1

PM-1 3 1/2" x 4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Notice to the user

Notice à l'utilisateur

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.

Visualizing Designs of Object-Oriented Real-Time Programs

by

Francois Pomerleau

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

Master of Engineering

Ottawa-Carleton Institute for Electrical Engineering
Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada, K1S 5B6
February 15, 1993

© copyright
1993 Francois Pomerleau

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Non file - Not to be filed

Non file - Not to be filed

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

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

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

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

ISBN 0-315-84112-5

Canada

Name Dr. Anne L. Jones, Ed.D.

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

Design of electronics and electrical

SUBJECT TERM

0549

SUBJECT CODE

U·M·I

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture 0729
Art History 0377
Cinema 0960
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION

General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0277
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0598
Physical 0523

Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language
General 0679
Ancient 0289
Linguistics 0290
Modern 0291
Literature
General 0401
Classical 0294
Comparative 0295
Medieval 0297
Modern 0298
African 0316
American 0591
Asian 0305
Canadian (English) 0352
Canadian (French) 0355
English 0594
Germanic 0311
Latin American 0312
Middle Eastern 0315
Romance 0313
Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy 0422
Religion
General 0318
Biblical Studies 0321
Clergy 0319
History of 0320
Philosophy of 0322
Theology 0469

SOCIAL SCIENCES

American Studies 0323
Anthropology
Archaeology 0324
Cultural 0326
Physical 0327
Business Administration
General 0310
Accounting 0272
Banking 0770
Management 0454
Marketing 0338
Canadian Studies 0385
Economics
General 0501
Agricultural 0503
Commerce/Business 0505
Finance 0508
History 0509
Labor 0510
Theory 0511
Folklore 0358
Geography 0366
Gerontology 0351
History
General 0578

Ancient 0579
Medieval 0581
Modern 0582
Black 0328
African 0331
Asia, Australia and Oceania 0332
Canadian 0334
European 0335
Latin American 0336
Middle Eastern 0333
United States 0337
History of Science 0585
Law 0398
Political Science
General 0615
International Law and Relations 0616
Public Administration 0617
Recreation 0814
Social Work 0452
Sociology
General 0626
Criminology and Penology 0627
Demography 0938
Ethnic and Racial Studies 0631
Individual and Family Studies 0628
Industrial and Labor Relations 0629
Public and Social Welfare 0630
Social Structure and Development 0700
Theory and Methods 0344
Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture
General 0473
Agronomy 0285
Animal Culture and Nutrition 0475
Animal Pathology 0476
Food Science and Technology 0359
Forestry and Wildlife 0478
Plant Culture 0479
Plant Pathology 0480
Plant Physiology 0817
Range Management 0727
Wood Technology 0746
Biology
General 0306
Anatomy 0287
Biostatistics 0308
Botany 0309
Cell 0379
Ecology 0329
Entomology 0353
Genetics 0369
Limnology 0703
Microbiology 0410
Molecular 0307
Neuroscience 0317
Oceanography 0416
Physiology 0433
Radiation 0821
Veterinary Science 0778
Zoology 0472
Biophysics
General 0786
Medical 0760

EARTH SCIENCES

Biogeochemistry 0425
Geochemistry 0996

Geology 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
Health Sciences
General 0566
Audiology 0300
Chemotherapy 0992
Dentistry 0567
Education 0350
Hospital Management 0769
Human Development 0758
Immunology 0982
Medicine and Surgery 0564
Mental Health 0347
Nursing 0569
Nutrition 0570
Obstetrics and Gynecology 0380
Occupational Health and Safety 0354
Ophthalmology 0381
Pathology 0571
Pharmacology 0419
Pharmacy 0572
Physical Therapy 0382
Public Health 0573
Radiology 0574
Recreation 0575

Speech Pathology 0460
Toxicology 0383
Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences
Chemistry
General 0485
Agricultural 0749
Analytical 0486
Biochemistry 0487
Inorganic 0488
Nuclear 0738
Organic 0490
Pharmaceutical 0491
Physical 0494
Polymer 0495
Radiation 0754
Mathematics 0405
Physics
General 0605
Acoustics 0986
Astronomy and Astrophysics 0606
Atmospheric Science 0608
Atomic 0748
Electronics and Electricity 0607
Elementary Particles and High Energy 0798
Fluid and Plasma 0759
Molecular 0609
Nuclear 0610
Optics 0752
Radiation 0756
Solid State 0611
Statistics 0463
Applied Sciences
Applied Mechanics 0346
Computer Science 0984

Engineering
General 0537
Aerospace 0538
Agricultural 0539
Automotive 0540
Biomedical 0541
Chemical 0542
Civil 0543
Electronics and Electrical 0544
Heat and Thermodynamics 0348
Hydraulic 0545
Industrial 0546
Marine 0547
Materials Science 0794
Mechanical 0548
Metallurgy 0743
Mining 0551
Nuclear 0552
Packaging 0549
Petroleum 0765
Sanitary and Municipal 0554
System Science 0790
Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY

General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

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

Visualizing Designs of Object-Oriented Real-Time Programs

submitted by Francois Pomerleau, B. Eng.
in partial fulfillment of the requirements for
the degree of Master of Engineering.

Thesis Supervisor

Chair

Department of Systems and
Computer Engineering

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
April 16, 1993

Abstract

This thesis describes a *timethread-oriented* approach to help discovering reusable abstractions in object-oriented real-time systems. The approach uses *end-to-end* patterns of activities rippling through a system in response to external stimuli as the means for helping to discover reusable abstractions. In the *timethread-oriented* approach, related *end-to-end* patterns of activities are visually expressed to analyze their commonalities and specializations. This analysis helps identifying objects, as well as driving the design of classes from which these objects are constructed.

Case studies illustrate the applicability of the *timethread-oriented* approach and the use of conventional object-oriented techniques to enhance software reuse opportunities in real-time systems. In particular, the *timethread-oriented* approach is illustrated by abstracting the design of various data link protocols in a small communication system example (Chapter 4). The same example also illustrates how conventional object-oriented techniques may be used to ease the integration of objects in different tasking environments.

Finally, a discussion about the applicability of the *timethread-oriented* approach and the use of conventional object-oriented techniques in practical environments is included.

Acknowledgements

I would like to express my appreciation to Dr. R.J.A. Buhr (Department of Systems and Computer Engineering), my thesis supervisor, for his valuable suggestions and continuing encouragement during the creation of this thesis.

I would also like to thank Ron Casselman for his technical and moral support during numerous discussions we had about this thesis. Thanks to Greg Bond and Bill Foster for reviewing an early draft of this work.

Many thanks to Hlne, for her encouragement and for inciting me to return to university on a full time basis to complete this work.

I gracefully acknowledge the financial support provided by the DOORS¹ project that is jointly funded by Bell Northern Research and the National Science and Engineering Research Council (NSERC).

1. Design of Object-Oriented Real-Time Systems.

Table of Contents

Chapter 1	Introduction	1
1.1	Motivations	1
1.2	Thesis Objective	5
1.3	Thesis Scope	6
1.4	Approach	7
1.5	Thesis Contributions	8
1.6	Thesis Outline	9
Chapter 2	Background	10
2.1	Issues in Software Reuse	10
2.1.1	The Nature of Object Oriented Programming	12
2.2	The Nature of Real-Time Embedded Systems	15
2.3	Summary	16
2.4	Related Work	17
Chapter 3	Re-engineering a Non-OO Program	19
3.1	The Approach	19
3.1.1	The Network Simulator	21
3.2	Objectifying Virtual Circuit Managers	30
3.3	Objectifying Messages, Packets, and Frames	37
3.4	Virtual Circuit Nodes, Messages, Packets, and Frames as Objects	44
3.5	Summary	51
Chapter 4	Towards an Architectural Framework for Real-Time Systems	52
4.1	The Approach	52
4.1.1	The Message Transfer Utility System	54
4.2	Increasing Software Reuse in the MTU System	57
4.2.1	Creating Reusable Abstractions by Analyzing Timethreads	59
4.2.2	Creating Reusable Abstractions by Analyzing Interface Protocols	73
4.3	Towards an Implementation of the MTU System	76
4.3.1	Mapping to DRAGOON	79
4.4	Summary	90
Chapter 5	Conclusions	92
5.1	A Research Summary	92
5.2	Testing Strategies and Observations	93
5.3	Assessment of Stated Goals	94
5.4	Assessment of Engineering Value	95
5.5	Future Work	96

Table of Contents

Appendix A

A.1	Overview of the Ada-9x Language Proposal.....	102
A.2	Overview of the DRAGOON Language.....	106

Appendix B

B.1	Abstracting Data Link Protocols.....	110
B.2	Abstracting Framing Policies.....	113

List of Figures

Figure 1	Network Simulator as Cooperating Peers.....	21
Figure 2	Network Simulator Overview Diagram.....	23
Figure 3	Establishing a Connection	25
Figure 4	Using a Connection.....	26
Figure 5	Destroying a Connection	27
Figure 6	Sending Packets Protocol.....	29
Figure 7	Virtual Circuit Nodes as Passive Objects Inside VCMgrs	31
Figure 8	Original Ada-83 Program - VCMgrs with Nested Procedures.....	31
Figure 9	Ada-83 Program - RemoteConnect and LocalConnect Procedures.....	32
Figure 10	Ada-83 Code for Processing Packets.....	33
Figure 11	Virtual Circuit Node Objects Created Inside VCMgrs.....	34
Figure 12	Class Hierarchy for Virtual Circuit Nodes	34
Figure 13	Class Definitions for Virtual Circuit Nodes	35
Figure 14	Body of the SourceVCNodeClass.....	36
Figure 15	Message, Packet, and Frame Objects Flowing Between Layers	37
Figure 16	Class Hierarchy for Messages, Packets, and Frames.....	38
Figure 17	Class Definitions For Packet Classes.....	39
Figure 18	Messages, Packets, and Frames as Objects.....	40
Figure 19	Transforming a Frame to an ASCII String (Data Packets).....	41
Figure 20	Package Body for the DataPktClass - AsString Method	41
Figure 21	Package Body for Abstract Packet Class.....	42
Figure 22	Package Body for the DataPktClass - Parse Method.....	43
Figure 23	Recreating Messages, Packets, and Frames in the Physical Layer	44
Figure 24	OMT Class diagram - Node Types Versus Packet Types	45
Figure 25	VCNodes, Messages, Packets, and Frames as Objects.....	46
Figure 26	Ingalls's Technique to Resolve Doubly Polymorphic Situation.....	47
Figure 27	A Packet Object Interacting with a Virtual Circuit Nodes Object.....	48
Figure 28	Class Definitions for Virtual Circuit Nodes and Packets	49
Figure 29	Package Body for SourceVCNodeClass.....	50
Figure 30	Package Body for AckCreatePktClass.....	50
Figure 31	Files Transferred Between Nodes.....	54
Figure 32	Virtual and Physical Timethreads - Enveloping Files	55
Figure 33	End-To-End Behaviour Patterns of the MTU System.....	56
Figure 34	Sequences of Activities when Transmitting a File in the MTU	57
Figure 35	Objectifying the MTU System.....	58
Figure 36	Send-And-Wait Protocol.....	60

List of Figures

Figure 37	Sliding Window Go Back Protocol - Rx Window Greater Than One.....	61
Figure 38	Generalized Timethreads for Data Link Protocols	63
Figure 39	Interfaces of Data Link Protocol Objects	64
Figure 40	Decision Points for the Data Link Protocol Object	65
Figure 41	Data Link Protocols Class Diagram.....	66
Figure 42	Structure of the Data Link Protocol Abstract Class..	67
Figure 43	Structure of the Send-And-Wait protocol (Concrete) Class	68
Figure 44	Allocation of Timethreads to Processes.....	69
Figure 45	Putting the Data Link Protocol Object in an Actor.....	70
Figure 46	Encapsulating the Data Link Protocol in a Third Party	71
Figure 47	Separating the Data Link Protocol Object	72
Figure 48	Framing Policy - Starting and Ending Characters with Char. Stuffing	74
Figure 49	Abstracting Framing Policies.....	75
Figure 50	Adding Protocol Adapters Between Objects	75
Figure 51	Potential Objects in the MTU System	77
Figure 52	Annotated Timethreads Describing Dynamic Memory Allocation	78
Figure 53	Active Objects in the MTU System.....	79
Figure 54	DRAGOON - MTU Subsystem M	80
Figure 55	File System Behaviour by Readers and Writers	81
Figure 56	Subsystem M - Producer Object	82
Figure 57	DRAGOON - MTU Subsystem P.....	83
Figure 58	DRAGOON - Partial Implementation of MTU Subsystem P.....	84
Figure 59	DRAGOON - Abstract Data Link Protocol Class	85
Figure 60	DRAGOON - Send and Wait Protocol Class	86
Figure 61	DRAGOON - MTU Subsystem L	87
Figure 62	MTU Data and MTU Subsystem Class Hierarchy	88
Figure 63	DRAGOON - Class Specifications for MTU Subsystems	88
Figure 64	Initialization of the MTU System	89
Figure 65	DRAGOON - MTU System Body.....	89
Figure 66	Pre-defined Interfaces for Various Actors Interaction Structures.....	91
Figure 67	Ada-83 - Example of Variant Programming.....	103
Figure 68	Ada-9x - Variant Programming - Type Definitions.....	104
Figure 69	Ada-9x - Variant Programming - Code Body.....	105
Figure 70	DRAGOON - Behavioural Inheritance.....	107
Figure 71	DRAGOON - Defining Sequential and Behavioured Classes.....	107
Figure 72	DRAGOON - Common Behavioural Classes.....	108

List of Figures

Figure 73	DRAGOON - Active Object*	109
Figure 74	Sliding Window Go Back Protocol - RX Window Size Equals One	111
Figure 75	Sliding Window with Selective Repeats Protocol	112
Figure 76	Framing Policy- Character Count	113
Figure 77	Framing Policy - Simplified Bit Stuffing	114

Glossary of Notations

MachineCharts Notation

Box		Boxes represent aggregation. They map to programming constructs such as packages in Ada or modules in Modula-2, or architectural components such as subsystems or layers.
Actor		Actors are active boxes. They are autonomous, self-directing components. They map to tasks in Ada or tasks as supported by real-time executives.
Reactor		Reactors are a constrained version of actors. They are not autonomous self-directed components but provide protected services to other actor in a mutually exclusive manner. They map to semaphores or mailboxes in real-time executives.
Button		Buttons represent procedural abstractions, and map to re-entrant procedures or functions in programming languages. Buttons may be combined with boxes.
Gate		Gates are also procedural abstractions but impose synchronization among actors visiting them.
Store		Place for keeping data. They map to variables in programming languages.
Template		Symbolizes a set of plans for constructing a component.
Finger		Fingers are attached to buttons or gates.
Waiting Place Triggers		Connectors attached to gates specifying the controlling component for opening and closing the guard associated with a gate. Filled circle corresponds to closing the guard, the blank circle corresponds to opening the guard.
Install		The install arrow indicates the installation of an instance from a template.
Remove		The remove arrow indicates the explicit removal of an instance.
Waiting Place		The waiting place marker may be placed beside gates to indicate places where waiting may occur until some condition is satisfied. The condition may be described textually. In Ada, they map to guards associated with task entries.
Timer		The timer icon indicates that interactions are constrained with time.
Replicate		Indicates multiple instances of components.
Data Flow		The data flow icon indicates the transfer of data between components.
Object		Objects are passive data-centric components. They map to instances of classes in object-oriented programming languages.

Glossary of Notations

TimeThread Notation

A timethread links activities performed by the system from some stimulus and terminating with some eventual response. The *thread start* represents the location where a stimulus launches a timethread. The *thread end* represents the system's final response to the original stimulus. The path represents activities performed by the system in time and space resulting from the original stimulus.

A timethread launches another timethread (black) and continues its activities.

The waiting place represents a spot along a timethread where the timethread waits for an event to arrive. A waiting place marker is drawn as a large filled circle along a timethread.

The outgoing (black) timethread is launched after both incoming timethreads arrive.

The incoming (black) timethread launches both outgoing timethreads simultaneously. Both outgoing timethreads are concurrent.

The outgoing (black) timethread is launched when either of the incoming timethreads arrive. The first to arrive launches the outgoing one.

The incoming (black) timethread launches one of the outgoing timethreads under some condition. The condition may be specified textually.

The timer symbol represents a special waiting place along a timethread where a timethread may wait for a certain period of time.

The install and remove arrows symbols represent the location where components are dynamically installed and removed along a timethread.

The data flow symbol represents the movement of data along a thread. Instances of classes can also flow along timethreads.

The relationship symbol indicates the cause-effect relationships between timethreads.

Glossary of Notations

Object Modelling Technique Notation - Class Diagram

Class

A class describes a group of objects with similar properties, common behaviour, common relationships to other objects, and common semantics. A class may have attributes and operations. A class attribute is a data value held by the objects in a class, and are listed in the middle part of the class icon. A class operation is a function or transformation that may be applied to or by objects in a class, and are listed in the bottom part of the class icon.

Binary Association

Associations describe the relationships between classes and represent a group of links with common structure and common semantics. A link represents a physical or conceptual connection between objects. Associations may be binary, ternary, or higher order. A binary association connects two classes of objects, for example, a person *owns* a car. A ternary connects three classes, for example, a person *works-on* a project *using* a programming language.

Ternary Association

Multiplicity of Associations

Multiplicity specifies how many instances of one class may relate to an instance of another class. For example, many persons *work-for* a company, or a person *owns* many cars. Multiplicity constraints the number of objects in a relation.

Aggregation (Assembly)

Aggregation is a form of association in which an aggregate object is made of components. For example, a documents is made of paragraphs; paragraphs are made of sentences; and sentences are made of words. Aggregation is often referred as the *part-of* relationship.

Generalization (Inheritance)

Generalization (inheritance) is the relationship between a class and one or many versions of it. For example, a square is a kind of rectangle except that its height and width are equal. Generalization is often referred as the *is-a* relationship.

Chapter 1 Introduction

1.1 Motivations

“Developing software for embedded systems, in which the computers to be programmed are responsible for controlling encapsulating electromechanical systems, is one of the most difficult and challenging fields of software engineering. Not only is this type of software required to respond rapidly and reliably - in ‘real-time’ - to events in the external environment, but often has to execute under severe hardware and memory constraints. Moreover, embedded systems tend to be extremely large, and require sizable teams of people to program them.” - Colin Atkinson [1]

Large software systems, including real-time embedded ones, represent significant capital investments. As a result, large software systems tend to stay around and evolve over time. Many of today's large software systems were built from traditional approaches such as top-down functional decomposition. Using functional decomposition, the structure of the system is derived from the functions it performs. Meyer [39] states that even though top-down functional decomposition is a logical, well-organized discipline, it suffers from several flaws including: it fails to take into account the evolutionary nature of software systems; and it does not promote reusability. Object-oriented (OO) techniques are often advocated to promote software reuse and to ease the maintenance of software systems. Rumbaugh [43] states that, “the main benefit is not reduced development time; object-oriented development may take more time than conventional development, because it is intended to promote future reuse and reduce downstream errors and maintenance.”

Should OO techniques be employed for the development of large real-time embedded systems (LRT) to ease maintenance and promote reuse? OO literature suggests they should [1], [6] or sees no reasons why they should not be used in most software developments [43]. Often, OO literature gives examples on how to develop such systems by uniformly

applying the OO paradigm throughout the entire system development. However, we observe that OO literature rarely addresses two important aspects of large real-time embedded systems development: 1) *evolution* including the introduction of the OO paradigm in existing systems of which may not have been designed using the OO paradigm; and 2) *temporal issues* including competing multiple concurrent threads of control, or critical response time.

This thesis is part of a ongoing research project (DOORS) which explores ways of introducing OO techniques in the development of large real-time systems [11]. Many attempts to use the OO paradigm in the real-time domain have been made at the programming language level. For example, DRAGOON¹ [11] is an OO extension of the Ada language; Ada already supports constructs to deal with real-time issues but not all features of OO. Actra [52] is a multitasking/multiprocessing extension of Smalltalk; Actra constrains parallelism to large active components under which many small synchronous objects may exist. Other approaches such as the RTC++ language [32] are based on fine grain concurrency and incorporate constructs to deal with real-time constraints in the languages. This thesis emphasizes architectural issues over languages issues. By architecture, we mean ways to think about programs in terms of components such as modules, layers, tasks, and objects, and the way these components interact. By interactions between components, we mean a way to think about components in operation as the program executes rather than thinking about the file structure of a program as a compiler may see it. The work of Shaw [47] on large scale systems motivates emphasizing architectural issues over programming issues. Shaw states that: "Large-scale systems involve design at a level above the traditional module. At this design level the objects of interest are subsystems and their interactions, rather than data structures and algorithms". In the context of this thesis, we emphasize the modeling of systems which combine elements of OO design and real-time design in terms of components and communication paths, over the description of data structures and algorithms.

Visual design notations are important in diverse branches of engineering such as electrical, or civil, and are also important in software engineering. Standard well-defined notations permit the communication of design intent, ease the tackling of more advanced problems, and assist in the checking of the consistency and correctness of designs, perhaps through automated tools [6]. There are a significant number of visual design notations in the domain of software engineering [6], [19], [43], [54]. The criteria for selecting a good notation are subjective and hard to isolate, but basically have to do with simplicity, clarity,

1. An overview of the DRAGOON language can be found in Appendix A.1.

expressiveness, and the capability to capture the essence of designs. This thesis uses visual notations to describe architectural issues.

