValueOf(screen, Screen),
    Screen- writeLn("Waiting for destination floor"), !.
print_status :-
    theValueOf(state, servicing).

% *****
%
% method: do_next_action
%
% description: - the entry to the state machine algorithm of the individual
%               lift managers: one clause for each of the possible
%               states: idle, moving, servicing, and waiting
%
% *****

do_next_action :-
    theValueOf(state, idle),
    waitToAccept(Entry),
    (Entry = floorRequest(_);
     Entry = reset(_)), !.
do_next_action :-
    theValueOf(state, moving),
    getNextFloorTime(NextFloorTime),
    repeat,
    waitToAccept(_, absolute, NextFloorTime, _, processMovingEntry(at_floor), Result),
    Result = timeout, !.
do_next_action :-
    theValueOf(state, servicing),
    calcNextState(ButtonIdList),
    restoreButtons(ButtonIdList), !.
do_next_action :-
    theValueOf(state, waiting),
    waitToAccept(Entry),
    (Entry = liftButton(_);
     Entry = reset(_)), !.

% *****
%
% method: getNextFloorTime(Time:<time>:W)
%
% description: - gets the time when the lift will arrive at the next floor
%
% *****

```

```

getNextFloorTime([Hrs,Mins,Temp]) :-
    getTime([Hrs,Mins,Secs]),
    Secs < 55,
    Temp is Secs + 5.
getNextFloorTime([Hrs,Temp1,Temp2]) :-
    getTime([Hrs,Mins,Secs]),
    Mins < 59,
    Temp1 is Mins + 1,
    getSecs(Secs,Temp2).
getNextFloorTime([Temp1,0,Temp2]) :-
    getTime([Hrs,Mins,Secs]),
    Temp1 is Hrs + 1,
    getSecs(Secs,Temp2).

% *****
%
% method:  getSecs(CurrentSecs:<integer>:R,
%              NextFloorSecs:<integer>:W)
%
% description: - gets the next floor's seconds given the current seconds
%
% *****

getSecs(55,0).
getSecs(56,1).
getSecs(57,2).
getSecs(58,3).
getSecs(59,4).

% *****
%
% method:  setData(OnReq:<floor button request>:R,
%                  Cfloor:<integer>:R
%                  Cdir:<idle, up, or down>:R)
%
% description: - sets the lift's lift data according to the request in OnReq
%
% *****

setData([_,Floor,Dir],Cfloor,idle) :-
    Floor >= Cfloor.
    makeTheValueOf(data,[Cfloor,up,Floor,Dir]).
setData([_,Floor,Dir],Cfloor,idle) :-
    makeTheValueOf(data,[Cfloor,down,Floor,Dir]).
setData([_,Floor,Dir],Cfloor,up) :-
    makeTheValueOf(data,[Cfloor,up,Floor,Dir]).
setData([_,Floor,Dir],Cfloor,down) :-
    makeTheValueOf(data,[Cfloor,down,Floor,Dir]).

% *****
%
% method:  setState(OnReq:<floor button request>:R,
%                  Cfloor:<integer>:R)
%
% description: - sets the new state from the obtained on request
%
% *****

setState([_,Cfloor,_],Cfloor) :-
    theValueOf(state,idle),
    makeTheValueOf(state,servicing).
setState([_,_]) :-
    theValueOf(state,idle),
    makeTheValueOf(state,moving).
setState([_,Cfloor,_],Cfloor) :-
    theValueOf(state,moving),
    makeTheValueOf(state,servicing).
setState([_,_]).

% *****
%
% method:  processServiceEntry(ButtonPush:<ButtonId & {floor #}>:R)
%
% description: - called when a button has been pushed while servicing a floor
%
% *****

```

```

processServiceEntry(ButtonPush) :-
    theValueOf(offreqs, OffReqs),
    addLiftButton(ButtonPush, OffReqs, NewOffReqs),
    makeTheValueOf(offreqs, NewOffReqs).
processServiceEntry(_).