Complex software systems require many views [9], [11], [55]. Booch [6] states that, “It is impossible to capture all the subtle details of a complex software systems in just one kind of diagram”. For example, OO designers tend to think about OO programs in terms of objects, classes, class hierarchies, and class relationships and most OO visual notations provide different models to depict these abstractions. A program may be viewed in different models or mixes of models for different purposes. For example, a programmer may be interested in viewing the class hierarchies of a program for code reuse purposes whereas a designer may be interested in the tasking structure of a program for performance reasons. In this thesis, we combine elements of OO design and real-time design, creating situations where it may not be clear at first which model to use, and making the visualization of these situations difficult to do. We explore possibilities of visualization when combining elements of OO design and real-time design.

Real-time systems built around subsystems, modules, and layers may be relatively static in nature. In these cases, visual design notations, such as the MachineCharts notation [8], depict systems in terms of components and communication paths. OO programs may be highly dynamic [3], [6]. We consider three factors contributing to make OO programs dynamic as follows:

Firstly, components in OO programs are often created and destroyed during the execution of the program. For example, in window based human interfaces, a program may create a window to prompt the user and destroy it afterward. Booch [6] states that, “classes are largely static in the design of a system, whereas objects are much more transitory, in that many of them may be created and destroyed during the execution of a single program”. Bailey [3] also remarks that OO systems tend to be highly dynamic, with instances created and destroyed over the lifetime of the system. Bailk γ adapts an earlier version of the MachineCharts notation called Buhr diagrams [7] to express OO concepts, and remarks that his iconic notation presents a somewhat limited view of the structure of OO systems, each diagram representing a snapshot in time of the system; relatively little temporal information is shown. Bailey indicates that a means for connecting a sequence of the snapshots to provide an easily understood description of the temporal behaviour of OO systems would be desirable.

Secondly, OO programs are often built around the concept of polymorphism, which often makes the communication paths between components dynamic in nature (run-time binding). For example, a device driver may invoke a display method over a

collection of heterogeneous graphical shapes. The selection of the appropriate piece of code depends on the type of graphic elements and may occur at run-time. From a programming viewpoint, run-time binding may be considered as a feature of OO languages that is especially useful for maintenance purposes. From an architectural viewpoint, programs are viewed in terms of components and communication paths; it may be important to characterize the nature of communication paths in relation to the fact that they are statically or dynamically established [9]. We define *statically-bound* components those that have fixed and statically established communication paths such as often occurring in real-time systems built around subsystems, modules and layers. We also define *dynamically-bound* components those that have temporary and dynamically established communication paths such as often occurring in OO programs which use run-time binding. However, there may not be a one to one mapping between the use of run-time binding in an OO program and its architectural model. For example, an OO program may use run-time binding sporadically where run-time binding is used during configuration phases only. Between these phases, it may be better to think about the program in terms of *statically-bound* components. A real-time program may be conceived as a collection of *statically-bound* components but implemented in an OO programming language using run-time binding. We observe that choosing a model for a program in terms of *statically-bound* or *dynamically-bound* components or a mix of them seems to be case dependant.

Thirdly, components in OO programs often play different roles during the execution of a program. For example, in a model capturing the relationships between teachers and students, a person may act as a teacher during the day, and the same person may act as a student by taking a course at night. Often, in the OO literature, examples such as teachers and students are captured using models similar to or extensions of entity relationship (ER) [18] diagrams. In these cases, such a relatively static view of a design may be adequate and sufficient but in the real-time domain, showing components in operation is also important. One may imagine depicting the example with components in operation, in this case, a set of diagrams showing components and communication paths could be used to show different scenarios such as the person acting as a student at night, or the same person acting as a teacher during the day. In this example, the relative infrequent variation of roles of the person permits one to depict the components in operation using a relatively small set of diagrams. Extending the same example to cases where the same person answers questions from different people residing in different locations yields more complicated scenarios. The person may answer differently depending upon who is asking the question, either the teacher during the night course or one the students during the day; and yet answer differently when the person is asked a question informally in a corridor or

in an examination room. In the extended example, the variation of roles of the person may depend not only on the time, but on the individual with whom they are interacting or the place where the question is asked or a combination of these factors. Depicting the components in operation for the extended example would require a significant number of diagrams, each diagram representing a snapshot in time of a particular set of components connected by a particular set of communication paths. When pushed to a limit, the variation of role playing of components may not just happen at some infrequent points in time (as in the first example mentioned above) but continuously (as the extended example suggests). Among others, recent work on contracts [30] has inspired the work of Buhr [9] who suggests that it may not be useful to depict these situations in terms of components and communication paths; to draw them, it may be necessary to move to a higher level of abstraction in terms of contracts and roles.

This thesis is concerned with visualizing designs of object-oriented real-time (OORT) programs. Given the importance of software reuse, this thesis is also concerned with exploring ways to enhance software reuse in real-time systems. OO techniques are often advocated to promote software reuse but do not guarantee it. OO programs must be designed for reusability [34]. One important aspect of software reuse is the discovery of reusable abstractions. Finding reusable abstractions is difficult, and in general, it seems that an abstraction is usually discovered by generalizing from a number of concrete examples. An approach for helping real-time designers to discover reusable abstractions while focusing on real-time issues appears desirable. Developing such approach is the prime motivation of the work of this thesis.

1.2 Thesis Objective

The objective of the thesis is to contribute (in the context of an ongoing research project-DOORS) to the development and illustration of a *timethread-oriented* approach for helping designers to discover reusable abstractions in real-time systems.

Together with the prime objective, sub-objectives are pursued as follows:

- 1) To illustrate the use of different visual design notations in a coordinated manner to depict OORT programs.
- 2) To illustrate the use of conventional OO techniques to enhance software reuse in real-time systems.
- 3) To emphasize the re-engineering aspect of software development given that OO literature rarely addresses the introduction of the OO paradigm in existing systems.

1.3 Thesis Scope

There is no consensus from experts in the OO community on the exact characteristics required for an object-oriented approach to software development [6], [41], [43]. In the context of this thesis, we consider the basic characteristics as polymorphism, inheritance, and encapsulation.

This thesis is concerned with visualizing programs in operation. The programming examples contain architectural components that have very limited variations of role playing during the execution of the program and use a modest degree of polymorphism (dynamic binding with simple patterns of interactions between components). The limited variation of role playing and the modest degree of dynamic binding enables the visualization of the programs in terms of components and communication paths without moving to a higher level of abstraction².

We observe that combining elements of OO design and real-time design creates situations where it may not be clear, at first, which architectural model (*statically-bound* or *dynamically-bound* components) or a mix of them should be used to model these situations. Defining cookbook recipes for choosing one model over another for a specific program is beyond our reach at present. Through a set of examples, we explore ways of depicting these situations, and based on our observations, we suggest some guidelines regarding the selection of a model or a mix of models depicting programs that combine elements of OO design and real-time design.

We use visual notations to cover the scope of our studies:

- We use a visual notation that is strong at depicting real-time issues, namely the MachineCharts notation. We annotate conventional MachineCharts structure diagrams to accommodate the visualization of OO concepts.
- We use a visual notation strong at depicting *end-to-end* behavioural patterns, namely the Timethreads notation. A timethread is a path traced over some view of a system to show behaviour. We use timethreads over conventional MachineCharts diagrams to describe end-to-end behaviour of OORT programs.
- And, for explanatory purposes and to provide a complementary view of OORT programs, we adopt one ER-like notation, namely Rumbaugh's OMT. We choose

2. A parallel thesis by Casselman explores ways of visualizing systems such as those exhibited by OO programs built around frameworks which contain components that have highly variable roles. The approach is based on the concepts of roles, contracts, and unethreads.

Rumbaugh's notation because it provides a clean graphic representation of the relationships between entities in the problem domain including *is-a* (inheritance) and *using* relationships. In this context, we do not need to do an exhaustive comparison of existing textbook OO notations because we utilize a restricted set of OO elements that most OO notations are capable of depicting. We use Rumbaugh's OMT class diagrams in a coordinated manner with the MachineCharts notation to express designs of OORT programs.

1.4 Approach

The approach taken in this thesis for achieving its objectives is as follows:

- Study different documented programming examples and design examples from the object-oriented domain and the real-time domain for discovering representative design patterns in both domains.
- Experiment with conventional OO techniques in the examples from the real-time domain for finding ways of incorporating OO techniques in real-time programs.
- Experiment with visual notations to express real-time concerns, OO concerns, and end-to-end behaviour in OORT programs for representing designs of OORT programs.
- Analyze the programming and design examples from the real-time domain for finding ways of discovering software reuse opportunities. This includes: 1) *programmatic* aspects (e.g., code reuse through class inheritance); and 2) *architectural* aspects (e.g., component reuse through techniques that ease the integration of operational components in different environments).

These activities occur in the context of two case studies:

- The first case study uses an existing program from the real-time domain. The program is modified in steps by applying OO techniques on it. The existing program is written in Ada-83 and the modifications to the program are mapped to Ada-9x. Ada-9x is a proposed revision of the Ada-83 language to include, among others, full object-oriented programming facilities [59]. The case study investigates the use of OO techniques in real-time system by objectifying data elements to enhance software reuse at the level of individual components.
- The second case study starts from existing specifications and design alternatives for a small-scale communication system example [13], [14] and investigates the use of

OO techniques in it to promote software reuse of larger structures than the previous study. These larger structures include individual components, as well as, algorithms and control code which coordinate these components. The study also explores ways of enhancing software reuse in real-time systems by applying OO techniques to ease the integration of objects (operational components) in different environments. The study provides a partial implementation of the communication system in an interesting OO real-time language called DRAGOON. DRAGOON treats objects as units of reuse and distribution and offers an interesting set of features for dealing with concurrency and synchronization issues.

1.5 Thesis Contributions

The prime contribution of this thesis is a set of illustrative examples in which various visualization techniques and OO techniques are applied to enhance software reuse in real-time system designs. Within this contribution, there are specific achievements:

- A technique based on timethreads to help discovering reusable abstractions in real-time systems (Section 4.2.1).

Other achievements are in the area of:

- 1) **Re-engineering software systems.** This includes illustrating the introduction of OO techniques in an existing real-time program (Chapter 3).
- 2) **Using visual techniques to express the designs of OORT programs.** This includes demonstrating the use of different textbook notations in a coordinated manner to express designs of OORT programs.
- 3) **Refining the timethread notation.** This includes annotating timethreads to reason about creation and destruction of instances in time and place from a system level perspective (Section 4.3).
- 4) **Mapping design diagrams to implementation technologies.** This includes the mapping of various design diagrams into two implementation technologies namely Ada-9x (Chapter 3) and DRAGOON (Chapter 4).

1.6 Thesis Outline

Chapter 2 summarizes the key issues and concepts relevant to this thesis.

Chapter 3 emphasizes the re-engineering aspects of software development and discusses the influence of applying OO techniques on the original program. The chapter presents ways of introducing the OO paradigm into an existing real-time design example by the means of gradually transforming its implementation program by applying OO techniques in steps. The chapter also explores ways of visualizing the design of OO real-time systems.

Chapter 4 explores the use of OO techniques to promote software reuse in real-time systems. The chapter investigates the use of timethreads as a means to discover reusable abstractions. The chapter also explores various OO techniques to increase software reuse in real-time systems.

Chapter 5 concludes the research and presents avenues of future work.

Chapter 2 Background

This chapter summarizes the key issues and concepts relevant to this thesis. This thesis uses existing visual design notations to depict designs of OORT programs and maps design diagrams to different object-oriented programming languages. Including a complete summary of these design notations and object-oriented programming languages in this chapter would hinder the flow of this thesis. Instead, we explain the notations and programming languages as we use them in the case studies (Chapter 3, Chapter 4). Nevertheless, readers are assumed to have basic knowledge about these design notations and about object-oriented programming languages in general. We list the design notation's icon sets used in this thesis in a glossary at the beginning of this thesis. We also summarize the object-oriented programming languages used in this thesis in Appendix A. Detailed descriptions of the design notations and object-oriented programming languages used in this thesis may be found in [1], [8], [15], [43], [57].

2.1 Issues in Software Reuse

In a broad sense, software reusability is the reapplication of different kinds of knowledge about one system to another similar system in order to reduce the effort of development and maintenance of that system [5]. Examples of kinds of knowledge include: domain knowledge, development experiences, design decisions, architectural structures, designs, code, etc.

There is a great diversity of software engineering activities that involve some form of software reuse [37]. In a survey of the field, Krueger [37] classifies up to eight different approaches to software reuse including: reuse of high-level language constructs, design and code scavenging, source code components, software schemas, application generators, and transformational systems. The survey emphasizes the importance of abstractions in all of the approaches. Krueger describes that abstraction is an essential feature in any reuse

technique; without abstractions, developers are forced to examine and analyze reusable artifacts to determine: what each artifact does, when it could be reused, and how to reuse it. Krueger concludes that raising abstraction levels may be difficult because useful abstractions for large, complex, reusable software artifacts will typically be complex [37]. To reuse these software artifacts, developers must either be already familiar with their abstractions, or take the time to understand them which may be difficult and time consuming (defeat the purpose of reuse).

Different approaches of software reuse occur in practical environments. For example, developers sometimes adopt an ad hoc approach which consists of scavenging fragments from existing software systems and use them as parts of the new software development. Scavenging is inherently limited because developers are not isolated from the details, they often work at the same level of abstraction as when they build software from scratch [37]. In the ideal case, programmers can adapt large fragments of source code without significant modification; in the worst case, they spend more time locating, understanding, and debugging the scavenged code fragments than writing them from scratch.

Another approach relates the reuse of source code components (often referred as *Software IC* [22]). Source code components such as packages, functions, or other forms of modules are written, tested, and stored for the purpose of reuse. This approach has obtained some success in narrow domains such as numerical computation, but has been less successful with collections of general-purpose components. One dilemma with this approach relates the size of components versus reuse potential. As a component increases in size, the payoff involved when reusing the component also increases; unfortunately, as it grows in size, it becomes more and more specific which reduces reuse potential and may increase the cost of modifying the component for reuse [5].

One important aspect of software reuse is reuse of design [5], [8]. Design reuse relates the reuse of design-level abstractions such as ideas about organizing peer components in cooperating teams, ideas about designing interfaces and internal structures of components, and ideas about interconnecting components in a system [8]. The payoff potential for reuse of design is high because even when the code itself is not reusable, the general design ideas are often quite reusable. For example, senior designers often exploit previous experiences by reusing patterns of designs (i.e., pre-structured design information) to tackle new problems. One obstacle to reuse of design is that the design information often resides only within the minds of senior designers. To foster reusability of design, there is a need to capture the information from senior designers and make it available in useful forms to others to take advantage of it. Biggerstaff et al. [5] argue that one important inhibiting

factor to capture design information for reuse is the lack of a representation for design that fosters reusability.

Biggerstaff et al. [5] suggest that a representation for design to foster reusability should be able to represent knowledge about implementation structures in factored forms. For example, the code of a component represents a variety of requirements and design decisions which may be difficult to separate in order to understand the component and modify it. Representing components in factored forms may ease reuse of some aspects of it (e.g., design structure) without having to extract that information from the code. The need for multiple views in software designs has been advocated by others [6], [11].

Also, Biggerstaff et al. suggest that the representation for design to foster reusability should be able to create partial specifications of design information that can be incrementally extended. This property arises because most often software designs are created incrementally. Often, the design process involves discovering the major parts of the architecture of a system. Over time, details are added which may cause the reorganization of the major parts. The partial architectures are highly reusable, whereas, the details are typically not. Including too much details may impair the reusability of partial architectures. A representation should allow the specification of partial architectures, as well as, permit incremental completion of details over time. The need for partial specifications of designs is also advocated by others [11], [53].

Other inhibiting factors to software reuse have to do with the *not-invented-here* syndrome, and corporation management policies [5].

OO techniques are often advocated to promote software reuse and ease of maintenance of software system [1], [34]. In the next section, we describe the characteristics of object-oriented programming as considered in this thesis and discuss how they promote software reuse.

2.1.1 The Nature of Object Oriented Programming

Object-oriented programming (OOP) is an approach of implementing software systems in which programs are organized as collections of cooperating objects. Each object is an instance of a class; a class is a generic specification for an arbitrary number of similar objects. Classes are organized in hierarchies; class hierarchies model the commonalities and the specializations of objects in the problem domain.

In the OOP community, analysts, designers and programmers tend to define the nature of objects in slightly different ways. As a design model, Booch [6] defines an object as follows: "An object has state, behaviour, and identity; the structure and behaviour of similar objects are defined in their common class". There are also some disputes in the OO community on the exact characteristics of OOP [41], [43]. OOP is often characterized by three concepts: *inheritance*, *polymorphism*, and *encapsulation*.

Inheritance is a relationship among classes, where one class shares the structure or behaviour defined in one or more other classes. A classical example of use of inheritance is the definition of graphical objects such as rectangles and squares. A square is a *kind-of* rectangle except that its height and length are equal. In this case, the square inherits its features from the rectangles but augments the definition of the rectangle by restricting the height and the length to an equal value.

Inheritance promotes code reuse by defining a *kind-of* hierarchy (or sometime referred as *is-a*) among classes in which a subclass inherits from one or more superclasses; a subclass typically augments or redefines the existing structure or behaviour of its superclasses. Class inheritance supports a style of programming called *programming-by-difference* where programmers define new classes by picking closely related classes and expressing the differences between the old and the new ones [34]. One important benefit of class inheritance is that it enables extensions to be made to a class while leaving the original code intact.

Polymorphism means the ability to take several forms [39]. In OOP, this refers to the ability of different kinds of objects to respond to the same message in a different way. A classic example of polymorphism is displaying different graphical shapes such as rectangles or ovals on a device. Every graphical shape responds to the same message for displaying itself on the device but does it in its own way (e.g., the work involved in displaying a rectangle is different than for an oval). Adding to the same example, suppose that a device driver implements a routine for displaying a collection of heterogeneous graphical shapes. The code of the routine simply notifies every element of the collection to display itself without knowing the actual type of the element. The selection of the appropriate piece of code depends on the type of graphic elements and occurs at run-time (*run-time binding*).

The concept of polymorphism promotes software reuse and ease of maintenance by encouraging the construction standard protocols among components and by reducing the name space of programs [34]. Defining standard protocols helps developers to define a common vocabulary throughout the application programs. For example, mathematicians

reuse the same names for arithmetic operations for various kinds of objects such as matrices, and complex numbers. Polymorphism enhances the power of defining standard protocols. For example, languages with no polymorphism at all, such as Pascal, disallow the use of the same name for different procedures (within the same scope). As a result, programs written in these languages often contain procedures with different but similar names for conceptually related operations (e.g., *MatrixPlus*, *ComplexPlus*). In languages supporting polymorphism, conceptually related operations may be assigned the same name in different kinds of objects. As a result, there is a smaller number of method names in the application which reduces the amount of information that a maintainer may have to learn.

The concept of run-time binding also promotes reusability. The run-time binding mechanism completes the separation of the act of requesting a service from that of deciding which method will be executed to satisfy it [1]. Conventional programming techniques usually require the client to perform both the act of requesting a service and to decide which method to perform to satisfy the request, thus unnecessarily spreading implementation knowledge among the components of a system [1].

Encapsulation is the grouping of parts [11]. Encapsulated parts may be arranged in a way that separate the ones that are externally visible from the ones that are not (information hiding). A classical example of encapsulation is the implementation of a stack. Clients use the stack through its visible interfaces such as *push* or *pop*. The internal structure of the stack is not known to clients and in theory, the stack's designer is free to use any type of data structures such as arrays or linked lists without interfering with the proper functioning of its clients (note that performance requirements may influence the internal structure).

In relation to OOP, encapsulation promotes software reuse by enabling the logical grouping of objects. When used in relation with the concept of information hiding, encapsulation may prevent interdependence between objects in a way that the implementation of an object can be changed without affecting objects that use its services. Encapsulation is not unique to OOPs, but the ability to combine data structure and behaviour in a single unit (object) makes encapsulation cleaner and more powerful than in conventional languages that separate data and behaviour [43].

2.2 The Nature of Real-Time Embedded Systems

A typical real-time embedded system reacts to events from the external environment and/or controls events to the external environment under timing constraints [33]. Real-time embedded systems often have stringent requirements for robustness and continuous operation [11]. Examples of real-time systems are telephone switching systems, avionic systems, and industrial process control systems.

Karam [36] characterizes a real-time embedded system as follows: it implements a single or special-purpose application (it is not a general purpose computer); there is a real or apparent concurrency; it exhibits a primarily reactive behaviour to external events; and it usually consists of rigidly constrained sub-systems and interfaces.

The characteristics of real-time systems make developing software for them one of the most difficult problem in the field of software engineering [1], [53]. In addition to the complexity inherent to a given application domain, real-time designers often need to deal with the complexity induced by real-time concerns. For example, the presence of concurrency may introduce potential problems such as deadlock, starvation and critical races that may arise during execution [36]. Also, real-time systems are often composed of a large number of components. These components often executed concurrently with one another to carry the intended behaviour of the system. The characteristics of real-time systems and a potentially large number of components make understanding the behaviour of the system as a whole very difficult.

One approach to understand the behaviour of a system is to consider every component in isolation. This approach is often difficult to apply in practice due to complex and subtle interaction patterns between components [53]. Another approach is to consider patterns of activities crossing the system in response to external stimuli [33], [9], [53]. These patterns of activities may involve the collaboration of a number of components in the system. One advantage of this approach is that it enables one to take a more global and intuitively cleaner approach to describe behaviour in real-time systems [53].

The difficulties in developing real-time embedded systems force system designers to place an increasing emphasis on software reuse [1]. Real-time systems represent an area where both source component reuse and design reuse are important. The characteristics of real-time systems arise the need for libraries of design-level abstractions to describe various patterns of organizing and interconnecting concurrent components [8].

2.3 Summary

This thesis is concerned with exploring ways of visualizing designs of object-oriented real-time programs. Given the importance of software reuse, this thesis is also concerned with exploring ways of enhancing software reuse in real-time systems.

Visualizing Designs of Object-Oriented Real-Time Programs

Visualization techniques may assist designers to reason about real-time systems [53]. Buhr [8] suggests a visual design notation called MachineCharts to depict real-time systems. The notation permits ones to draw design diagrams indicating the nature of components such as processes, modules, objects, and subsystems and the way these components interact. The notation conveys enough information that it is often possible for designers to mentally play behaviour patterns moving through components. In this thesis, we use the MachineCharts notation as a starting point to depict OORT programs.

Buhr [9], [15] proposes a visual notation based on the concept of timethread to express end-to-end behaviour in real-time systems. A timethread links activities performed by the system from some stimulus and terminating with an eventual response. Timethreads can be used to visually reason about time critical activities, and synchronization problems. In this thesis, we use timethreads to reason about behavioural patterns of activities in OORT programs.

Many visual design notations have been proposed to express designs of OO programs [6], [43], [48], [54]. Most notations model OO programs using the concept of objects and classes and tend to express the relationships between objects and classes from different views (often using various kinds of diagrams). Among the different views, many notations capture the relationships between classes of objects using models similar to or extensions of ER [18] diagrams. In this thesis, we adopt one ER-like notation to express the relationships between various classes of objects in OORT programs.

Enhancing Software Reuse in Real-Time Systems

In this thesis, we explore the use of object-oriented techniques to enhance software reuse in real-time systems. OO techniques are often advocated to promote software reuse but do not guarantee it, OO programs must be designed for reusability [34]. One important aspect of software reuse is the discovery of reusable abstractions.

Finding reusable abstractions is difficult, and in general, it seems that an abstraction is usually discovered by generalizing a number of concrete examples [34]. In the context of

real-time systems, the behaviour of a system may be expressed in terms of end-to-end behavioural patterns [53]. One may infer that generalizing concrete end-to-end behavioural patterns may help to discover reusable abstractions. In this thesis, we explore a *timethread-oriented* approach to express end-to-end behaviour in OORT programs to help discovering reusable abstractions in real-time systems.

2.4 Related Work

Jacobson [33] proposes an object-oriented design method for large real-time systems. The method is centered around the concept of *use cases*. A *use case* describes a sequence of transactions in a dialogue between a user and the system. A designer may specify the behaviour of a system using a set of concrete *use cases* which are expressed using structured English. Relationships between related *use cases* may be captured by a form of inheritance. *Use cases* drive the design by helping to discover components (*blocks*) and interfaces of components in the system. Jacobson depicts the behaviour of the system using *interaction diagrams* which illustrates the sequence of execution of *blocks*.

The concept of *use cases* relates to the concept of timethreads because it describes sequences of activities occurring in the system upon receiving an external stimulus. However, the depiction of *use cases* differs from timethreads because Jacobson's *interaction diagrams* imply control flow between components whereas timethreads express *cause-effect* flow. Jacobson's method is similar to our *timethread-oriented* approach because stimulus-response patterns are used to drive the design including the discovery of objects, as well as, interfaces of objects.

Booch [6] proposes an object-oriented design (OOD) method centered around the concept of objects. OOD uses a relatively large set of concepts and techniques to describe the design and implementation of OO programs. Among others, Booch introduces the concept of *mechanisms* to describe any structure whereby groups of related objects work together to provide some behaviour. An example of a *mechanism* relates the drawing mechanism commonly used in graphical user interfaces where several objects collaborate to present an image to a user. Booch depicts *mechanisms* using *object diagrams* in coordination with *timing diagrams* to describe the flow of messages among objects in response to stimuli from the environment (e.g., a mouse click event in graphical user interfaces).

Booch's *mechanisms* relate to timethreads because they attempt to describe higher level behaviour than the behaviour of individual objects. Like Jacobson's *interaction diagrams*,

the depiction of Booch's *mechanisms* differs from timethreads because Booch's *object diagrams* imply control flow whereas timethreads do not. Also, Booch's method differs from our *timethread-oriented* approach because Booch prioritizes the discovery of objects and subsequently adds behaviour to these objects whereas we use behavioural patterns to help identifying objects and subsequently partition the behaviour to individual objects.

Wirfs-Brock et al. [54] propose an object-oriented design method based on the concepts of *responsibilities*, *collaborations* and *contracts*. The *responsibilities* of an object are all of the services it provides in all the contracts it supports. *Contracts* represent the grouping of related responsibilities. *Collaborations* represent the set of requests from a client to a server in fulfillment of a client responsibility. The patterns of *collaborations* within an application reveal the flow of control and information in the system during execution. *Collaborations* patterns are depicted using *collaborations graphs* whose prime purpose is to help visualizing the behaviour of the system using walkthroughs of scenarios. Wirfs-Brock's method relates to our *timethread-oriented* approach because it stresses the importance of reasoning about patterns of activities in the design of object-oriented programs.

There exists a number of published design methods in the OO community supported by visual design notations [19], [43], [48]. Most of these methods emphasize the description of the behaviour of individual objects in the system. These methods relate to our work because they are based on the object-oriented concepts and that they emphasize the use visual techniques to describe designs of OO programs.

Chapter 3 Re-engineering a Non-OO Program

In this chapter, we explore small scale software reuse opportunities by objectifying various data elements in an existing real-time system while preserving the module and tasking structure of the system. The existing real time system is a program written in Ada-83 to simulate a *connection-oriented* network. We objectify: 1) the state information contained in components responsible for managing virtual circuits; and 2) components such as messages, packets, and frames responsible for conveying data in the simulated *connection-oriented* network. Objectifying various data elements drives the creation of class hierarchies to capture the specializations and commonalities of the various data objects. The benefits of objectifying the various data elements are: 1) we can exploit code reuse through class inheritance; and 2) we can ease future maintenance.

We progressively re-engineer the network simulator program by applying OO techniques on it and map the resulting design to Ada-9x [57]. The progressive modifications of the program create interesting combinations of elements of OO design and real-time design. We use visual techniques to describe the design of the original real-time program and its re-engineered version. Finally, we discuss the influence of applying OO techniques on the original program. A summary of the Ada-9x language may be found in Appendix A.1.

3.1 The Approach

OO techniques are often advocated to promote reusable software and ease the maintenance of software systems. One contributing factor to software reuse is the concept of class inheritance. Class inheritance allows subclasses to share operations and internal structure of ancestor classes.

We explore software reuse opportunities by objectifying various data elements in the network simulator program. We express the design of the original program using the MachineCharts notation and the Timethread notation. We use timethreads over conventional MachineCharts diagrams to express *end-to-end* behaviour of the network simulator program which helps us to understand its design and to identify candidate objects for reuse opportunities.

We analyze the design and the program text of the network simulator to discover potential class hierarchies. This analysis reveals that components responsible for managing virtual circuits (virtual circuit managers) have different roles in a virtual circuit and that these components maintain a state variable for distinguishing the different roles. We objectify the state information contained in virtual circuit managers while preserving their tasking structure (Section 3.2). We encapsulate the state information in a passive object and capture the commonalities and specializations of this object using class hierarchies. We treat a virtual circuit manager as an active shell encapsulating the passive object. The specialized behaviour (role) of a virtual circuit manager in a virtual circuit is achieved by dynamically creating an instance of the passive object inside the virtual circuit manager.

The concepts of polymorphism and run-time binding also contribute to software reuse and ease of maintenance of software systems [34]. Polymorphism reduces the name space of programs [34]. Run-time binding conceals implementation knowledge among components by separating the act of requesting a service from deciding which method to execute to satisfy the request [1]. We objectify messages, packets, and frames in the network simulator and create class hierarchies of them (Section 3.3). We present a partial implementation of a design alternative for transmitting message, packet and frame objects over communication links. We consider the case where nodes of the network simulator do not share the same address space. We incorporate the concepts of polymorphism and run-time binding in the design of services for translating message, packet and frame objects into a format suitable for transmission.

Combining the result of objectifying various data elements in the network simulator leads to a situation often referred as multiple polymorphism (Section 3.4). We observe, as others do [29], [17], that such situations arise in other OO programming examples. In these situations, conventional OO programming practice breaks down, leading to code that may not be properly modular [31]. We use a technique due to Ingalls [31] to implement the multiple polymorphism in a language supporting single polymorphism only and discuss its influence on the resulting program.

3.1.1 The Network Simulator

The network simulator [12] is a program written in Ada-83 to simulate a connection-oriented network based on the OSI¹ model. The program has been written at Carleton University to provide an Ada example² for teaching and research purposes. We use the work of Scott [45] as a starting point to express the design of the simulated network using the MachineCharts notation³ and also use timethreads to describe *end-to-end* behaviour.

Tanenbaum [51] describes the connection-oriented service by making an analogy to the telephone system. When users of the telephone system wishes to talk to someone, they pick up the phone and dial the number, talk to the callee and hang up the phone. Similarly, users of a connection-oriented service obey a three phases protocol; the first phase involves establishing a connection, the second phases involves using the connection, and the last phases involves destroying the connection.

Depicting the Design using the MachineCharts Notation

In the network simulator, each node has four layers, a Host, a Network, a Link, and a Physical layer. The program simulates bidirectional physical connections and uses an alternating bit protocol to transfer data between the nodes of the network. Figure 1 represents OSI layers as cooperating peer components. The figure shows one Host component interacting with one Network component, and many Link components interacting with the Network component. Every Link component interacts with a Physical component.

Figure 1 Network Simulator as Cooperating Peers

1. Open System Interconnection model proposed by the International Standards Organization (ISO).
2. The existing version is a tested and executable program.
3. Scott extracts the design of the simulated network using the MachineCharts notation. Scott also explores design alternatives for the network simulator in which layers of the communication system are viewed as instances of abstract data types and describes the resulting designs using the MachineCharts notation.

Figure 2 gives a detailed conceptual view of the network simulator using the MachineCharts notation. The figure shows a stack of cooperating peers with nested components. The four cooperating peers replicate the four lowest layers of the OSI model.

Often in communications networks, the Host layer encapsulates several user tasks which entertain a bidirectional transfer of messages with user tasks located in remote Host layers. However, in the network simulator, user tasks are replaced by two *actors*, one (the Source) for generating messages to remote nodes and the other one (the Sink) for sinking messages received from remote nodes. The simulator limits the transfer of messages between Source and Sink actors to a single direction. There is no bidirectional transfer of messages between actors at the Host layer.

The Network layer manages a limited number of virtual circuits which share the physical connections between the nodes. For users of the Network layer, virtual circuits create the illusion of multiple physical connections even though there really exists only one. Every virtual circuit requires a set of virtual circuit managers (VCMgr), one per node. VCMgrs maintain the virtual connection between the nodes and have one of three roles depending upon their relative location in the virtual circuit. A VCMgr can source, forward, or be the terminal point for messages. In a source node, a VCMgr takes care of requests for either establishing or closing a connection and sends data messages to its peer. In a destination node, a VCMgr acknowledges requests for establishing or closing a connection and receives data messages from its peer. In a forwarding node, a VCMgr forwards all messages.

The Network layer has another major active component, namely, the Dispatcher. The Dispatcher allocates local VCMgr to local or remote clients, and manages a list of available VCMgrs. The remaining active components of the Network layer are part of a *glue structure*⁴ used for buffering packets between the Network layer and the Link layer and for preventing unnecessary waiting by the VCMgrs.

The Link layer manages the transfer of data over the physical connection. The Link layer implements an alternating bit protocol in which every frame sent is expected to be acknowledged within a limited period of time. The Link actor has two *stores* for outgoing and incoming packets. Note that in the original implementation program, these stores are located outside the actor, we show them internally because no other components beside the Link actor access them to emphasize encapsulation. The Tick actor periodically notifies the

4. Glue structures provide the machinery to support the protocols by means of which peer actors may interact [8].

Link actor. The PktTransporter actor is part of the glue structure between the Link and the Network layers.

The Physical layer sends and receives frames. Components in the Physical layer simulate what is normally implemented in hardware and low level software components.

Figure 2 Network Simulator Overview Diagram

Visualizing End-to-End Behaviour using Timethreads

The following set of diagrams use timethreads over simplified MachineCharts diagrams to show the macroscopic sequences of activities occurring when establishing, using and destroying a connection from a source node through a forwarder to the destination node.

Figure 3 shows a timethread describing the sequences of activities for establishing a connection. The timethread starts from the Host layer on the left side of the diagram and terminates in the Dispatcher in the Network layer on the right side of the diagram. Assuming that resources were available, the Dispatcher allocates a virtual circuit to a client which then requests the virtual circuit manager to establish the connection with the remote node. The VCMgr accepts the request and launches another timethread representing the low level activities associated with establishing the connection. These low level activities involve formatting packets and frames and routing them to the destination node. The figure shows the case where the source node and the destination node are not connected directly; the request for establishing a connection is forwarded by a relay node. When the Dispatcher in the relay node gets the request, it allocates a VCMgr to the remote visitor and commands the VCMgr to establish a connection between itself and the next node in the direction of the destination node. When the destination node finally receives the request, it allocates a VCMgr and commands the VCMgr to acknowledge the establishment of the connection. Acknowledgement is then forwarded by every intermediate relay node until it gets back to the source node which then unblocks its client.

Figure 4 shows a timethread describing the sequence of activities occurring when using a connection⁵. Following the timethread, a message is sent from the host layer to the Network layer and forwarded until it gets to the destination node. In the destination node, the message is routed to the Host layer. Sending messages does not involve the Dispatcher.

Figure 5 shows the sequences of activities involved in destroying a connection. Following the timethread, the source node makes a request to destroy the connection which is then forwarded by the relay node until it gets to the destination node. In the destination node, the VCMgr acknowledges the request and reports its availability to the Dispatcher (black timethread). The VCMgrs in the source node and in the relay nodes forward the acknowledgement and report their availability to their Dispatchers.

5. Note that in the simulated connection-oriented network, high level messages between hosts are not acknowledged.

Figure 3 Establishing a Connection

Figure 4 Using a Connection

Figure 5 Destroying a Connection

Visualizing the Data Link Protocol using Timethreads

Figure 6 describes the protocol for transmitting packets between nodes in the network simulator using timethreads. The Link layer transmits one packet at a time using an alternating bit protocol. The figure shows three cooperating peers corresponding to the three lowest layers of the simulator program.

Following the timethread annotated as *Put Packet* starting from the Network layer on the left side of the diagram, a request to transmit a *data* packet is made to the Link layer. In the Link layer, the Link actor puts the packet on an outgoing buffer list (TxBuffer). The timethread proceeds through the TxBuffer to illustrate that the Link actor gets a packet from the TxBuffer and places the packet in a frame before requesting the Physical layer to transmit the frame to a remote node. The timethread comes back from the Physical layer and arrives at a *waiting place*. The waiting place corresponds to the Link actor waiting for the reception of an acknowledgement or a time-out event. The figure shows the cause-effect relationships between the waiting place and the triggering events. If a time-out event occurs first, the Link actor retransmits the packet if the maximum number of retries is not exceeded; otherwise, it removes the packet from the TxBuffer and returns the packet to the Network layer. If an acknowledgement event arrives first, the Link actor removes the transmitted packet from the TxBuffer and transmits the next outstanding packet. If there is no outstanding message, the Link actor simply waits for the next packet to transmit as depicted by the timethread end symbol. The Tick actor in the Link layer periodically notifies the Link actor at fixed time intervals. The Link actor uses the periodic notifications for managing the time-out period for the transmitted packet. Textual annotations in italics beside the threads guide the reader about activities occurring along the timethreads. We use arrows on timethreads to prevent ambiguous interpretations.

At the bottom of the diagram, from left to right, a timethread illustrates the transmission of a frame to the remote node. The timethread shows that when the remote node receives a frame, the machinery in the Physical Layer routes the frame in the Link layer. In the case of *data* packets, the Link actor extracts the *data* packet from the frame, stores it in an incoming buffer list (RxBuffer), and puts an acknowledgement packet in the TxBuffer. Some time later, the acknowledgement packet is transmitted to the remote node. The timethread annotated as *Route Packet* corresponds to the Link actor opening its guard which allows the PktTransporter actor in the Link layer to obtain a packet from the Link actor and route it to the Network layer.

Figure 6 Sending Packets Protocol

3.2 Objectifying Virtual Circuit Managers

We view virtual circuit managers as playing three distinct roles in the network simulator. We observe that the original Ada-83 code incorporates code to differentiate between the different roles of the VCMgrs. With procedural languages, programmers often need to include code in application programs to reflect the commonalities and specializations of the problem. Object-orientation offers a solution by supporting a way to capture those commonalities and specializations. In the case of the VCMgrs, one approach is to create a class to capture their commonalities and three subclasses to capture their specializations based on their roles in a virtual circuit.

In the original simulation program, a VCMgr plays one role in a virtual circuit for the duration of a connection. When the connection terminates, the VCMgr becomes available and may be subsequently allocated to play the same or a different role in a new connection. The transitory nature of role playing suggests a model where VCMgrs of different types could be dynamically created and destroyed in response to requests from clients. However, dynamically creating and destroying active components may put a demand on the system which may not be acceptable for performance reasons.

Another approach is to treat a VCMgr as an active shell encapsulating a passive object. A different instance of the passive object is dynamically created depending upon the role of the VCMgr in the virtual circuit. The instance resides in the VCMgr for the duration of the connection. The total number of VCMgrs is still fixed and all VCMgrs are created at start up time as in the original simulator program. The demand on the system for creating passive objects is generally lower than for active ones.

Figure 7 shows an overview of the approach. OSI layers are represented as peer components. The VCMgr component has a dynamically installed object called a virtual circuit node (VCNode). A dynamically established communication path attaches the VCMgr actor and its nested object (VCNode) to show that they are bound at run-time (run-time binding).

Figure 7 Virtual Circuit Nodes as Passive Objects Inside VCMgrs

In the original simulator program, three nested procedures in the VCMgr are executed upon receiving requests: to establish a connection from a local client (LocalConnect); to establish a connection from a remote node (RemoteConnect); or to forward messages or requests (ProcessPkt) from a remote node. Figure 8 shows part of the Network layer of the original Ada-83 program. Different timethreads show the three scenarios.

Figure 8 Original Ada-83 Program - VCMgrs with Nested Procedures

The original Ada-83 program uses a state variable (*MyRole*) to distinguish the different roles of VCMgrs. The manipulation and the testing of this state variable is scattered throughout the code. Figure 9 shows a part of the body of the original VCMgr task. The figure shows a type definition for the different roles, and highlights the lines of code (italic) where manipulations of the state variable occur in the code for the *RemoteConnect* and the *LocalConnect* procedures. For explanatory purposes and to illustrate the differences between the original program and its OO version, we replicate fragments of program text from the original program to subclasses refining the VCMgrs. Arrows on Figure 9 (and on Figure 10) show fragments of program text to be replicated.

Figure 9 Ada-83 Program - RemoteConnect and LocalConnect Procedures

Figure 10 shows parts of the code of the original processPkt procedure. The figure shows in bold characters the lines of code where explicit testing is made for distinguishing the particular roles of the VCMgr (testing the state of *MyRole*). In our approach, we capture the commonalities and specializations of the VCMgrs through subclassing, thus, removing the need for the state variable.

Figure 10 Ada-83 Code for Processing Packets

Figure 11 shows a part of the Network layer. The figure shows a VCMgr with a nested object called VCNode. We use the creation arrow to illustrate that an object is dynamically created inside the VCMgr. We use the *template* icon beside the creation arrow to show that a different type of instance is created depending upon the role of the VCMgr (Source, Forward, Terminal). The nested object (VCNode) has three methods. Every method is specialized in a subclass. When a VCMgr receives a request, it invokes the appropriate method of the VCNode object. Run-time dispatch automatically selects the appropriate piece of code depending upon the type of the VCNode object.

Figure 11 Virtual Circuit Node Objects Created Inside VCMgrs

Figure 12 shows the class hierarchy for virtual circuit nodes (VCNode). An abstract class called AbstractVCNodeClass captures the commonalities of the VCNodes, and the three subclasses capture their specializations.

Figure 12 Class Hierarchy for Virtual Circuit Nodes

Figure 13 shows the Ada-9x version of the VCNodes. An abstract class defines a type (VCNode) plus three methods namely RemoteConnect, LocalConnect, and ProcessPkt which are the operations supported by the type. The type is extended for the three roles of

VCNodes. The remaining part of the figure shows the subclass definitions. Note that the code in the figure has been simplified to give an overview. Ellipses means that more parameters are present but not shown.