% .....
%
% method: calcNextState(ButtonIdList:<list of button object pointers>:N)
%
% description: - calculates the next state as a floor is about to be serviced
%              - returns in ButtonIdList a list of buttons which should
%              be made visible when the floor has been serviced
%
% .....

calcNextState([OnButtonId]) :-
    theValueOf(data, [Cfloor, _, Dfloor, Ddir]),
    theValueOf(onreqs, OnReqs),
    theValueOf(offreqs, OffReqs),
    member([OnButtonId, Cfloor, _], OnReqs),
    not(member([_, Cfloor], OffReqs)),
    processOnOnly(Cfloor, Dfloor, Ddir, OnReqs).
calcNextState([OffButtonId]) :-
    theValueOf(data, [Cfloor, Cdir, _, _]),
    theValueOf(onreqs, OnReqs),
    theValueOf(offreqs, OffReqs),
    member([OffButtonId, Cfloor], OffReqs),
    not(member([_, Cfloor, _], OnReqs)),
    processOffOnly(Cfloor, Cdir, OnReqs, OffReqs).
calcNextState([OffButtonId, OnButtonId]) :-
    theValueOf(data, [Cfloor, _, Dfloor, Ddir]),
    theValueOf(offreqs, OffReqs),
    efface([OffButtonId, Cfloor], OffReqs, NewOffReqs),
    makeTheValueOf(offreqs, NewOffReqs),
    theValueOf(screen, Screen),
    Screen-(write("Letting people off at floor "), writeLn(Cfloor)),
    theValueOf(onreqs, OnReqs),
    member([OnButtonId, Cfloor, _], OnReqs),
    processOnOnly(Cfloor, Dfloor, Ddir, OnReqs).

% .....
%
% method: processOnOnly(Cfloor:<integer>:R,
%                      Dfloor:<integer>:R,
%                      Ddir:<up or down>:R,
%                      OnReqs:<list of existing on requests>:R)
%
% description: - processes only an on request by removing the on request
%              from the list of on requests to be processed, outputs an
%              appropriate message to the output window, updates the lift's
%              data, and sets the state to moving because people are being
%              let on: the lift must move with people in it
%
% .....

processOnOnly(Cfloor, Dfloor, Ddir, OnReqs) :-
    efface([_, Cfloor, _], OnReqs, NewOnReqs),
    makeTheValueOf(onreqs, NewOnReqs),
    theValueOf(screen, Screen),
    Screen-(write("Letting people on at floor "), writeLn(Cfloor)),
    setNewData(Cfloor, Dfloor, Ddir),
    ((theValueOf(offreqs, {}),
      makeTheValueOf(state, waiting)),
     makeTheValueOf(state, moving)).

% .....
%
% method: setNewData(Cfloor:<integer>:R,
%                   Dfloor:<integer>:R,
%                   Ddir:<up or down>:R)
%
% description: - if necessary modifies the lift's lift data: this is necessary
%              if the lift is now at its destination floor: the destination
%              floor is set to -1 to indicate it is not moving to get to an
%              on request
%
% .....

```

```

setNewData(Floor,Floor,Ddir) :-
    makeTheValueOf(data,[Floor,Ddir,-1,Ddir]),
setNewData(.,.,.).

% .....
%
% method: processOffOnly(Cfloor:<integer>:R,
%                      Cdir:<up or down>:R,
%                      OnReqs:<list of existing on requests>:R,
%                      OffReqs:<list of existing off requests>:R)
%
% description: - processes only an off request by removing the request from
%               the list of existing off requests, outputting an appropriate
%               message to the output window, and by setting the lift data
%               and state as necessary
%
% .....

processOffOnly(Cfloor,Cdir,OnReqs,OffReqs) :-
    efface(.,Cfloor),OffReqs,NewOffReqs),
    makeTheValueOf(offreqs,NewOffReqs),
    theValueOf(screen.Screen),
    Screen-(write("Letting people off at floor "),writeIn(Cfloor)),
    setNewDataAndState(Cfloor,Cdir,OnReqs,NewOffReqs)

% .....
%
% method: setNewDataAndState(Cfloor:<integer>:R,
%                      Cdir:<up or down>:R,
%                      OnReqs:<list of existing on requests>:R,
%                      OffReqs:<list of existing off requests>:R)
%
% description: - sets the lift's lift data and state after an off request has
%               been processed
%               - the lift is idle if there are no more on or off requests
%               - the lift is moving without a destination floor if there are
%                 only off requests
%               - the lift retains its previous lift data and is moving if
%                 there is at least one on request (and maybe some off requests)
%
% .....

setNewDataAndState(Cfloor,.,[],[]) :-
    makeTheValueOf(data,[Cfloor,idle,-1,idle]),
    makeTheValueOf(state,idle),
setNewDataAndState(Cfloor,Cdir,[],[_]) :-
    makeTheValueOf(data,[Cfloor,Cdir,-1,Cdir]),
    makeTheValueOf(state,moving),
setNewDataAndState(.,.,.,.) :-
    makeTheValueOf(state,moving),