Figure 13 Class Definitions for Virtual Circuit Nodes

Figure 14 shows the body for the virtual circuit node of type SourceVCNodeClass. The methods have been created by copying parts of the original procedures from the Ada-83 program. Arrows on Figure 14 point to code that has been copied from the original procedures with the same shading as shown by Figure 9 and Figure 10.

```

package body SourceVCNodeClass is
  procedure RemoteConnect(aVC : in out SourceVCNode; ...) is
  begin
    -- Not Possible, We could raise an exception1.
  end RemoteConnect;

  procedure LocalConnect(aVC : in out SourceVCNode; ...) is
    PTemp: Packet;
    NextNode.NodeLink: NetElem;
  begin
    MyStatus := WaitCreateAck;
    PTemp := (CreateVC, MyVCN, NetId, MyName, ...);
    -- Find Route and put packet in Tx buffer
  end LocalConnect;

  procedure ProcessPkt(aVC: SourceVCNode; P: in Packet) is
    PTemp: Packet;
    NextNode.NodeLink: NetElem;
  begin
    if P.PType = NakCreate then
      LocalConnectStatus := false;
      HostWait := false;
      MyStatus := Idle;
    elsif P.PType = AckCreate then
      RightNode.VCN := P.SendVCN;
      LocalConnectStatus := true;
      HostWait := false;
      MyStatus := Idle;
    elsif P.PType = Data then
      -- Not Possible here
    elsif P.PType = DestroyVC then
      -- Not Possible
    elsif P.PType = AckDestroy then
      MyStatus := Idle;
      NetId.Dispatcher.VCReady(MyVCN);
    end if;
  end ProcessPkt;
end SourceVCNodeClass;

```

From original Ada-83

From original Ada-83
ProcessPkt procedure

From original Ada-83
ProcessPkt procedure

From original Ada-83
ProcessPkt procedure

1. Sometimes abnormal situations happen during the normal operation of a program. The Ada Language provides a mechanism to deal with exceptions. For example, a request to establish a remote connection should never happen to a VCMgr playing the Source role. Here, the RemoteConnect method could raise an exception.

Figure 14 Body of the SourceVCNodeClass

Note that in the Ada-9x version, there is no need to distinguish or test for the different roles of the VCNodes. The specialization of VCNodes has been captured in the subclasses. However, the code of the VCNode objects still includes explicit testing for the different types of packets. Figure 14 shows the body of the code for processPkt method. The text in bold characters emphasizes the lines of code where explicit testing is done for packet types. In the next section, in the same way that we did with virtual circuit managers, we capture the commonalities and specializations of packets through subclassing.

3.3 Objectifying Messages, Packets, and Frames

In this section, we capture the relationships between messages, packets and frames through subclassing. We treat messages, packets, and frames as objects. We consider the case where nodes of the network simulator do not share the same address space. We illustrate one design alternative for transmitting message, packet and frame objects between nodes of the network simulator. The design alternative involves translating message, packet, and frame objects in ASCII strings. We first illustrate the translation of objects to ASCII strings and map the design diagrams to code fragments. We also illustrate the reconstruction of message, packet and frame objects from ASCII strings in detail.

Figure 15 shows an overview of this approach. OSI layers are represented as cooperating peers with references to message, packet, and frame objects flowing between them. The figure also shows ASCII strings (FrameString) moving between nodes of the network simulator.

Figure 15 Message, Packet, and Frame Objects Flowing Between Layers

Figure 16 shows a set of class hierarchies for messages, packets and frames and the relationships among them. Different relationships between entities are possible depending upon the analysis model. In our approach, we base the relationships among entities on the idea that in most communications systems, during transmission of the data, frames envelop packets, and packets envelop messages. This suggests defining the relationships among frames, packets, and frames as follows: a message is *part-of* a data packet, and a packet is *part of* a frame.

Figure 16 Class Hierarchy for Messages, Packets, and Frames

We consider the case where nodes of the network do not share the same address space; thus, nodes can not directly exchange and operate on objects obtained from other nodes. We translate message, packet, and frame objects in a format suitable for transmission over communication links. In our approach, on the transmit side, we translate message, packet, and frame objects to ASCII strings; on the receive side, we create message, packet, and frame objects from ASCII strings. ASCII strings encode enough information to reconstruct message, packet, and frame objects.

Figure 17 shows the class definitions for packet classes in Ada-9x. We encapsulate the different type definitions for packets into a package called PktClass. In the PktClass package, we define an enumerated type called PktType which is used for the reconstruction of packets from packet strings. The abstract class called AbstractPktClass defines a set of procedures (methods) and fields common to all packets. The AsString procedure converts a packet object to an ASCII string. The Parse procedure parses an ASCII string and initializes the internals of a packet object. The CreatePacketFromStr procedure is a class wide method and creates a packet from a packet string; the type of the packet depends on the information encoded in the packet string.

```

package PktClass is
  PKTTYPEOFFSET : constant := ...; -- Define constants for parsing the strings
  VCNOFFSET : constant := ...;
  SENDEROFFSET : constant := ...;
  type PktType is (Data,CreateVC,DestroyVC,AckCreate,AckDestroy,NakCreate);
  package AbstractPktClass is
    type Pkt is tagged private;
    type PktPtr is access Pkt;
    procedure AsString(aPkt: in Pkt; aString: out Text);
    procedure Parse(aPkt: in out Pkt; aString: in Text);
    procedure CreatePacketFromStr(aString: in Text; aPkt: out Pkt'Class);
  private
    type Pkt is tagged
      record
        VCN: integer;
        SenderNode: NetElem;
        PType : PacketType;
      end record;
  end AbstractPktClass;

  package DataPktClass is
    type DataPkt is new AbstractPktClass.Pkt with private;
    type DataPktPtr is access DataPkt;
    procedure AsString(aPkt: in DataPkt; aString: out Text);
    procedure Parse(aPkt: in out DataPkt; aString : Text);
  private
    type DataPkt is new AbstractPktClass.Pkt with
      record
        Message: MessagePtr;
        MessageStr: Text;
      end record;
  end DataPktClass;

  package CreateVCPktClass is
    type CreateVCPkt is new AbstractPktClass.Pkt with private;
    type CreateVCPktPtr is access CreateVCPkt;
    procedure AsString(aPkt: in CreateVCPkt; aString: out Text);
    procedure Parse(aPkt: in out CreateVCPkt; aString: in Text);
  private
    type CreateVCPkt is new AbstractPktClass.Pkt with
      record
        OriginateNode: NetElem;
        ReceiverNode: NetElem;
      end record;
  end CreateVCPktClass;
end PktClass;

```

Figure 17 Class Definitions For Packet Classes

Figure 18 gives an overview of the sequences of activities occurring when transmitting and receiving message objects. We assume that a connection has been established with a remote node. The figure illustrates the creation, the manipulation and the movement of message, packet, and frame objects in the system. The timethread annotated as *Send Message* on the right side shows the sequence of activities occurring when sending messages to remote nodes. The timethread describes the creation and the initialization of message, packet, and frame objects. In the Physical layer, frame objects are translated to ASCII strings which are subsequently transmitted to the remote node using the UART.

Figure 18 Messages, Packets, and Frames as Objects

Figure 19 shows a detailed view of the translation of a frame object in an ASCII string. The figure shows the case of a frame object containing a data packet (reference). The figure illustrates a common effect of using OO techniques where algorithms become fragmented among related objects. In this example, parts of the work for translating frame objects to ASCII strings are fragmented between frame, packet, and message objects. The algorithm becomes fragmented because specific objects best know how to apply the algorithm on themselves. Atkinson [1] pointed out that organizing software in terms of objects is more than just an elegant techniques for encouraging modularity and abstraction. It also tends to decentralize knowledge and data in a system, thus, confining implementation details to those parts that really need to access to them.

In our example, data packet objects encapsulate message objects whereas control packet do not. The particularities (implementation details) of the translation of packets to ASCII string is hidden in the respective `AsString` methods of the different types of packets. Specifically, the `AsString` method for data packet objects uses the services of its nested message objects, whereas the `AsString` method for control packet objects does not.

Figure 19 Transforming a Frame to an ASCII String (Data Packets)

Figure 20 shows the body of the `AsString` method for the `DataPktClass`. Translating a packet into an ASCII string shows an example of another effect of using OO techniques where the algorithm becomes fragmented in the class hierarchy. Here, the `AsString` method uses the services of its ancestor (`AbstractPktClass`) but translates the specific information concerning itself (its type and local fields). Figure 21 shows the body of the `AsString` method for the `AbstractPktClass` which translates the fields common to all packets. We assume the existence of system services for manipulating strings such as concatenating them.

```
package body DataPktClass is
  procedure AsString(aPkt: in out CreatePkt; aString: out Text) is
    parentStr, msgStr : Text;
  begin
    AsString(Pkt(aPkt), parentStr);
    CONVERTTOSTRING(aPkt.PType, aString, PKTTYPEOFFSET);
    -- Convert message (message is a reference to a message object)
    AsString(aPkt.Message, msgStr);
    CONCATENATE(parentStr, msgStr, aString);
  end AsString;
  ...
end DataPktClass;
```

Figure 20 Package Body for the `DataPktClass` - `AsString` Method

The translation of frames to ASCII strings is relatively simple. However, recreating messages, packets, and frames from ASCII strings is slightly more complicated. Figure 18

shows a timethread on the left side describing the sequence of activities occurring when receiving a frame string from a remote node. Frame objects are recreated from encoded frame strings. After creation, the Rx actor in the Physical layer invokes the parse method of the frame object passing a string. The parse method initializes the local fields and extracts the portion of the string corresponding to packet and message objects. A similar creation and initialization process repeats for packet and message objects. We annotate the creation arrow with a data flow icon to show that the creation process uses the information contained within a string.

Figure 21 shows the body of the AbstractPktClass. The figure shows a class wide method called CreatePacketFromStr which creates a packet of a specific type using the information encoded in the packet string. Note that hard-coded values are assumed to be defined as offsets for the parsing of the strings.

```

package body AbstractPktClass is
  procedure AsString(aPkt: in Pkt; aString: out Text) is
  begin
    CONVERTTOSTRING(aPkt.VCN,aString,VCNOFFSET);
    CONVERTTOSTRING(aPkt.SenderNode,aString,SENDEROFFSET);
  end AsString;

  procedure CreatePacketFromStr(aString: in Text; aPkt: out PktPtr) is
    pTemp : PktPtr;
    pType : PktType;
  begin
    CONVERTFROMSTRING(SUBSTRING(aString,PkTTYPEOFFSET,...),pType);
    case pType is
      when Data =>
        pTemp := new DataPkt;
      when CreateVC =>
        pTemp := new CreateVCPkt;
      when AckCreate =>
        ...
    end case;
    aPkt := pTemp;
  end CreatePacketFromStr;

  procedure Parse(aPkt: in out PktPtr; aString: in Text) is
  begin
    CONVERTFROMSTRING(SUBSTRING(aString,VCNOFFSET,...),VCN);
    CONVERTFROMSTRING(SUBSTRING(aString,SENDEROFFSET,...),SenderNode);
    ...
  end Parse;
end PktClass;

```

Figure 21 Package Body for Abstract Packet Class

Figure 20 shows the body of the Parse method for the DataPktClass. Similar to translation of packets to strings, the parsing of strings is fragmented between the ancestor and descendants. The MessageStr field contains a substring corresponding to message objects. The MessageStr field is used when creating message objects in the Network layer.

```

package body DataPktClass is
  ..
  procedure Parse(aPkt: in out DataPktPtr; aString: in Text) is
  begin
    -- Parse parent fields
    Parse(PktPtr(aPkt), aString);
    -- Parse local fields, extract host message from incoming substring
    SUBSTRING(aString.MSG_OFFSET..... MessageStr);
  ..
  end Parse;
end DataPktClass;

```

Figure 22 Package Body for the DataPktClass - Parse Method

For simple cases such as above, the code remains relatively easy to understand. Here, fragmenting and distributing the algorithm provide an elegant way to minimize and reuse the code. However, for complex cases, the fragmentation and the distribution of algorithms may make the understanding of the code difficult because control flow may bounce up and down the class hierarchy [50]. In our example, we may imagine the AsString method of the AbstractPktClass calling a local method which has been refined by its descendants. When invoking the local method, run-time dispatch would occur, thus, forcing control flow to go down the class hierarchy and possibly up again if the local method invokes its ancestor. Taenzer [50] relates these cases to the yoyo problem, and suggests a way to reduce the complexity of the message control trees by using methods of reasonable sizes. However, Taenzer mentions that there is a trade off between the size of the methods and the complexity of the message control trees; if methods are small, there tends to be more of them to carry out the needed behaviour, resulting in larger and more complex message control trees; if methods are large, there may be less of them but it may be harder to reuse the code, and descendants may have to replicate larger amounts of code to override the behaviour of their ancestors. The fragmentation and distribution of algorithms may also raise performance issues if the needed behaviour of a system is implemented using a large number of small methods that are fragmented in various class hierarchies and among related objects. As a result, a larger number of message dispatches may occur which may decrease the performance of the system.

Our approach for translating frame, packet, and message objects to ASCII strings and the reverse operation (recreating them from ASCII strings) is not symmetrical. The translation to strings is done in the lowest layer, whereas the recreation of messages, packets and frames is distributed among OSI layers. An alternative could be to recreate message, packet, and frame objects in the Physical layer similar to translating objects to strings. Figure 23 illustrates the alternative. This alternative may be more symmetrical in

relation to the translation of frame objects to strings but does not appear desirable because it may create an undesirable load on the Physical layer. This undesirable load is caused by parsing and creating packet and message objects in the Physical layer. Figure 23 explores a way to visualize such an alternative.

Figure 23 Recreating Messages, Packets, and Frames in the Physical Layer

3.4 Virtual Circuit Nodes, Messages, Packets, and Frames as Objects

In this section, we continue to progressively incorporate elements of OO design in an existing real-time design. We combine the two approaches described in the previous sections to capture the commonalities and the specializations of VCNodes, messages, packets and frames. Combining the two approaches creates a situation often referred as multiple polymorphism. We use a technique due to Ingalls [31] to implement the multiple polymorphic situation in a language supporting single polymorphism⁶ only and illustrate its effects on the code. To limit the number of diagrams, we concentrate on describing the interactions between packet objects and VCNode objects. The diagrams in this section could be used as patterns for describing similar interactions such as occurring between message objects and VCNode objects.

We reuse the same analysis model for frames, packets and messages in which during transmission of the data, frames envelop packets, and data packets envelop messages. We also reuse the approach of dynamically creating a passive object (VCNode) in the VCMgr actor to process packets and messages objects coming from the Link and the Host layers. Figure 24 shows an OMT class diagram representing relationships between message,

6. Most OOPs support single polymorphism only, perhaps the best known exception is the Common Lisp Object System (CLOS) [56].

packet, frame and VNode objects. The figure shows a relationship called “Process” between two class hierarchies namely the AbstractPktClass and the AbstractVNodeClass meaning that different kinds of VNode objects process different kinds of packet objects.

Figure 24 OMT Class diagram - Node Types Versus Packet Types

The processing of packet objects by VNode objects depends on both the type of packet objects and the type of VNode objects. For example, upon receiving a data packet, a VNode object of type Forward would relay the packet to the next node towards the destination node, whereas a VNode object of type Terminal would extract the message from the data packet and route the message to its Host layer. As another example, upon receiving a request to destroy the connection (VCDestroy), a VNode object of type Forward would relay the request, whereas, a VNode object of type Terminal would acknowledge the request and report itself available to its dispatcher. This creates a situation where conventional OO practice using languages supporting single polymorphism breaks down.

Conventional OO practice breaks because in most OOPs, a message is sent to a distinguished receiver object, and the run-time “type” of the receiver determines the appropriate method to invoke by the message; other arguments of the message are passed to the invoked method but do not participate in the message lookup [17]. In our case, the selection of the appropriate method (code for processing packet objects by VNode objects) depends on both the type of packet objects and the type of VNode objects (double polymorphism).

We apply Ingalls’s [31] technique to solve the multiple polymorphic situation. Ingalls’s technique uses a sequence of single polymorphic message dispatches to solve the multiple polymorphic situation. The technique involves defining two sets of methods which we refer

as the *processing* methods and the *selecting* methods. The *processing* methods contain the code related to every combination of objects involved in the multiple polymorphic situation (e.g., VCNode and packets). For example, in our case, VCNode objects (Source, Forward, and Terminal) define a method to process one kind of packet (DataPkt, AckCreatePacket, etc.) such as ProcessDataPkt or ProcessAckCreatePkt. The *selecting* methods are responsible for selecting one of the *processing* methods using a sequence of single polymorphic message dispatches.

Figure 25 provides an overview of our approach showing components in operation. OSI layers are represented as cooperating peers. We show references to message and packet objects flowing between layers. The VCNode object is dynamically created inside the VCMgr object. We use the roller icon⁷ to show that the objects (aMsg, and aPacket) are created elsewhere but dynamically installed in the VCMgr object (a reference is passed).

The figure illustrates a *selecting* method on the VCNode object which is responsible for initiating the resolution of the double polymorphic situation. This method forces message dispatch on packet objects by invoking another *selecting* method defined in the packet object. In turn, the *selecting* method of the packet object forces message dispatch on VCNode objects by calling one specific *processing* method of the VCNode object depending upon its own type.

Figure 25 VCNodes, Messages, Packets, and Frames as Objects

7. The Roller icon was inspired from the work of Buhr and Casselman [9].

Figure 26 illustrates the sequence of invocation of the *selecting* and *processing* methods in relation to time. The figure illustrates the sequence of invocation between an instance of the AckCreatePktClass (anAckCreatePkt) class and an instance of the SourceVCNodeClass (aSourceVCNode). We define a *selecting* method called RelayPkt in the packet object (anAckCreatePkt). Internally, the RelayPkt method invokes one specific *processing* method of the VCNode object depending upon the type of the packet object. For example, instances of the DataPktClass class invoke the ProcessData method whereas instances of the AckCreatePktClass class invoke the ProcessAckCreate method.

The resolution of the doubly polymorphic situation starts by invoking of the ProcessPkt method. The ProcessPkt method invokes the RelayPkt method of the packet object (anAckCreatePkt) forcing dispatch to occur on packet type. In turn, the RelayPkt method invokes the ProcessAckCreate method of the VCNode object forcing dispatch to occur on VCNode type. The ProcessAckCreate contains the code to process packet objects of type AckCreateVCPktClass by VCNode objects of type SourceVCNodeClass.

Figure 26 Ingalls's Technique to Resolve Doubly Polymorphic Situation

Figure 27 illustrates the interaction between an instance of the AckCreatePktClass (anAckCreatePkt) class and an instance of the SourceVCNodeClass (aSourceVCNode) in the context of the VCMgr. The figure shows a timethread describing a sequence of invocations that occur upon receiving a packet (anAckCreatePkt) from the Link layer. The sequence starts by the activation of the PktArrival gate of the VCMgr actor. The PktArrival gate invokes the ProcessPkt method of the nested VCNode object (aSourceVCNode) which starts the resolution of the doubly polymorphic situation.

Figure 27 A Packet Object Interacting with a Virtual Circuit Nodes Object

Figure 27 shows that VCNode objects and packet objects (InPkt) invoke one another methods. In strongly typed languages, this creates a circular dependency that needs to be resolved by the compiler. The problem is determining which of the two class definitions should be declared first because they both depend on one another. A work around the problem is to define intermediate classes to resolve the dependency in steps⁸.

Figure 28 shows a skeleton class definitions for VCNodes and packets. The figure shows an intermediate class definition called `AbstractBaseVCNodeClass` class. The `DataPktClass` class defines a method called `RelayPkt`. The `SourceVCNodeClass` class defines a set of methods for processing specific packet types such as `ProcessData` or `ProcessAckCreate`. We emphasize the lines of code related to type definitions for VCNode objects as italic characters and for packets as underlined characters to help the reader to follow the intermediate steps. The code has been simplified.

8. **Caveat:** At the time of writing, no Ada-9x compilers were available. We based the approach of using intermediate classes to resolve circular dependencies on experimentations done with another strongly typed object-oriented language namely the C++ language [38].


```

package AbstractBaseVCNodeClass is
  type VCNode is tagged private;
  private
    type VCNode is record
      -- All fields common to all Virtual Circuit Nodes
      myName : NetElem;
    end record;
  end AbstractBaseVCNodeClass;

with AbstractBaseVCNodeClass;
package PktClass is
  type PktType is (Data, CreateVC, DestroyVC, AckCreate, AckDestroy, NakCreate);
  package AbstractPktClass is
    type Packet is tagged private;
    type PacketPtr is access Packet;
    procedure RelayPkt (Packet, VCNode'CLASS);
  ...
  end AbstractPktClass;
  package DataPktClass is
    type DataPkt is new AbstractPktClass.Packet with private;
    type DataPktPtr is access DataPkt;
    procedure RelayPkt (DataPkt, VCNode'CLASS);
  ...
  end DataPktClass;
  -- Similar definitions for other types of packets
end PktClass;

with AbstractBaseVCNodeClass;
package AbstractVCNodeClass is
  procedure ProcessPkt (aVC : VCNode; P: in Packet'CLASS) is <>;
  procedure ProcessData (aVC : VCNode; P: in Packet'CLASS) is <>;
  procedure ProcessAckCreate (aVC : VCNode; P: in Packet'CLASS) is <>;
  -- Similar definitions for all other procedures related to packet processing
  ...
end AbstractVCNodeClass;

with AbstractVCNodeClass;
package SourceVCNodeClass is
  type SourceVCNode is new VCNode with private;
  procedure ProcessPkt (aVC : SourceVCNode; P: in Packet'CLASS) is <>;
  procedure ProcessData (aVC : SourceVCNode; P: in Packet'CLASS) is <>;
  procedure ProcessAckCreate (aVC : SourceVCNode; P: in Packet'CLASS) is <>;
  ...
  private
    type SourceVCNode is new VCNode with
      record
        -- All fields private to Source Virtual Circuit Node.
      end record;
  end SourceVCNodeClass;

```

Figure 28 Class Definitions for Virtual Circuit Nodes and Packets

Figure 29 shows part of the package body of the SourceVCNodeClass class. The figure shows a set of methods for processing packets which have been created by cutting parts of the ProcessPkt method shown by Figure 14. The figure also shows the ProcessPkt method which invokes the RelayPkt method of the incoming packet object.

```

package body SourceVCNodeClass is
  procedure ProcessNakCreate (aVC: SourceVirtualConnect; P: in Packet'CLASS) is
  begin
    -- Simply return a negative status to my host (waiting on WAIT)
    LocalConnectStatus := false;
    HostWait := false;
    MyStatus := Idle;
  end ProcessNakCreate;

  procedure ProcessAckCreate (aVC: SourceVirtualConnect; P: in Packet'CLASS) is
  begin
    RightNode.VCN := P.SendVCN;
    LocalConnectStatus := true;
    HostWait := false;
    MyStatus := Idle;
  end ProcessAckCreate;

  procedure ProcessData (aVC: SourceVirtualConnect; P: in Packet'CLASS) is
  begin
    null; -- Original Ada-83 does nothing for data packets in the case of Source nodes.
  end ProcessData;

  -- Other procedures related to packet processing.

  procedure ProcessPacket(aVC: SourceVirtualConnect; P: in Packet'CLASS) is
  begin
    RelayPkt (P.aVC); -- Invoke the relay method of the packet object.
  end ProcessPacket;
end SourceVCNodeClass;

```

Figure 29 Package Body for SourceVCNodeClass

Figure 30 shows the RelayPkt method for the AckCreatePktClass. The RelayPkt method invokes the ProcessAckCreate method. The underlined lines of code in Figure 29 and Figure 30 emphasize the methods activated along the timethread shown in Figure 27.

```

package body AckCreatePktClass is
  procedure RelayPkt (P: in DataPkt; aVC: VCNode'CLASS) is
  begin
    ProcessAckCreate(aVC, P); -- Invoke the Process packet method of the incoming VCNode object
  end RelayPkt;
end AckCreatePktClass;

```

Figure 30 Package Body for AckCreatePktClass

3.5 Summary

We explore small scale software reuse opportunities by objectifying various data elements in the network simulator program. We objectify: 1) the state information contained in components responsible for managing virtual circuits; and 2) components such as messages, packets, and frames responsible for conveying data in the simulated *connection-oriented* network. Objectifying various data elements drives the creation of class hierarchies to capture the specializations and commonalities of the various data objects which promotes code reuse through class inheritance.

We modify the network simulator program by applying OO techniques on it. The resulting code is significantly different than the original program. In procedural languages, procedures tend to be relatively large and often incorporate a significant amount of code to reflect the specializations of the problem. Using OO techniques, it is often possible to capture the commonalities and specializations of the problem using class inheritance. Generally, algorithms tend to be fragmented in class hierarchies and among related objects. Consequently, in OO programs, methods tend to be small. Figure 10, Figure 14, and Figure 29 show a progression of the fragmentation of the original Ada-83 procedure in many smaller methods. We discuss how the fragmentation of algorithms in OO programs may make code difficult to understand because of possibly complex message control trees [50].

Combining the result of objectifying various data elements in the network simulator program leads to a multiple polymorphic situation. Similar situations arise in other OO programming examples. For instance, displaying different kinds of graphical shapes such as rectangles and ovals on different kinds of devices such as printers and terminals⁹ leads to such a situation. We apply a technique due to Ingalls [31] to implement the multiple polymorphic situation in a language supporting single polymorphism only. Ingalls's technique preserves the benefits of good OO programming style at the expense of forcing the programmer to create and maintain *extra-dispatching* code. Tools may ease the creation and maintenance of this *extra-dispatching* code. Hebel et al. [29] describe a special browser for this purpose. We use the technique and visually describe the design as a result of applying Ingalls's technique to a system that combines elements of OO design and real-time design.

⁹ The code for displaying a rectangles on dot matrix printers is likely to be different than for terminals.

Chapter 4 Towards an Architectural Framework for Real-Time Systems

In this chapter, we illustrate a *timethread-oriented* approach to explore large scale reuse opportunities in a real-time system. Timethreads express *end-to-end* behaviour and global sequencing. A timethread links activities performed by the system. We analyze a timethread specification to identify: 1) *coordination objects* that embody the global sequencing and coordination information; and 2) *policy objects* that are responsible for performing activities. This analysis serves as the basis for designing classes from which these objects are constructed. Class hierarchies capture the commonalities (abstract) and the specializations (subclasses) of the *coordination objects* and the *policy objects*. The benefits of the *timethread-oriented* approach are: 1) we can reuse and refine the timethread sequencing information defined in an abstract coordination class; and 2) we can combine various kinds of policies for different system requirements.

We demonstrate the *timethread-oriented* approach by abstracting the design of three versions of a data link protocol in a small communication system example. We present the design of an abstract *coordination object* and one *policy object* in detail. We explore the feasibility of an architectural framework that could result from applying this approach and we provide a partial implementation of this architectural framework in the language DRAGOON. Finally we discuss the limitations of our approach.

4.1 The Approach

Johnson states, “One of the most important kinds of reuse is reuse of design” [34]. A collection of abstract classes can be used to express an abstract design. A collection of

abstract classes describing an abstract design is often referred as a *framework* in the OO community. Frameworks have been advocated to promote the reuse of algorithm and control code, as well as individual components. An example of a large and complex framework is the user interface framework of Smalltalk [23]. One characteristic of frameworks is that application specific methods defined by the user to adapt the framework for a special purpose are often called from within the framework rather than from the user's application code [34]. As a result, frameworks often coordinate and sequence activities of the application programs. Frameworks are often used for the design and implementation of standard user interfaces but are not limited to user interfaces. In this chapter, we investigate the creation of a small-scale framework for capturing the design of various data link protocols in a *point-to-point* communication system. The small-scale framework represents a step towards creating a reusable abstract design for communication systems.

Johnson states, "Finding new abstractions is difficult. In general, it seems that an abstraction is usually discovered by generalizing from a number of concrete examples" [34]. We explore the creation of reusable abstractions using timethreads (Subsection 4.2.1). We analyze a set of commonly used data link protocols by drawing timethreads for them. Drawing the set of timethreads helps to discover the commonalities and the specializations of the various protocols. Analyzing the commonalities and specializations permits us to generalize the behaviour of the various data link protocols in terms of abstract events and activities occurring along timethreads. This generalization serves as the basis for defining classes of *coordination objects* that embody the sequencing and coordination of activities along timethreads. We capture the commonalities and specializations of these coordination objects in a class hierarchy.

Timethread analysis also helps to discover decision points and generic activities along timethreads. We assign the responsibility to perform these generic activities to various *policy objects*. We explore the design of one *policy object* in detail. We explore the discovery of reusable abstractions by generalizing various framing policies in terms common interface protocols (Subsection 4.2.2). This generalization drives the creation of a class hierarchy for framing policies. We first express the framing policies using finite state machines (FSM). We encapsulate these FSMs in objects and analyze the required interfaces for these objects. We also explore ways of organizing the framing objects in their environment to improve flexibility and extensibility.

In the second section of this chapter, we partially implement the small-scale communication system in an interesting language called DRAGOON to illustrate with concrete examples some of the concepts discussed in the first section. DRAGOON is an

Ada-based OOP and incorporates OO concepts such as inheritance, polymorphism and dynamic binding. DRAGOON treats objects as units of reuse and distribution and offers an interesting set of features for dealing with concurrency and synchronization issues. A summary of the DRAGOON language may be found in Appendix A.2.

4.1.1 The Message Transfer Utility System

In this subsection, we describe the specification of the Message Transfer Utility (MTU) system. The MTU system was conceived by Buhr [13], [14] as a tutorial example for teaching purposes. We borrow the specifications and design alternatives from Buhr and use them as a starting point for exploring ways of promoting software reuse in the MTU system. The MTU is a small communication system supporting bidirectional transfer of ASCII files between two personal computers attached by a serial link. The MTU system implements a reliable transfer of data commonly referred as an *acknowledged connectionless* service [51]. We assume that an operator starts the execution of the program on both computers at about the same time and from then on, both computers are always ready to receive files from the remote computer. Figure 31 shows a high level view of the MTU system. The figure shows a bidirectional transfer of files between computers.

Figure 31 Files Transferred Between Nodes

The MTU software is organized around a stack of subsystems. Figure 32 illustrates the software organization (M, P, L) of the MTU system. The figure also illustrates the envelopment¹ of files at different subsystem levels during the transmission of files between computers. Two timethreads describe the logical movement of files between nodes (virtual) and the envelopment of files during transmission (physical).

Subsystem M is responsible for reading a set of ASCII files from the File System and for transmitting them using the services of *Subsystem P*. *Subsystem M* transmits the files one at a time to the remote computer. *Subsystem M* is also responsible for storing files received from the remote computer.

1. The approach of illustrating the envelopment of messages and packets was inspired from the work of Buhr [13].

Subsystem P provides a reliable transmission service to *Subsystem M*. *Subsystem P* hides the details of a raw transmission facility and transforms it into a link that appears free of transmission errors to *Subsystem M*. *Subsystem P* fragments messages into *data packets* and uses the services of *Subsystem L* for transmitting packets to the remote computer. *Subsystem P* is also responsible for rebuilding fragmented messages from data packets received from the remote computer before passing the message on to *Subsystem M*.

Subsystem L is responsible for enveloping packets into frames and monitoring low level activities during the transmission and the reception of frames between computers. Enveloping packets into frames involves delimiting packets with frame boundaries. *Subsystem L* insures a transparent transfer of packets (independently of their contents) to the remote computer. *Subsystem L* is also responsible for detecting frame boundaries and extracting packets from frames.

Figure 32 Virtual and Physical Timethreads - Enveloping Files

Generally, providing a reliable communication link between computers involves defining a common protocol for exchanging data between computers. In *acknowledged connectionless* service, each packet is generally acknowledged by the receiver, thus, permitting the sender to know if the data has arrived safely [51].

Communication protocols must take into account transmission errors of different kinds. For example, data packets and control packets may be partially corrupted by spurious link

problems or complete packets may be lost owing to temporary link failure. When packets are partially corrupted, the receiver may send a *negative acknowledgement* (nak) to indicate the sender the need for retransmission. When entire messages are lost (not delivered), the receiver has no chance to acknowledge them. To prevent the sender from waiting indefinitely for the acknowledgment, the sender generally maintains a timer. The timer is set with a timeout period based on the period of time that it normally takes for a packet to be transmitted and for its acknowledgment to be received. When the timer expires, the sender generally attempts to retransmit the packet. The timeout period and the number of times a packet is retransmitted may vary in function of the implementation of a system. Retransmissions of packets may cause the reception of duplicate packets which must be discarded by the receiver. Figure 33 shows various *end-to-end* behaviour patterns of the MTU system illustrating success and failure scenarios.

Figure 33 End-To-End Behaviour Patterns of the MTU System

Figure 34 shows a sequence of activities when transmitting a file in the MTU system. Text in italic characters guides the reader along the timethreads. The sequence starts from the *Subsystem M* in left side of the diagram where a file is read and a message is initialized with the content of the file. The message flows down to *Subsystem L* where it is fragmented into data packets. Data packets are transmitted to the remote node using the services of

Subsystem M. In its original design, the MTU sends one packet at a time and waits for an acknowledgment from the remote node (we investigate different transmission protocols for the MTU system). The MTU repeats this cycle until all packets have been transmitted. On the receive side, messages are reconstructed from data packets. If an unexpected packet arrives, the negative acknowledgment is sent which forces the retransmission of the expected packet. For simplicity of the design, we assume that the physical link does not go down indefinitely and that retransmissions of packets will ultimately succeed. Note that the existing MTU specifications define the interfaces of subsystems.

Figure 34 Sequences of Activities when Transmitting a File in the MTU

4.2 Increasing Software Reuse in the MTU System

We analyze the commonalities and the specializations of various Data Link protocols using timethreads. This analysis helps us to generalize activities and patterns of execution of these activities for the different Data Link protocols. We encapsulate the information related to the sequencing and coordination of these activities in an object (*coordination object*). We also identify different policies that are generally defined in Data Link protocols such as the number of retransmissions of a packet or the way packets are framed and treat

these policies as objects on their own (*policy objects*). Timethreads analysis also helps us to define some of the interfaces of objects.

Figure 35 illustrates our approach. The figure shows (left) timethreads describing a generic sequence of activities occurring when transmitting a file (refer to Figure 38 for details). We encapsulate segments (shaded area) of the timethreads related to the transmission protocol in an object (DataLinkProtocol). We also encapsulate smaller segments of timethreads related to various policies in different policy objects (e.g., Retries Policy, Timeout Policy, and Framing Policy). The DataLinkProtocol object is responsible for coordinating its nested policy objects and for sequencing the execution of generic activities along its associated timethread segments. The policy objects are responsible to carry out activities to achieve the policies defined in the Data Link protocols.

Figure 35 Objectifying the MTU System

In Subsection 4.2.1, we explore the creation of reusable abstractions using the *timethread-oriented* approach. The approach involves drawing timethreads to describe the behaviour of various data link protocols. In Subsection 4.2.2, we also explore the discovery of reusable abstractions by generalizing various framing policies in terms of common interface protocols. In both subsections, we explore ways of organizing objects to improve flexibility and extensibility.

4.2.1 Creating Reusable Abstractions by Analyzing Timethreads

Johnson states, "Creating new abstract classes is very important, but not easy. It is always easier to reuse a nicely packaged abstraction than to invent it". We generalize a set of data link protocol commonly used in *acknowledged connectionless* services. We analyze three data link protocols namely the *Send-And-Wait*, the *Sliding Window with Go-back* (with variances of receive window size) and the *Sliding Window with Selective Repeats* protocols [42], [51], [49]. This analysis serves as the basis for creating a class of *coordination objects* for data link protocols and various classes of policy objects.

Drawing Timethreads for the Data Link Protocols

Figure 36 and Figure 37 show timethreads for the *Send-And-Wait* protocol and the *Sliding Window with Go-Back* protocol. Diagrams showing the *Sliding window with Selective Repeats* and the *Sliding Window with Go-Back* (Receive Window size of one) protocols may be found in Appendix B.1 (refer Figure 74 and Figure 75). We annotate the timethreads with text to guide the reader through the sequences of activities occurring when transferring files in the MTU system for all data link protocols. The text in italic emphasizes the particular functions performed by each protocol. Note that we assume the existence of buffers to hold packets for transmission. For explanatory purposes, we illustrate the transmission of data packets and control packets between computers (bottom of the figures).

Drawing the timethreads helps us in the discovery of the commonalities and the specializations of the various protocols. For example, upon receiving an unexpected packet, the *Send-And-Wait* protocol sends a negative acknowledgment (nak) whereas the *Sliding Window with Go-back* does not. Shaded areas along the timethreads on Figure 36 and on Figure 37 illustrate specializations between the two protocols.

The Send-And-Wait Protocol

The *Send-And-Wait* is one of the simplest protocol and so it is widely used [42]. During normal operation, the transmitter sends a packet and waits for an acknowledgment before sending the next packet. As a result, the transmission of two successive packets is separated by a dead time. The protocol *Send-And-Wait* has the merits of simplicity and robustness but has the drawback of using the transmission channel inefficiently, particularly, for short packets exchanged on circuits that have long propagation delay [42]. On the positive side, the *Send-And-Wait* protocol has a low memory usage.

Figure 36 Send-And-Wait Protocol

The Sliding Window Based Protocol

The *Sliding Window* based protocols attempt to improve the performance by permitting the transmitter to send up to a predefined number of data packets before blocking, instead of just one. The sender anticipates the reception of a positive acknowledgment in the future. In general, the implementation of *Sliding Window* protocols include complex algorithms for managing packets and may require a significant amount of memory for buffering packets if the windows are large [42]. With *Sliding Window with Go-Back*, when the receiver detects the first unexpected packet, it waits for a transmission. Any subsequent

packets are stored in the Receive window (if within the bound). When the expected packet finally arrives, the receiver has a series of consecutive packets which may be processed. At that point, it sends an acknowledgment with a packet number corresponding to the upper end of the Receive window.

Figure 37 Sliding Window Go Back Protocol - Rx Window Greater Than One

Different variances of *Sliding Window* exist, these variances are mostly related to trade-offs between bandwidth and data link subsystem buffer space [51]. For example, upon receiving an error, the receiver using the *Sliding window with Go-Back* protocol does not send a negative acknowledgment but waits for the transmitter to timeout. In this case, the

transmitter retransmits all outstanding packets. In *Sliding Window with Selective Repeats*, when the receiver detects an error, it sends a negative acknowledgment with a packet number which indicates the transmitter to retransmit that particular packet only.

Generalizing the Data Link Protocols in Terms of Timethreads

Analyzing the commonalities and specializations permits us to generalize the behaviour of the various data link protocols in terms of abstract events and activities occurring along timethreads. Figure 38 shows timethreads representing a generalization from this analysis. The bold text beside the timethreads represents abstract events. Black rectangles along the timethreads represent activities. The processing performed during these activities may vary from one protocol to another. For example, upon receiving a retransmission request (**Retry**), some protocol may retransmit one packet whereas others (*Sliding Window with Go-back*) may retransmit many. We map these activities along timethreads to *deferred* methods in various abstract classes. These *deferred* methods are refined in subclasses.

We enforce the generality of classes of objects by forcing the execution of the *deferred* methods even if, in some cases, the refined version of the *deferred* methods would not do any processing. For example, when receiving an unexpected packet number, the *Send-And-Wait* protocol sends a negative acknowledgment whereas the *Sliding Window with Go-Back* does not (refer to Figure 36 and Figure 37). In the generalized timethread (refer to Figure 38), upon receiving an unexpected packet, we force the execution of a method called *ProcessUnexpectedPkt*. When refining the method, the *Send-And-Wait* protocol class would send a negative acknowledgment, whereas the *Sliding Window with Go-Back* protocol class would not do anything. Forcing the execution of *deferred* methods may ease the refinement of abstract classes because it gives the opportunity to subclasses to react to events that otherwise may have not been known at the subclass level.

We allocate segments of timethreads to various objects. The shaded areas on Figure 38 represent segments of timethreads allocated to objects. The larger shaded area represents the segments of timethreads allocated to the *DataLinkProtocol* object. The *DataLinkProtocol* object coordinates nested objects and the execution of various local methods. We also delegate smaller segments of the timethreads in policy objects. Some of these *policy objects* become instance variables of the *DataLinkProtocol* object.

Figure 38 Generalized Timethreads for Data Link Protocols

Reasoning about Interfaces of Objects using Timethreads

We use timethreads to reason about interfaces of objects. Timethreads facilitate this reasoning by: 1) helping to visualize potential interfaces at object boundaries; and 2) providing visual indications about the relationships between input and output interfaces. We analyze points of the timethreads crossing the boundary of objects (shaded areas on Figure 38). For incoming timethreads, we determine if the timethreads carry abstract events to be processed by the object. For outgoing timethreads, we determine if the timethreads carry abstract events generated by the object to be processed by its environment.

We map the processing of incoming abstract events to interface methods. We map the generation of outgoing abstract events to *output ports*². We simulate output ports using methods. The purpose of output ports is to de-couple the internal behaviour of an object from its environment by adding a level of indirection. Output ports may adapt protocols

2. Real-Time Object-Oriented Modelling (ROOM) [46] defines a model for developing real-time systems based on a structural element called an *actor*. An *actor* represents an active entity which may only interact with its environment through ports. Ports represent openings through which information (messages) may flow in or out of *actors*. Buhr and Casselman [9] suggest that such a model fosters design reuse by promoting component pluggability. Elliot et. al. [24] propose a similar model to promote the reusability of software components and demonstrate its feasibility in Ada.

between objects or re-route messages. Others suggest that constraining components to interact solely through ports (inbound and outbound interactions) promotes the reusability of components [9], [24].

Figure 39 illustrates a set of potential interfaces of the DataLinkProtocol object (small triangles at the boundary of the larger shaded area). The figure makes a visual distinction between incoming and outgoing interfaces. The same idea may be recursively applied to nested objects (e.g., Retry Policy object). To simplify the diagram, Figure 39 illustrates the idea by showing interfaces for the data link protocol object only.

Figure 39 Interfaces of Data Link Protocol Objects

Other criteria including process configurations may influence the layout of interfaces of objects. Defining a general rule for discovering interfaces using timethreads is outside the scope of our studies. In our specific example, we use timethreads to reason about interfaces of objects.