% .....
%
% method: processMovingEntry(Request:<"at_floor" or [ButtonId,Floor]>:R)
%
% description: - this method is called when a button push is received by the
%               lift manager while the lift is moving or if the lift reaches
%               the next floor
%               - a liftButton may be added to the list of off requests
%
% .....

processMovingEntry(at_floor) :-
    theValueOf(data,[Cfloor,Cdir,Dfloor,Ddir]),
    nextFloor(Cfloor,Cdir,NextFloor),
    makeTheValueOf(data,[NextFloor,Cdir,Dfloor,Ddir]),
    theValueOf(onreqs,OnReqs),
    member(.,NextFloor,.,OnReqs),
    makeTheValueOf(state,servicing),
processMovingEntry(at_floor) :-
    theValueOf(data,[Cfloor,.,.,.]),
    theValueOf(offreqs,OffReqs),
    member(.,Cfloor,OffReqs),
    makeTheValueOf(state,servicing),
processMovingEntry(at_floor),
processMovingEntry(ButtonPush) :-
    theValueOf(offreqs,OffReqs),
    addLiftButton(ButtonPush,OffReqs,NewOffReqs),
    makeTheValueOf(offreqs,NewOffReqs),
processMovingEntry([ButtonId,._]) :-
    ButtonId-makeTheValueOf(show,true).

```

```

% *****
%
% method: addLiftButton(ButtonPush:<lift button & floor>:R,
%               OffReqs:<list of off requests>:R,
%               NewOffReqs:<list of off requests>:M)
%
% description: - adds the given lift button to the list of off requests given
%               by OffReqs and puts the resulting list in NewOffReqs
%               - a button is valid in any of the following 6 cases:
%
%               1) moving up with no destination floor and the button is
%                  greater than the current floor
%               2) moving down with no destination floor and the button is
%                  less than the current floor
%               3) moving up with a destination direction of up and the
%                  button is greater than the current floor
%               4) moving down with a destination direction of down and the
%                  button is less than the current floor
%               5) moving up with a destination direction of down and the
%                  button is between the current floor and the destination
%                  floor
%               6) moving down with a destination direction of up and the
%                  button is between the current floor and the destination
%                  floor
%
% *****

addLiftButton([ButtonId,Floor],OffReqs,NewOffReqs) :-
    theValueOf(data,[Cfloor,up,-1,_]),!,
    Floor > Cfloor,
    append([ [ButtonId,Floor] ],OffReqs,NewOffReqs).
addLiftButton([ButtonId,Floor],OffReqs,NewOffReqs) :-
    theValueOf(data,[Cfloor,down,-1,_]),!,
    Floor < Cfloor,
    append([ [ButtonId,Floor] ],OffReqs,NewOffReqs).
addLiftButton([ButtonId,Floor],OffReqs,NewOffReqs) :-
    theValueOf(data,[Cfloor,up,_,up]),!,
    Floor > Cfloor,
    append([ [ButtonId,Floor] ],OffReqs,NewOffReqs).
addLiftButton([ButtonId,Floor],OffReqs,NewOffReqs) :-
    theValueOf(data,[Cfloor,down,_,down]),!,
    Floor < Cfloor,
    append([ [ButtonId,Floor] ],OffReqs,NewOffReqs).
addLiftButton([ButtonId,Floor],OffReqs,NewOffReqs) :-
    theValueOf(data,[Cfloor,up,Dfloor,_]),
    Floor > Cfloor,
    Floor <= Dfloor,
    append([ [ButtonId,Floor] ],OffReqs,NewOffReqs).
addLiftButton([ButtonId,Floor],OffReqs,NewOffReqs) :-
    theValueOf(data,[Cfloor,down,Dfloor,_]),
    Floor < Cfloor,
    Floor >= Dfloor,
    append([ [ButtonId,Floor] ],OffReqs,NewOffReqs).

% *****
%
% method: nextFloor(Cfloor:<integer>:R,
%               Cdir:<up or down>:R,
%               NextFloor:<integer>:M)
%
% description: - calculate the next floor given the current floor and the
%               current direction
%
% *****

nextFloor(Cfloor,up,NextFloor) :- NextFloor is Cfloor + 1.
nextFloor(Cfloor,down,NextFloor) :- NextFloor is Cfloor - 1.

```

END

1|2|0|4|9|1

FIN