Capturing Decision Points along Timethreads

Figure 40 illustrates decision points along timethreads. The decision points occur at *or-forks* along timethreads. We capture decision points in *deferred* methods (refer to Figure 41) of the data link protocol object. These *deferred* methods may be used to refine the

control aspect of the data link protocol object. For example, determining if the transmission of a message is completed (**TxDone**) may involve testing the size of the Transmit window in window based protocols, whereas, it may involve testing the number of outstanding packets in the *Send-And-Wait* protocols.

Figure 40) Decision Points for the Data Link Protocol Object

Creating an Abstract Class for Data Link Protocols

The abstract class defines a set of *template* methods and a set of *deferred* methods. The set of *template* methods coordinate local objects (e.g., Retries policy object) and sequence the execution of *deferred* methods. The occurrence of events such as the arrival of a packet forces the execution of one *template* method. For example, receiving a positive acknowledgment results in executing one *template* method called *PutAckPacket*. In turn, the *template* method may use the services of local objects or invoke various *deferred* methods to provide the specific behaviour of a protocol. For example, upon receiving a transmission timeout event, some protocols may retransmit one packet whereas others may retransmit many. Figure 41 shows a class diagram for Data Link protocols. The abstract Data Link class includes a set of *template* and a set of *deferred* methods. It also includes a set of instance variables such as *aPacketizingPolicy*, or *aRetryPolicy* which represent the

policies that have been captured in objects, and a container object (*anOutstandingPktBuffer*) to hold packets for transmission.

Figure 41 Data Link Protocols Class Diagram

Figure 42 illustrates the class structure of the abstract Data Link Protocol class. The figure shows the wiring between *template* methods, *deferred* methods and nested objects. We draw *template* methods (thicker border) on the edge of the Data Link Protocol object to indicate that they represent the prime service points of the Data Link protocol objects. The *deferred* methods represent parts of the abstract data link protocol class that can be refined by its descendants using class inheritance. The figure also illustrates that the *policy* objects may be seen as pluggable components to indicate that different kinds of *policy* objects may be interchanged for different system requirements. Figure 42 also shows a set of methods (right side) that are used to simulate output ports. These methods encapsulate all outgoing invocations of the Data Link Protocol object to other objects. We attempt to improve reusability and flexibility by de-coupling the Data Link Protocol object from its environment. Other alternatives such as Protocol Adaptation may be used to further isolate an object from its environment. We investigate the Protocol Adaptation technique in Subsection 4.2.2.

Figure 42 Structure of the Data Link Protocol Abstract Class

Figure 43 illustrates the structure of the concrete subclass for the *Send-And-Wait* protocol. Two timethreads overlay the diagram. One timethread shows the activities occurring when a timeout event occurs (`TimeOut`), here, the *ProcessRetries* method is invoked which would retransmit the last packet transmitted (assumed to be kept in the top of the *anOutstandingPktBuffer* object). The other timethread shows the sequence of activities occurring when a request to transmit a message arrives from *Subsystem M* (`PutMsg`). In that case, the packetizing policy object (*aPacketizingPolicy*) fragments the message and invokes the *ProcessPutMsg* method. The *ProcessPutMsg* method would transmit the first packet of the packet buffer.

Figure 43 Structure of the *Send-And-Wait* protocol (Concrete) Class

The concrete subclass for the *Sliding Window with Go-back* protocol would add two nested objects representing the Transmit and the Receive windows (TxWindow and RxWindow in Figure 41) to Figure 43. Upon receiving a request to transmit a message, the *ProcessPutMsg* method would fill the Transmit window (TxWindow) and transmit the packets by invoking the *PutDown* method (presumably sending many instead of just one). Upon receiving a timeout event, the *ProcessRetries* method would scan the Transmit window and retransmit all packets in the Transmit window. The Transmit and Receive windows could also be defined in the abstract class. In that case, the size of the windows would be initialized to one for the *Send-And-Wait* protocol. Reorganizing the structure of classes to make them more general is likely to happen over time. Here, defining the Transmit and Receive windows in the abstract class may lead to a more general class structure.

Integrating the Data Link Protocol and Policy Objects in Processes

We have discussed the creation of abstract classes for a data link protocol object and various *policy objects* by analyzing timethreads. We express the commonalities and specializations of the various data link protocols in terms of *abstract events* and *activities* occurring along timethreads. We allocate segments of timethreads to a *coordination object* and various *policy objects* (refer to Figure 38). The occurrence of abstract events drive the execution on various methods of these objects. These objects need to be integrated in the context of processes in the MTU system. We discuss one possible integration of the data link protocol object and the timeout policy object in the context of one possible process allocation for the MTU system. We also discuss the limitations of our approach.

Figure 44 illustrates one possible process allocation for the MTU system. We cut the timethreads and allocate segments of them to various processes. The shaded areas (left) represent the segments of timethreads that are allocated to processes (right). More precisely, the shaded areas illustrate the thread of control of the processes (right). The scissors illustrate the places where the timethreads are cut. These cuts represent places where interactions between processes occur.

Figure 44 Allocation of Timethreads to Processes

We cut the timethreads in a way that enables us to encapsulate the data link protocol object in one process. Figure 45 shows the details of *Subsystem P*. *Subsystem P* encapsulates two actors (*Controller*, and *Tick*). The *Controller* actor encapsulates the data

link protocol object (*aDataLinkProtocol*). Similarly, the *Tick* actor encapsulates a timeout policy object (*aTimeoutPolicy*). The figure shows the wiring between the *Controller* actor and its nested object. Activating the gates of the *Controller* actor results in the invocation of the interface methods of the nested data link protocol object. In turn, interface methods of the data link protocol object invoke local methods including the one simulating output ports. We assume a mechanism to bind output ports of objects to their environment when the system starts up. For example, the *TxDone* output port of data link protocol object (*aDataLinkProtocol*) is bound to the control of the *Wait* gate of the *Controller* actor, whereas, the *SetTimer* output port is bound to one of the gate of the *Tick* actor. Two timethreads overlay the diagram to illustrate the interactions occurring between the *Controller* actor, its nested object (*aDataLinkProtocol*) and the *Tick* actor when transmitting a file. The sequence starts when receiving a request to transmit a file (*Put message*) from a client. The sequence finishes when the last acknowledgement packet for the transmitted message arrives which results in unblocking the client (*TxDone*).

Figure 45 Putting the Data Link Protocol Object in an Actor

We may reuse the data link protocol object in slightly different tasking organization (different glue structures) as long as the state information contained in the data link protocol object does not need to be distributed among processes. For example, a transporter could be put between the *Controller* actor and *Subsystem M* to prevent unnecessary waiting of the *Controller* actor when delivering messages to *Subsystem M*. The transporter would wait at a gate of the *Controller* actor to pick up messages. The *PutUp* output port of the data link protocol object would be controlling the guard condition of the gate at which the transporter is waiting for picking messages. Using output ports eases the reuse of the data link protocol object for different tasking organizations (glue structures) around the actor encapsulating

the data link protocol object. We demonstrate the reuse of the data link protocol object in a different tasking organization in section 4.3.1.

However, the reuse of the data link protocol object in the context of a process configuration where the state information contained in the data link protocol object is distributed among processes is limited. For example, in the MTU system, it may be desirable to separate the transmission and reception of files as independent activities throughout the different subsystems. In this case, *Subsystem P* may contain two actors, one actor being responsible for the transmission and the other one for reception of messages. A direct reuse of the data link protocol object may not be possible because parts of the state information of the data link protocol object may be needed by both processes.

An approach would be to encapsulate the data link protocol object in a third party (protected). The two actors in *Subsystem P* would access the needed state information of the data link protocol object by using the services of the third party. Figure 46 illustrates the approach. This approach appears unattractive because it may result in a complex design and hides interaction paths between actors (Tx and Rx) in the third party (which may result in hidden deadlock paths). Timethreads illustrate an equivalent sequence of interactions between various objects and actors when transmitting a message in relation to the previous approach (refer to Figure 45).

Figure 46 Encapsulating the Data Link Protocol in a Third Party

Another approach would be to separate the data link protocol objects in two distinct objects. One of the objects would encapsulate the state information concerning the transmission of packets, whereas, the other one would encapsulate the state information for the reception of packets. Both objects would interact to provide the same behaviour as the data link protocol object does. Figure 47 illustrates the approach. At the bottom, we show

different process allocations for *Subsystem P*. When the process allocation results in two actors in *Subsystem P*, the two objects would be encapsulated in the distinct actors (glue structures may be required between actors). When the process allocation results in a single actor in *Subsystem P*, both objects would reside in the single actor. The figure (top) also shows a possible set of interfaces for both data link protocol objects (RxProtocol, TxProtocol). New interfaces would be required to permit the two objects to interact. Class hierarchies could be created to capture the specializations of the various data link protocols. The figure shows timethreads to illustrate typical interaction sequences between the various objects and actors during reception of a message (last data packet received).

Figure 47 Separating the Data Link Protocol Object

Reusable abstractions are likely to evolve towards more general ones over time as attempts to reuse them in new contexts occur [34]. In the context of our example, attempting to reuse the data link protocol object in different process configurations results in reorganizing the reusable abstraction by separating it into smaller abstractions. Smaller abstractions are generally easier to reuse than larger ones [34]. Separating the data link protocol object in two distinct objects may increase the likelihood of software reuse.

The class hierarchies for the data link protocol object and the policy objects represent a step towards creating a small-scale framework for communication systems. A framework is an abstract design for a particular kind of application [34]. The generality of a framework depends on the concrete examples that the framework has been generalized from. Here, the framework has been abstracted from a relatively small number of representative examples (data link protocols). As such, attempting to extend this framework to support other data

link protocols may involve reorganizing the class hierarchies and the structure of the abstract classes of the framework. For example, a new data link protocol may affect the generic timethread specification (refer to Figure 38) such as adding new decision points to it. As a result, new deferred methods may be needed in the abstract class for data link protocol objects. Such a change would imply modifying the existing subclasses to comply with the abstract class definition.

This illustrates one aspect of frameworks in relation to their reusability. A mature framework's application specific behaviour is usually defined by refining methods of subclasses [34]. In these cases, the structure of the framework remains intact and the reuse payoffs are high. However, attempting to reuse a framework in a way that does not fit within the generality of the framework (as defined by its abstract classes) may involve reorganizing the framework. Here, reorganizing the structure of the framework decreases the reuse payoffs but may lead towards a more general framework which may be reusable through subclass refinements in the future.

4.2.2 Creating Reusable Abstractions by Analyzing Interface Protocols

In this subsection, we abstract different framing policy objects in terms of interface protocols. We abstract the commonalities and the specializations of various framing policies, namely: *Starting and Ending Characters with Character Stuffing*; *Character Counting*; and *Starting and Ending Flags with Bit Stuffing* [51]. We express the various framing policies in terms of FSMs. We encapsulate the FSMs in objects and generalize these objects in terms of common interface protocols. This generalization serves as the basis for creating a class hierarchy for framing policy objects. Creating class hierarchies promotes software reuse by reusing code (programmatic aspect of software reuse). We also explore OO techniques to increase the likelihood of software reuse. Applying these OO techniques promotes software reuse by easing the integration of objects in different environments (architectural aspect of software reuse).

In the MTU system, we assume that the framing of packets is done in the lowest subsystem (*Subsystem L*, refer Figure 35). Figure 48 illustrates a set of objects (top) that could be used in the MTU system. Two objects called the *Framer* and the *DeFramer* reside in the lowest subsystem of the MTU system. These objects provide services to other objects (presumably active ones) for transmitting and receiving packets by operating on outgoing (OutPkt) and incoming (InPkt) packets. At the bottom of the figures, we show two FSMs

to describe the *Starting and Ending Characters with Character Stuffing* framing policy³. The FSMs interact with the environment solely through methods defined by the framing policy objects (*aFramingPolicy*, or *aDeFramingPolicy*).

The *Starting and Ending Characters with Character Stuffing* policy frames packets on a character basis, thus, interfaces for reading or storing individual characters are likely to be needed. Other operations such as *reset* are mostly needed for recovering from errors or during the initial phase of a transmission. In our example, on the transmit side, the framing policy object (*aFramingPolicy*) defines three methods namely: *Get*, *reset*, and *read*. On the receive side, the object called *aDeFramingPolicy* defines *Put*, *store*, and *reset*.

Figure 48 Framing Policy - Starting and Ending Characters with Char. Stuffing

Other framing policies may require other interfaces. The *Character Counting* framing policy uses the size of the packet to delimit the end of the frame, in that case, an interface returning the size of the outgoing packet is likely to be needed. The *Bit Stuffing* framing policy operates on bits, thus, an interface providing a bit oriented view of a packet is likely to be needed. Figure 49 illustrates the result of abstracting the frame policies in terms of

3. The policy delimits frame boundaries by having each frame start with the ASCII characters DLE STX (Data Link Escape, Start of Text) and end with DLE ETX (End of TeXt) [51]. If the destination loses tracks of a frame boundaries, it may recover it by monitoring the starting or ending sequence of characters. The problem with this approach comes when transmitting binary data [51]. In which case, it may easily happen that the sequence of characters used for delimiting frames may appear in the data. To prevent the from happening, the sender stuffs an extra DLE character before any accidental DLE character in the data. The destination monitors two consecutive DLE characters and removes one of them before processing the data.

common interface protocols. Details about the *Character Counting* and the *Bit Stuffing* policies may be found in Appendix B (Refer to Figure 76 and Figure 77).

Figure 49 Abstracting Framing Policies

Figure 50 illustrates a technique used in OO programs which consists of inserting an intermediary object between cooperating objects. These intermediary objects are often called *Protocol Adaptors*. *Protocol Adaptors* aim at easing the reuse of components by adapting interface protocols. Figure 50 illustrates two examples of different protocol adaptors between a framing policy object and different kinds of packet objects. On the left side, the packet object (*OutPkt1*) may have an interface protocol resembling the one of a stack, thus, here, a read request is translated to a *Pop* operation. On the right side, the interface protocol of the packet object (*OutPkt2*) may have been designed following Meyer's guideline for methods that query the state of an object [39]. Meyer suggests that methods that query the state of an object (functions in Eiffel terms) should never change the state of the object⁴. In this case the figure shows that a read request translates to a *Get* operation (which only results the value of the character) followed by a *Next* operation (which moves to the next character).

Figure 50 Adding Protocol Adaptors Between Objects

Using protocol adaptors may increase the likelihood of reusing the framer object without changing the framing policy object (which may hide complex algorithms) to work with different kinds of packets (presumably designed following different guidelines by different people). However, using *protocol adaptors* adds levels of messaging between

4. Meyer argues that side-effect-producing functions conflict with the classical notion of a function in mathematics. A mathematical function will always give the same result when applied to the same argument; in contrast, functions with side effects may not [39].

cooperating objects which may be undesirable under certain time critical operations.

Using protocol adaptors has some similarities with using glue structures such as transporters and buffers to bind peer actors in the real-time domain. Glue structures provide the machinery to support the protocols by means of which peer actors may interact [8]. Glue structures may facilitate the reuse of actors in different tasking architectures by adapting the protocols (tasking interactions) of these actors to new environments. Furthermore, greater design reuse potential may be achieved by standardizing glue structures (canonical forms) [8]. Designers familiar with a good set of such structures may be able to apply them to new problems and may increase their chances to get the design right the first time compared to designers that are not.

Similarly, *protocol adaptors* facilitate software reuse by adapting the message protocol of objects. We suggest, as others do [20], [35], that classifying and documenting commonly used OO design patterns (*protocol adaptors* being one of them) may have a positive impact on the development of OO systems. Similarly to the glue structures in the real-time domain, designers familiar with these common OO design patterns could be able to apply them on new problems and may increase the likelihood to come up with a correct design. With a good collection of OO design patterns, designers could reuse design knowledge gained by others instead of having to discover it themselves.

4.3 Towards an Implementation of the MTU System

In this section, we explore a partial implementation of the MTU system in the DRAGOON language to illustrate with code some of the techniques described in section 4.2. We arbitrarily choose to implement the MTU system using the *Send and Wait* protocol and the *Starting and Ending Characters with Character Stuffing* framing policy.

Potential Objects in the MTU System

The previous subsections use behavioural patterns in the MTU system to help discover objects. As a result, they treat data link protocols and various policies as objects. Other objects may also be found in the MTU system by emphasizing data abstraction. For analysis purposes, Shlaer and Mellor [48] suggest various sources of potential objects such as tangible things, roles, events, and interactions. In the MTU system, devices such as the UART and data elements such as packets and messages seem to be natural candidates for objects. Booch [6] argues that real-world objects are not the only kinds of objects in a design, other important objects are the invention of the design process. In the MTU system, container objects such as temporary buffers to hold packets for transmission may also be

regarded as objects. In the same vein, class hierarchies may sometimes be created to ease the implementation of a program [27]. For example in the MTU system, subsystems may be treated as instances of classes belonging to a common class hierarchy. In this case, the ancestor class may be partial and may serve as a way to guarantee that subclasses obey a common protocol. Using behavioural patterns appears to be complementary to emphasizing data abstraction to discover objects in OORT programs. Figure 51 summarizes potential objects in the MTU system.

Figure 51 Potential Objects in the MTU System

Reasoning About Dynamic Storage Allocation with Timethreads

OO systems may be highly dynamic, with instances created and destroyed over the lifetime of the system [3], [6]. When no automatic garbage collection is available, dynamic storage allocation may be a common source of problems in OO implementations [21]. We use annotated timethreads to reason about the creation and deletion of objects in the system in time and place.

Figure 52 shows one possible scenario describing the dynamic allocation of packet and message objects in the MTU system. The timethread starts in *Subsystem M* in the upper left corner of the diagram. *Subsystem M* creates a message object (*OutMsg*) and passes it to *Subsystem P*. *Subsystem P* fragments the message object in packet objects (the cycling loop icon on the creation arrow indicates the creation of many instances) and initiates the transfer of packet objects using the services of *Subsystem L*. In our approach, *Subsystem L*

makes a copy of packet objects to transmit and adds framing characters as the packet is transmitted (transmit frames as strings). When a frame string is received by the remote computer, *Subsystem L* creates a packet object (*InPkt*) from the incoming string and passes the packet object to *Subsystem P*. Once *Subsystem P* has processed the incoming packet object, *Subsystem P* indirectly destroys it using a reference. When *Subsystem P* receives a positive acknowledgment, it destroys the packet that was just transmitted (assuming the top of the outstanding packet buffer) and transmits the next packet in the buffer. When all packet objects have been transmitted (*Tx done*), *Subsystem P* cleans up (destroying the last packet object in the outstanding packet buffer) and waits for the next message to transmit.

Figure 52 Annotated Timethreads Describing Dynamic Memory Allocation

We attempt to deal with the creation and destruction of objects from a global perspective because we see the memory as a global resource shared by many components of the system. Timethreads may depict *end-to-end* behaviour in a system. Annotating timethreads permits us to reason about dynamic memory allocation from a global perspective.

4.3.1 Mapping to DRAGOON

DRAGOON approaches synchronization with the philosophy of separating, as far as possible, synchronization concerns from functionality [26]. The functionality of a method is described in a sequential class (concrete) without concerns for synchronization. The required synchronization constraints (or the required behaviour) for a particular problem is then superimposed on the sequential method using a form of multiple inheritance called *behavioural inheritance*. DRAGOON deals with concurrency by defining *active objects*. An *active object* encapsulates an independent thread of control called a *thread*⁵. A *thread* may be explicitly defined in the class body or inherited from an ancestor.

Allocating Timethreads Segments to Active Objects

Figure 53 shows timethreads representing the transfer of files in the MTU system. Activities along timethreads (left side) are allocated to the same number of components (right side) as shown in Figure 44 except that here a *reactor* is used instead of an *actor* in *Subsystem P*. As such, there is no difference in the functions that are performed by each component illustrated by Figure 44 and Figure 53. The minor variations between the two cases come from practical concerns such as the number of physical processors and tradeoffs between overheads of process interactions versus exploiting potential concurrency.

Figure 53 Active Objects in the MTU System

⁵ In this chapter, the word "thread" refers to threads in DRAGOON.

The case shown here may be more suitable for a single processor system because it causes a smaller number of interactions between processes to occur during transmission and reception of files than the case shown by Figure 44 while permitting to model the transmission and reception of files as concurrent activities (apparent concurrency).

Detailed Design of Subsystem M

Figure 54 shows a detailed view of *Subsystem M*. *Subsystem M* has two nested objects namely the *FileSystem* object and the *Producer* object. The *Producer* object reads files from the file system and uses the services of *Subsystem P* to transmit the files one at a time to the remote computer. We depict an *active* object in DRAGOON as a *reactor*, which may provide protection or synchronization, encapsulating an actor (a *thread* in DRAGOON's terms). The *FileSystem* object provides protected services for reading and storing files. We define a sequential class for *FileSystem* objects, and use DRAGOON's behavioural inheritance to superimpose the behaviour of *Readers and Writers* (predefined behavioural class) on the sequential class. We depict the resulting *behavioured* objects as *reactors*⁶.

Figure 54 DRAGOON - MTU Subsystem M

6. We may also use waiting place trigger to express detailed behaviour of *behavioured* objects.

Figure 55 shows code fragments for defining a *behavioured* class for the *FileSystem* object. *Behavioural* classes specify synchronization conditions of a set of abstract operations using logic predicates. The superimposition of a *behavioural* class over a sequential class involves matching the methods of the sequential class with the abstract operations defined in the *behavioural* class. The figure illustrates superimposing the behaviour defined by a *behavioural* class on top of a sequential class using a form of multiple inheritance. Figure 72 in Appendix A.2 describes behavioural classes including the one for *Readers and Writers*.

Figure 55 File System Behavioured by Readers and Writers

Figure 56 shows code fragments for the *Producer* object located in *Subsystem M* (refer to Figure 54). The *Producer* object encapsulates a thread. Methods such as *SetFileSystem* or *SetDownPeer* are used during initialization of the MTU system to pass instance identifiers of other instances such as *Subsystem P*.

Figure 56 Subsystem M - Producer Object

Detailed Design of Subsystem P

Figure 57 shows a detailed view of *Subsystem P*. We reuse the data link protocol object from Subsection 4.2.1. We encapsulate the data link protocol object (*aDataLinkProtocol*) in a behaviour object (*Controller*). The *Controller* object protects the *DataLinkProtocol* object from concurrent accesses. The *Sequencer* object serves as a synchronization place (wait for transmission completed) for the *Producer* object from *Subsystem M*. The *Lick* object encapsulates a *TimeOut* policy object. At the bottom of the figure, two timethreads illustrate the sequence of activities occurring when a request to transmit a message occurs and when the last acknowledgment packet arrives.

2 OF 2

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

Figure 57 DRAGOON - MTU Subsystem P

Figure 58 shows code fragments of *Subsystem P*. The *SubsystemP* class defines a set of instance variables for local objects such as the *Tick* or the *Controller*. The *Subsystem P* class also maintains a set of instance variables to hold instance identifiers of other objects in the MTU system such as *Subsystem M* and *Subsystem L*. We assume methods to set these instance variables. The *PutUp* and the *PutDown* methods represent the prime service points of *Subsystem P*. These methods use the services of local objects and coordinate their execution.

Figure 58 DRAGOON - Partial Implementation of MTU Subsystem P

The code for the *Controller* object would be relatively simple. The *Controller* object would be defined using a sequential class. Most of the methods of the *Controller* would simply invoke methods of the *DataLinkProtocol* object (refer Figure 57). The behaviour of the *Controller* object would be defined by superimposing a behavioural class (MUTEX, refer Figure 72) on the *Controller* class.

Implementing the data link protocol object involves mapping the class diagrams for the *AbstractDataLinkProtocol* class defined in subsection 4.2.1 (refer to Figure 42) and its descendant the *Send-And-Wait* protocol class to code. Figure 59 shows code fragments for the *AbstractDataLinkProtocol* class. The abstract class defines a set of methods. Some of these methods coordinate the execution of *deferred* methods refined in the subclasses. For example, the *PutAckPacket* method invokes the *IsTxDone* method to determine if the transmission of a message is completed. Depending upon the result, the *PutAckPacket* method coordinates the execution of other methods such as *ProcessTxDone* or *ProcessTxNotDone*. The *AbstractDataLinkProtocol* class also defines methods to simulate output ports. For example, *PutDown* methods is responsible for routing messages to *Subsystem L*. The figure also shows that an implementation of the MTU system may use

existing class libraries. For example, the buffer for holding outstanding packet may be constructed by actualizing a generic queue class with the class defining *Packet* objects.

Figure 59 DRAGOON - Abstract Data Link Protocol Class

Figure 60 shows code fragments for the *Send-And-Wait* protocol class. The class completes the *deferred* methods defined in its ancestor to refine the behaviour of the ancestor. For example, the *IsTxDone* method indicates if the transmission is completed by verifying the size of the outstanding packet buffer. Some of these methods also use services of the ancestor. For example, *ProcessTxNotDone* method invokes the *PutDown* method of the ancestor to route packets to *Subsystem L*.

Figure 60 DRAGOON - Send and Wait Protocol Class

Detailed Design of Subsystem L

Figure 61 shows the internal structure of *Subsystem L*. Two active objects (*Receiver* and *Transmitter*) manage the transmission and the reception of packets. These active objects encapsulate other objects responsible for the synchronization of local *threads* and for buffering and framing packets. The objects responsible for framing packets (*Framer*, and *DeFramer*) encapsulate instances of the *Starting and Ending Characters with Character Stuffing* framing policy (*aFramingPolicy*, and *aDeFramingPolicy*). Two timethreads overlay the diagram to illustrate the transmission and the reception of packets.

In a nutshell, implementing *Subsystem L* would involve defining classes for active objects for the *Receiver* and the *Transmitter* objects. Implementing local components of these active objects such as *Sequencer* or *PktBuffer* objects would involve defining sequential classes for them and superimposing the behaviour of predefined behavioural classes on them (refer to Figure 72 for *Sequencer* and *BoundedBuffer* classes). The code for the framing policy objects would implement FSMs (refer to Figure 48).

Figure 61 DRAGOON - MTU Subsystem L

Putting the System Together

Sometimes, it may be appropriate to create class hierarchies to ease the implementation of a program. In the MTU system, we consider subsystems as instances of subclasses belonging to a common class hierarchy. We also create a class hierarchy for data elements such as messages and packets. Figure 62 shows two class hierarchies which are defined to

ease the implementation of the program. These class hierarchies permit to treat the different subsystems and the data elements flowing across them in a generic manner.

Figure 62 MTU Data and MTU Subsystem Class Hierarchy

Figure 63 shows code fragments for the class specifications for MTU subsystems. The *PutUp* method and *PutDown* method are subsystem interfaces in which instances of subclasses of the *MtuData* (*Packet* and *Messages*) flow.

Figure 63 DRAGOON - Class Specifications for MTU Subsystems

Figure 64 illustrates the initialization of the MTU system. Case 1 shows the creation phase. In DRAGOON, instances of classes must be explicitly created using the *Create* method. The *Create* method may be refined in subclasses for the specific needs of the different subsystems (M,P,L). The creation repeats recursively to nested components. Case 2 shows a phase where exchange of instance identifiers occur. In the MTU, every subsystem is a descendant of the abstract class (*MtuSubsystem*). The *SetUpPeer* and *SetDownPeer* take an identifier to an instance of the abstract class (or any of its descendants). During the operational phase of the MTU system, the subsystems interact with one another using these instance identifiers. Run-time binding makes the automatic selection of the methods (*PutDown*, *PutUp*) depending upon the type of the subsystem instances. The exchange of identifiers applies to nested components. Case 3 shows the

starting phase. In DRAGOON, active objects must be explicitly started using the Start method.

Figure 64 Initialization of the MTU System

Figure 65 illustrates the code for the MTU system. The figure shows fragments of code for creating the subsystems (instances of classes), exchanging instance identifiers and starting the operation of the subsystems.

```

class body MtuSystem is
  subsystemM : MSubsystem;
  subsystemP : PSubsystem;
  subsystemL : LSubsystem;
thread:
  subsystemM.Create;
  subsystemP.Create;
  subsystemL.Create;

  subsystemM.SetDownPeer(subsystemP);
  subsystemP.SetUpPeer(subsystemM);
  subsystemP.SetDownPeer(subsystemL);
  subsystemL.SetUpPeer(subsystemP);

  subsystemM.Start;
  subsystemP.Start;
  subsystemL.Start;
end MSubsystem;
  
```

← Creation Phase

← Exchange ID Phase

← Starting Phase

Figure 65 DRAGOON - MTU System Body

4.4 Summary

We investigate the creation of a small-scale framework for capturing the design of various data link protocols in a *point-to-point* communication system. The small-scale framework represents a step towards creating a reusable abstract design for communication systems. We attempt to promote software reuse in the MTU system by using a timethread specification to identify: 1) *coordination objects* that embody sequencing and coordination information; and 2) *policy objects* that are responsible for performing activities.

Creating class hierarchies of *coordination objects* and *policy objects* may offer the possibility of a combinational mix of various kinds of objects to address different system requirements. For example, one may imagine creating class hierarchies of retries *policy objects*, in the simple case, the number of retransmissions (Retries Policy) may be fixed, but one may imagine that it may vary in function of the load of the network, or based on past history⁷.

We explore the creation of reusable abstractions by analyzing timethreads. We analyze a set of commonly used data link protocols by drawing timethreads for them. Drawing the timethreads helps in the discovery of the commonalities and the specializations of the various protocols. Analyzing the various timethreads help us to generalize the behaviour of the various data link protocol in terms of abstract events and activities occurring along timethreads. This generalization serves as the basis for defining an abstract class to capture the commonalities of the data link protocols. We refine the abstract class with a set of subclasses to capture the specializations of the protocols.

Creating reusable abstractions is important but difficult [34]. Reusable abstractions are unlikely to be right the first time and tend to evolve with attempts to reuse them in new contexts. In our example, we attempt to reuse the data link protocol object in different process configurations. This suggests us that a more general solution for defining data link protocol objects may be desirable. Dividing the state information contained in the data link protocol may be a potential approach for improving the reusability of data link protocol objects.

We also investigate the creation of reusable abstractions by analyzing interface protocol. We investigate OO techniques to facilitate reuse of components in different environment. We observe that applying OO techniques to promote software reuse often leads to adding a level of messaging between cooperating objects (for example, adding intermediary protocol adaptor objects). Applying these techniques in real-time systems

7. We acknowledge group discussions with Casselman which collaborated in solidifying these ideas.

must to be weighted with other factors such as performance of time critical operations and the complexity of the control flow in the system.

We explore a partial implementation of the MTU system in the language DRAGOON. We create class hierarchies to ease the implementation of the MTU system. One may further increase software reuse in the implementation of the MTU by pre-defining a set of interfaces of subsystems for various canonical actor interaction structures [7]. In our example, we start with a specification of interfaces for the MTU system (such PutUp and PutDown, refer to Figure 34). Figure 66 illustrates various canonical actor interaction structures with different layouts of interfaces for subsystems. On the right side, the figure illustrates the MTU's subsystems with a set of interfaces to accommodate various forms of actor interactions.

Figure 66 Pre-defined Interfaces for Various Actors Interaction Structures

Chapter 5 Conclusions

5.1 A Research Summary

Developing software for real-time embedded systems is one of the most difficult and challenging fields of software engineering. The difficulties force system designers to place an increasing emphasis on software reuse. One important aspect of software reuse is the discovery of reusable abstractions. In this thesis, we have explored an approach based on timethreads for helping real-time designers to discover reusable abstractions while focusing on real-time issues. We have illustrated the *timethreads-oriented* approach in the context of a small-scale communication system example. The *timethreads-oriented* approach helps us in discovering the commonalities and the specializations of various data link protocols in the communication system, as well as, driving the design of class hierarchies to capture these commonalities and specializations.

Summaries of individual research efforts are presented below.

1. Enhancing Software Reuse in Real-time Systems - We have illustrated, by means of examples, the use of conventional OO techniques to enhance software reuse in real-time systems. We have used conventional OO techniques to enhance two aspects of software reuse: 1) *programmatic aspects* which are related to enhancing code reuse; and 2) *architectural aspects* which are related to enhancing component reuse. We discuss these aspects below.

- *Programmatic aspects* - In Chapter 3, we illustrated the use of class inheritance to enhance code reuse opportunities by objectifying various data elements in the network simulator program.

In Chapter 4, we illustrated the creation of a small-scale framework to capture the design of various data link protocols in the MTU system. The small-scale

framework enhances reuse opportunities of algorithm and control code, as well as, individual components.

- *Architectural aspects* - In Chapter 4, we illustrated the use of conventional OO techniques to ease the integration of objects in the MTU system. We illustrated two techniques to ease the integration of objects in different environments: 1) *output ports* which were simulated with methods; and 2) *protocol adaptors* which involve inserting of intermediary objects. Using conventional OO techniques enhances the likelihood of reusing a given object in new environments.

2. Visualizing OORT programs - We have illustrated the use of different published visual design notations in a coordinated manner to express the designs of OORT programs. We use the MachineCharts notation to express real-time issues such as concurrency or tasking interaction, as well as, the operational aspects of OORT programs. We use OMT's class diagrams to express the relationships such as *is-a* or *part-of* between various classes of objects in OORT programs.

3. Refining the Timethread Notation - We have illustrated the use of annotated timethreads to reason about creation and destruction of instances in time and place. In particular, we have illustrated the use annotated timethread to reason about dynamic allocation of instances from a system level perspective (Section 3.3, Section 4.3).

4. Mapping to implementation technologies - We have mapped design diagrams to implementation technologies. In Chapter 3, we have mapped the re-engineered version of the network simulator program to Ada-9x. In Chapter 4, we have mapped the design diagrams of the MTU system to DRAGOON.

5.2 Testing Strategies and Observations

This section discusses testing strategies and observations made during the development of this thesis. We summarize the testing strategies and observations in relation to the goals of the thesis as outlined below.

The timethread-oriented approach

We used the MTU example in Chapter 4 to exercise the *timethread-oriented* approach. We observed that the timethread-oriented approach helped in discovering reusable abstractions. In particular, the *timethread-oriented* approach helped: 1) to identify objects (*coordination* objects, and *policy* objects); and 2) to design classes from which these objects are constructed. We also observed that the timethread approach helps to reason

reason about other design issues such as layouts of interfaces, process allocation or dynamic memory management. Some of these design issues may be inter-related. For example, the layout of interfaces of objects may be influenced by other criteria such as process configuration. The *timethread-oriented* approach facilitates the reasoning of different design issues under a common frame of reference.

Enhancing Software Reuse in Real-Time Systems

We used the two case studies to experiment with OO techniques to enhance software reuse opportunities in real-time system. We observed that code reuse could be achieved through class inheritance. For example, in Chapter 3, we objectify packets in the network simulator program where different kinds of packet objects such as DataPkt and AckCreatePkt packets share common structure and behaviour (refer to Figure 16).

We used the MTU system example to exercise different OO techniques to ease the integration of objects in different environments (e.g., simulated output ports, or protocol adaptor objects). We observed that using *output ports* (simulated) helped integrating objects in different tasking environment as long as the state information contained in these objects does not need to be distributed among tasks (Section 4.2.1). We also observed that applying either of the techniques often leads to adding levels of messaging between objects.

Visualization of OORT Programs

We use the two case studies to verify the feasibility of using different visual design notations to depict OORT programs. The criteria for selecting a good notation are subjective and hard to isolate, but basically have to do with simplicity, clarity, expressiveness, and the capability to capture the essence of designs. The criteria for evaluating the result of combining two design notations are likely to be of the of same kinds. We observed that we could express design of OORT program relatively clearly by separating real-time concerns (which we expressed using the MachineCharts notation) from 'data-oriented' concerns (which we expressed using OMT's class diagrams).

5.3 Assessment of Stated Goals

We believe that the *timethread-oriented* approach is relatively easy to understand because it is based on an approach that emphasizes end-to-end behavioural patterns. Emphasizing end-to-end behaviour enables one to take a more global and intuitively cleaner approach to describe behaviour in real-time systems [53]. The *timethread-oriented* approach helps us to discover reusable abstractions from a system level perspective. Although generalizing the timethread specifications in terms of abstract events and

activities may be difficult: translating the generalized timethread specifications to abstract class definitions appears to be relatively straightforward. Another attribute of the approach is that it enables one to reason about other design issues such as layout of interfaces, process configuration and dynamic instance allocation using the same timethread specification.

OO techniques may be used to enhance software reuse opportunities in real time systems. OO techniques such as *protocol adaptors* increase the likelihood of reusing a given component in new environments. It appears that the flexibility of a system is related to level of indirection between components that form the system. OOPLs supporting polymorphism (and dynamic binding) add a level of indirection between cooperating components (e.g., look-up table). OO techniques such as *protocol adaptors* further add levels of indirections. In the context of real-time systems, applying these OO techniques must to be weighted with other issues such as performance requirements.

5.4 Assessment of Engineering Value

We assess our work in relation to its applicability in practical design environments as outline below.

The timethread-oriented approach

The applicability of the timethread-oriented approach to practical examples of larger scale is still to be determined. The MTU system example incorporates real-time issues commonly encountered in real-time systems of larger scale. We believe that the *timethread-oriented* approach would scale up to express real-time concerns in larger systems. However, we envision that the functional complexity of large practical systems may be an obstacle to applying the *timethread-oriented* approach because, as others have pointed out [6], [37], [47], raising the level of abstractions in large complex systems is difficult. The *timethread-oriented* approach depends on appropriately abstracting the functional complexity of a system. In the case of the MTU system, the relatively simple functionality of the various data link protocols permits us to abstract them in terms of events and activities along timethreads. However, abstracting the functionality of large systems such as telecommunication switching systems is likely to be much more difficult.

Illustrative examples of OORT programs

Our work should be helpful to practical designers who combine elements of OO design and real-time design by showing, through a set of examples, some aspects of using OO techniques in the development of real-time systems. Particularly, we address some aspects of introducing OO techniques in existing real-time programs.

Tool support

Our work heavily uses visual design notations. Even for a small number of design diagrams, drawing and maintaining these diagrams represent a significant amount of work. To make our approach applicable to practical environments, tools to help drawing and maintaining design diagrams would be required.

5.5 Future Work

From the research activities occurring in this thesis, we identify issues that may form the basis of future research. These issues are summarized below:

- 1) Can we apply the *timethreads-oriented* approach to problems of larger scale? We illustrated a *timethread-oriented* approach for helping to discover reusable abstractions in a small real-time system. These abstractions drove the design of a small-scale framework representing a step towards creating a reusable abstract design for communication system. Research into applying the *timethread-oriented* approach to problems of larger scale such as telephone switching systems (call processing in particular) appears desirable.
- 2) Can we convey reusability concerns in system architecture diagrams? From an architectural perspective, it may desirable to visually suggest which components and the structure that they compose that were foreseen as likely or unlikely to change during the evolution of a software system. For example, in a layered communication system based on the OSI model, the structure of the layers such as Physical, Link and Network layers is likely to remain the same over the evolution of the system, whereas the internal of layers or the number of virtual circuits are more likely to evolve. During the conception of the system, the designer may intend to suggest that certain parts of the system were foreseen to stay the same over the evolution of the system and that modifying that structure may have a large impact. For example, in a communication system, adding intermediate layers between existing ones would have a significant effect on the structure of the system, whereas adding new virtual circuits may not. Given the importance of software reuse, future research into visual notations for depicting reusability concerns appears desirable.
- 3) Can we use timethreads to reason about system integration issues? When reporting experiences in software reuse, Berlin [4] identifies areas of conflicts when integrating well-designed but independently-developed software components including: 1) error handling; and 2) argument validation. Berlin says that error

handling requires recovering from problems at any level of the system. Recovery may include operations such as cleaning-up of inconsistent data, releasing resources, or undoing partial operations. Some of these operations may be best dealt from a system level perspective. Timethreads illustrate *end-to-end* behavioural patterns in software systems. In Chapter 4, we used annotated timethreads to reason about dynamic memory allocation from a system level perspective. Extending timethreads to reason about error or exception handling in relation to the normal flow of activities appears desirable. Investigating the use of timethreads to reason about other areas of conflicts such as arguments validation (illustrating *where* and *how* validation takes place) in the system also appears desirable.

References

- [1] C. Atkinson, *Object-Oriented Reuse, Concurrency and Distribution*, ACM press, Addison-Wesley, Reading, MA, 1991.
- [2] C. Atkinson, S. Goldsack, A. Di Maio, and R. Bayan, "Object-Oriented Concurrency and Distribution in DRAGOON," *Journal Object Oriented Programming*, Vol. 2, No. 6, March/April 1990, pp. 11-19.
- [3] D. L. Bailey, *A Framework for Developing Object-Oriented Real-Time Software*, Master's thesis, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, 1989.
- [4] L. Berlin, "When Objects Collide: Experience with Reusing Multiple Class Hierarchies," *Proceedings of OOPSLA/ECOOP'90*, Vol. 25, No. 10, pp. 181-193.
- [5] T. Biggerstaff (ed.), *Software Reusability*, Vol.1, ACM Press, Addison-Wesley, Reading, MA, 1989.
- [6] G. Booch, *Object Oriented Design with Applications*, Benjamin/Cummings, Redwood City, CA, 1991.
- [7] R.J.A. Buhr, *System Design With Ada*, Prentice-Hall, Englewood Cliffs, NJ, 1984.
- [8] R.J.A. Buhr, *Practical Visual Techniques in System Design with Applications to Ada*, Prentice-Hall, Englewood Cliffs, NJ, 1990.
- [9] R.J.A. Buhr, R. S. Casselman, "Architectures with Pictures," *Proceedings of OOPSLA/ECOOP'92*, Vol. 27, No. 10, pp. 466-483.
- [10] R.J.A. Buhr, R.S. Casselman, "Case Studies in Threads and Contracts," SCE-92-06, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, 1992.
- [11] R.J.A. Buhr, R.S. Casselman, M. Petras, J. Anderson, "Object-Orientation and the Design of Large Real-Time Systems," SCE-91-38, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, 1992.
- [12] R.J.A. Buhr, G.M. Karam, H.J. Scott, and R. Du, "Active ADTs in Ada Multitasking System Design," *Proceedings of Canadian Conference on Electrical and Computer Engineering*, Vol. 1, Sept. 1990, pp. 19.3.1-19.3.4.
- [13] R.J.A. Buhr, "Object-Oriented Real-Time System Design," Course Notes, 94.579 F92, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, 1992.
- [14] R.J.A. Buhr, "Real-Time System Design," Course Notes, 94.333 W92, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, 1992.
- [15] R. Casselman, R.J.A. Buhr, and F. Pomerleau, "Notations for Threads," SCE-92-07, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, 1992.

- [16] R.J.A. Buhr, "Pictures That Play: Design Notations for Real-Time & Distributed Systems", SCE-93-04, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, 1993.
- [17] C. Chambers, "Object-Oriented Multi-Methods in Cecil," *Proceedings of ECOOP'92*, pp. 33-56.
- [18] P.P.S. Chen, "The Entity-Relationship Model-Towards a Unified View of Data," *ACM Transactions on Database Systems*, Vol.1, No. 1, March 1976, pp. 9-36.
- [19] P. Coad, and E. Yourdon, *Object-Oriented Analysis*, Prentice-Hall, Englewood Cliffs, NJ, 1990.
- [20] P. Coad, "Object-Oriented Patterns," *Communications of the ACM*, Vol. 35, No. 9, September 1992, pp. 152-171.
- [21] D. Coleman, and F. Hayes, "Lessons from Hewlett-Packard's Experience of Using Object-Oriented Technology," *Proceedings of TOOLS 1991*, pp. 327-333.
- [22] B. Cox, *Object-Oriented Programming, An evolutionary Approach*, Addison-Wesley, Reading, MA, 1987.
- [23] P. Deutsch, "Design Reuse and Frameworks in the Smalltalk-80 System," *Software Reusability*, Volume 2, ACM Press, 1989, pp. 55-71.
- [24] A. Elliot, R.J. Gauthier and P.H. Welch, "Component Engineering in Ada-Some Problems and Some Advice," *Software Reuse with Ada*, ECSC-EEC-EAEC, Brussels-Luxembourg, 1990, pp. 39-50.
- [25] A. Goldberg and D. Robson, *Smalltalk-80; The Language and its Implementation*, Addison-Wesley, Reading, MA, 1983.
- [26] S. Goldsack, and C. Atkinson, "DRAGOON: A Tool for the Ada Programmer," *ACM TRI-Ada'91*, pp. 348-359.
- [27] D. Halbert and P. O'Brien, "Using Types and Inheritance in Object-Oriented Programming," *IEEE Software*, Vol. 4, September 1987, pp. 71-79.
- [28] D. Harel, "Statecharts: A Visual Formalism For Complex Systems," *Science of Computer Programming*, Vol. 8, 1987, pp. 231-274.
- [29] K.J. Hebel and R.E. Johnson, "Arithmetic and Double Dispatching in Smalltalk-80," *Journal of Object-Oriented Programming*, Vol. 2, No. 6, March/April 1990, pp. 40-44.
- [30] R. Helm, I. Holland, and D. Gangopadhyay, "Contracts: Specifying Behavioural Compositions in Object-Oriented Systems," *Proceedings OOPSLA/ECOOP'90*, Vol. 25, No. 10, pp. 169-180.
- [31] D. Ingalls, "A Simple Technique for Handling Multiple Polymorphism," *Proceedings of OOPSLA/ECOOP'86*, Vol. 21, No. 11, pp. 347-349.
- [32] Y. Ishikawa, H. Tokuda, and C. Mercer, "Object-Oriented Real-Time Language Design: Constructs for Timing Constraints," *Proceedings of OOPSLA/ECOOP'90*, Vol. 25, No. 10, pp. 289-298.

- [33] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, *Object-Oriented Software Engineering (A Use Case Driven Approach)*, Addison-Wesley, Reading, MA, 1992.
- [34] R. E. Johnson and B. Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming*, Vol. 1, No. 2, June/July 1988, pp. 22-25.
- [35] R. E. Johnson, "Documenting Frameworks using Patterns," *Proceeding of OOPSLA/ECOOP'92*, Vol. 27, No. 10, pp. 63-76.
- [36] G. M. Karam, *A Tool-set for Temporal Analysis in a Design Environment for Concurrent System*, Ph.D. Thesis, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, 1986.
- [37] C.W. Krueger, "Software Reuse," *ACM Computing Surveys*, Vol. 24, No 2, June 1992, pp. 132-180.
- [38] S. Lippman, *C++ Primer, 2nd Edition*, Addison-Wesley, Reading, MA, 1991.
- [39] B. Meyer, *Object-Oriented Software Construction*, Prentice-Hall, Englewood Cliffs, NJ, 1988.
- [40] B. Meyer, *Eiffel: The language*, Prentice-Hall, Englewood Cliffs, NJ, 1992.
- [41] O. Nierstrasz, "A Survey of Object-Oriented Concepts," *Object-Oriented Concepts, Databases and Applications*, ACM Press and Addison-Wesley, Reading, MA, 1989, pp. 3-21.
- [42] H. Nussbaumer, *Computer Communication Systems, Volume 1*, Wiley and Sons, New York, NY, 1990.
- [43] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-Oriented Modelling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.
- [44] J. Rumbaugh, "Controlling Propagation of Operations using Attributes on Relations," *Proceedings of OOPSLA/ECOOP'88*, Vol. 23, No. 11, pp. 285-296.
- [45] H.J. Scott, "Examining a Communications System Simulation," 94.531 Course Project Report, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, 1989.
- [46] B. Selic, G. Gullekson, J. McGee, and I. Engelberg, "ROOM: An Object-Oriented Methodology for Developing Real-Time Systems," *Proceedings of CASE'92, Fifth International Workshop on Computer-Aided Software Engineering*, July 1992, pp. 230-240.
- [47] M. Shaw, "Large Scale Systems Require Higher-Level Abstractions," *Proceedings of the Fifth International Workshop on Software Specifications and Design*, ACM press, 1989, pp. 143-146.
- [48] S. Shlaer and S. Mellor, *Object Lifecycles, Modelling the World in States*, Prentice-Hall, Englewood Cliffs, NJ, 1992.
- [49] W. Stallings, *Data and Computer Communications, Second Edition*, Macmillan Publishing, New York, NY, 1988.

- [50] D. Taenzer, M. Ganti, and S. Podar, "Object-Oriented Software Reuse: The Yoyo problem," *Journal of Object-Oriented Programming*, September/October 1989, Vol. 2, No. 3, pp. 30-35.
- [51] A. S. Tanenbaum, *Computer Networks*, Prentice-Hall, Englewood Cliffs, NJ, 1988.
- [52] D.A. Thomas, W. R. LaLonde, J. R. Pugh, "ACTRA-A Multitasking/Multiprocessing Smalltalk," SCS-TR-92, School of Computer Science, Carleton University, Ottawa, Ontario, Canada, K1S 5B6, 1986.
- [53] M. Vigder, *Applying Formal Techniques to the Design of Concurrent Systems*, Ph.D. Thesis OCIEE-92-03, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, 1992.
- [54] R. Wirfs-Brock, B. Wilkenson, and L. Wiener, *Designing Object-Oriented Software*, Prentice-Hall, Englewood Cliffs, NJ, 1990.
- [55] P. Zave, "A Compositional Approach to Multiparadigm Programming," *IEEE Software*, Vol. 6, September 1989, pp. 15-25.
- [56] G. Kiczales, J. des Rivieres, and D. Bodrow, *The Art of the Metaobject Protocol*, The MIT Press, Cambridge, MA, 1991.
- [57] Ada-9x Mapping, Volume I, Mapping Rational, Version 3.1, Intermetrics, Cambridge, Massachusetts, August 1991.
- [58] Ada-9x Mapping, Volume II, Mapping Specification, Version 3.1, Intermetrics, Cambridge, Massachusetts, August 1991.
- [59] Ada-9x Project Report, "Introducing Ada-9x", Office of the Under Secretary of Defense for Acquisition, Intermetrics Inc., Cambridge, Massachusetts, February 1993.

Appendix A

This appendix summarizes object-oriented programming languages used in this thesis.

A.1 Overview of the Ada-9x Language Proposal

Ada was developed in the late '70 by an international team sponsored by the United States Department of Defense. Ada is a strongly typed language for implementing large, complex software systems, especially real-time and embedded systems. Ada became an ANSI standard in 1983 and an ISO standard in 1987.

Ada was introduced with the aim of replacing the vast number languages and ad hoc techniques previously used with a standard programming paradigm that could be used throughout the industry to promote software reuse and programmer productivity [1]. Atkinson [1] describes that the introduction of Ada has not turned out to be the panacea for reuse that was originally hoped; in particular, generics have proved more difficult to use than was foreseen.

In 1988, a revision of the language commonly known as Ada-9x was undertaken. One of the important criteria of the language revision team is upward compatibility with most existing Ada applications. The enhanced functionality of the Ada-9x language is obtained by generalizing existing Ada features, and by removing special cases and restrictions [57]. Recent developments in object-oriented design methods, object-oriented programming languages and real-time programming have influenced the revision of the language.

Ada-9x supports single inheritance through type extensions. Ada-9x generalizes Ada-83's type facilities to provide more powerful mechanism for variant and class-wide program development and composition. In Ada-83, derived types provide a simple inheritance mechanism in which derived types inherit exactly the structure, operations, and any literals defined in their parent type. Also, in Ada-83, derived types may be augmented with additional operations. In Ada-9x, a record or a private type may be extended with new components or additional operations, or may override inherited operations.

Ada-9x introduces a mechanism for user-defined classes to support the paradigm of variant and class-wide programming. In Ada-83, variant programming may be accommodated using variant records. Figure 67 shows an example of an alarm system where different levels of alarms are defined. Different actions are performed in response to the arrival of different levels of alarms. The code embodies the variants of actions which must be distinguished during the execution of the program. For example, the HANDLE

procedure includes a case statement which tests the level of the incoming alarm, and forces the execution of specific actions accordingly.

```

package ALERT_SYSTEM is
  type PRIORITY is (LOW, MEDIUM, HIGH);
  type DEVICE is (TELETYPE, CONSOLE, BIG_SCREEN);
  type ALERT (P: PRIORITY) is
    record
      TIME_OF_ARRIVAL : CALENDAR.TIME;
      MESSAGE : TEXT;
      case P is
        when LOW => null;
        ACTION_OFFICER : PERSON;
        case P is
          when LOW | MEDIUM => null;
          when HIGH => TIMEOUT_AT : CALENDAR.TIME;
        end case;
      end case;
    end record;
  procedure DISPLAY(A: in out ALERT; on DEVICE);
  procedure LOG (A: in out ALERT; on DEVICE);
  procedure HANDLE (A : in out ALERT);
end ALERT_SYSTEM;

procedure HANDLE (A : in out ALERT) is
  A.TIME_OF_ARRIVAL := CALENDAR.CLOCK;
  LOG(A);
  DISPLAY(A, TELETYPE);
  case A.P is
    when LOW => null; -- do nothing in case of low alarms
    when MEDIUM | HIGH =>
      A.ACTION_OFFICER := ASSIGN_VOLUNTEER;
      DISPLAY(A, CONSOLE);
      case A.P is
        when LOW | MEDIUM => null;
        when HIGH =>
          DISPLAY(A, BIG_SCREEN);
          -- Do other stuff for high level alarms
        end case;
      end case;
  end case;
end HANDLE;

```

Figure 67 Ada-83 - Example of Variant Programming

The code of the HANDLE procedure is structurally complex because of the relationship between the MEDIUM and HIGH level alerts; and may require careful editing if new levels of alerts were implemented in the system. For example, the variant record called ALERT and the case statement in the HANDLE procedure may need to be modified if new levels of alerts were added such as VERY_HIGH_ALERT. Modifying code which uses variant records may be difficult and error-prone; and in many cases, such as adding new levels of alerts in our example, is likely to increase the structural complexity of the code (HANDLE procedure).

Figure 68 shows the Ada-9x equivalent code for the alert system. With Ada-9x, the programmer can use tagged type extension and subprogram dispatch to simplify the system, handling each variant as a derived type extension, eliminating variants and case statements. The commonalities and specializations of the different levels of alerts have been captured by defining an inheritance hierarchy. The figure shows a set of packages which encapsulate type definitions and a set of operations supported by the types. For explanatory purposes, we name the packages with the string "CLASS" to mimic OOPs supporting the class construct. The ALERT_CLASS package defines the features and operations common to all alerts. The other packages specialize the ALERT_CLASS either by adding elements to the ALERT type, or refining the operations supported by the root type. The keyword *tagged* enables run-time dispatch to occur.

Figure 68 Ada-9x - Variant Programming - Type Definitions

Figure 69 shows the code bodies for the different alert packages. The figure shows that instead of a single procedure (HANDLE) employing a case statement in the Ada-83 version, the Ada-9x solution distributes algorithms for handling alerts to the appropriate alert types.

```

package body ALERT_CLASS is
  procedure DISPLAY (anAlert : ALERT; on : DEVICE);
    -- Print message on device;
  end DISPLAY;
  procedure LOG(AnAlert : ALERT) is
    -- Write to disk
  end Log;
  procedure HANDLE(anAlert : in out ALERT) is
    anAlert.TIME_OF_ARRIVAL := CALENDAR.CLOCK;
    LOG(anAlert);
    DISPLAY(anAlert,TELETYPE);
  end HANDLE;
end ALERT_CLASS;

package body MEDIUM_ALERT_CLASS is
  procedure HANDLE(anAlert : in out MEDIUMALERT) is
    HANDLE(LOW_ALERT(anAlert));
    DISPLAY(anAlert.CONSOLE);
  end HANDLE;
end MEDIUM_ALERT_CLASS;

package body HIGH_ALERT_CLASS is
  procedure HANDLE(anAlert : in out HIGH_ALERT) is
    HANDLE(MEDIUM_ALERT(anAlert));
    DISPLAY(anAlert.BIG_SCREEN);
  end HANDLE;
end HIGH_ALERT_CLASS;

```

Call Ancestor

Figure 69 Ada-9x - Variant Programming - Code Body

The Ada-9x proposal includes numerous enhancements to the Ada-83. One of them relates data-oriented synchronization mechanisms and is referred as: *protected objects*. Protected objects provide a low-level, lightweight, data-oriented synchronization mechanism; they are primary for support of real-time applications [59]. In general concept, protected objects are very similar to monitors as found in Modula. They are passive constructions with synchronization provided by the language run-time system. However, they have an advantage over monitors because the protocol of a protected objects may be described by barrier conditions instead of the low-level signals of monitors.

Other enhancements relate the programming-in-the-large aspects of software developments. A complete description of the proposed enhancements to the Ada-83 language can be found in [57], [58], [59].

A.2 Overview of the DRAGOON Language

The DRAGOON¹ language emerged from the European Esprit project called DRAGON [1]. The DRAGON project investigated the development of tools and methods to support the goals of distribution and reuse in Ada. DRAGOON has been largely inspired from the language Eiffel [39], [40]. Like Eiffel, DRAGOON is a strongly typed language and supports multiple inheritance with facilities for deferring and redefining methods. Also like Eiffel, DRAGOON supports generic classes which may be parameterized with data and class types, but goes further than Eiffel by permitting the inheritance of generic classes. Unlike Eiffel, DRAGOON does not support automatic memory management (garbage collection).

Like most OOPs, DRAGOON supports the concept of classes which are seen as object types from which many identical objects may be generated. Unlike many OOPs, DRAGOON supports a mixed-paradigm and allows a complementary treatment of both 'values' and 'objects' by supporting all of the data typing features of Ada [2]. DRAGOON also permits the use of other conventional features of Ada such as packages (with some constraints), subroutines, and exceptions. DRAGOON restrains packages to be stateless; in DRAGOON state variables must be encapsulated in instances of classes. In DRAGOON, packages serve as 'template units' since they primarily contain declarations acting as 'templates' for the subsequent generation of variables. DRAGOON also differs from most OOPs by adopting the Ada convention of physically separating the specification (interface) and body (implementation) of class definitions.

The approach to synchronization in DRAGOON is based on the philosophy of separating, as far as possible, concerns for synchronization and functionality [26]. The functionality of a method is described in a normal sequential class (concrete) without concern for synchronization. The required synchronization constraints (or the required behaviour) for the problem at hand is then superimposed on the sequential method using a form of multiple inheritance called *behavioural inheritance*. The abstract behaviour is modelled as a special kind of class, called a *behavioural class*, which specifies synchronization conditions using logic predicates over abstract methods sets [1].

Figure 70 illustrates the concept, on the left side, the figure shows that a behaved class inherits from both a sequential class and a behavioural class. On the right side, the figure shows an example of an unibuffer class (holding one element). A MUTEX

1. Distributable, Reusable Ada Generated from an Object-Oriented Notation.

behaviour is superimposed on the sequential class to create a protected unibuffer. The protected unibuffer prevents mutual accesses from concurrent threads of control.

Figure 70 DRAGOON - Behavioural Inheritance

The superimposition of a behavioural class over a sequential class involves matching the methods of the sequential class with the abstract operation defined in the behavioural class. Figure 71 shows an example of a behavioured class called MUTEXUNIBUFFER which is defined from a behavioural class called MUTEX (refer Figure 72) and a sequential class called UNIBUFFER.

Figure 71 DRAGOON - Defining Sequential and Behavioured Classes

DRAGOON's authors describe that although in principle, there is an infinite number of different conceivable behavioural classes, they observe that in practice, a relatively small number of them is required to satisfy most applications. Figure 72 shows common behavioural classes.

```

-- per(OPS)  $\Leftrightarrow$  <boolean condition>
-- per(OPS)  $\equiv \bigvee_{op \in OPS} \text{per}(op)$ .
-- req(OPS) = number of requests for execution of methods in the set OPS.
-- act(OPS) = number of activations of methods in the set OPS.
-- fin(OPS) = number of completions of methods in the set OPS.
-- (><) = exclusion symbol

active(OPS) = act(OPS) - fin(OPS) -- number of currently active methods in the set OPS.
requested(OPS) = req(OPS) - act(OPS) -- number of outstanding calls for methods in the set OPS.

behavioural class MUTEX is
  ruled OPS;
  where
    per(OPS)  $\Leftrightarrow$  active(OPS) = 0;
    -- There is a permission to activate a method in OPS iff there are no members of the set
    -- currently active.
  end MUTEX

behavioural class SEQUENCER is
  ruled FOP, SOP; -- First OPERATION, Second OPERATION
  where
    per(FOP)  $\Leftrightarrow$  ACT(FOP) = fin(SOP)
    per(SOP)  $\Leftrightarrow$  fin(FOP) > act(SOP);
    -- There is a permission to activate a method in the set of FOP iff, for every activation of a
    -- FOP there has been a completion of a SOP. While SOP methods may be activated iff, the
    -- number of completions of FOP exceeds activations of SOP's methods.
  end SEQUENCER;

behavioural class READERSWRITERS is
  ruled ROP, WOP; -- Read OPERATION, Write OPERATION, Writers have priority
  where
    per(WOP)  $\Leftrightarrow$  (><); -- Zero activations of all methods mentioned in the behaviour
    per(ROP)  $\Leftrightarrow$  active(WOP) = requested(WOP) = 0;
    -- There is a permission to activate a method in the set of ROP iff, there are no currently
    -- executing WOP and there are no currently outstanding (i.e. queued) calls for WOP methods.
  end READERSWRITERS;

behavioural class BOUNDEDBUFFER is
  ruled GET_OPS, PUT_OPS, FULL_GUARD, OTHER_OPS;
  where
    per(PUT_OPS)  $\Leftrightarrow$  (><) and (not FULL_GUARD);
    per(GET_OPS)  $\Leftrightarrow$  (><) and (act(PUT_OPS) - act(GET_OPS) > 0);
    per(FULL_GUARD)  $\Leftrightarrow$  (><);
    per(OTHER_OPS)  $\Leftrightarrow$  (><);
  end BOUNDEDBUFFER;

```

Figure 72 DRAGOON - Common Behavioural Classes

The approach to concurrency in DRAGOON is based on defining *active* objects. An *active* object encapsulates an independent thread of control called a *thread*. A *thread* may be explicitly defined in the class body or inherited from an ancestor. Concurrency may exist at different level of encapsulation in DRAGOON. Although, an active object may possess at most one single sequential thread, it may contain an unlimited number of active objects, which provides it with the illusion of multiple internal threads. The creation of an active object does not automatically result in the activation of its thread. The activation of the thread must be done by means of a special method called START. Figure 73 shows an

example of an active object called PRODUCER which uses the services of a buffer for storing elements. The figure shows that after creation of active thread, the START method must be invoked to activate the thread.

Figure 73 DRAGOON - Active Object

A behaved object may also have an internal thread. A behaved object may impose constraints on the execution of its methods but it may not affect the execution of its thread. As a result, the thread of an active object is able to access its state variables concurrently with its methods (as they are executed by external threads). In DRAGOON, any internal objects in danger of being concurrently accessed must be protected with their own behaviour such as MUTEX. Synchronization between concurrent threads may be achieved by means of behaved objects such as SEQUENCERS.

DRAGOON also provides facilities for distributed systems and heterogeneous run-time environments. Under certain restrictions, an instance of a class may be executed as a separate program on a node of a network. These restrictions concern preventing the sharing of state with other objects, and the limitation of parameters of remote procedure calls to ordinary data types. Instances of classes may not be passed as messages over the network.

Appendix B

B.1 Abstracting Data Link Protocols

This section contains the timethread diagrams for the Sliding Window with Go Back protocol, and the Sliding Window with Selective Repeats protocol. These timethreads help generalizing different data link protocols commonly used in *acknowledged connectionless* services. Refer to Section 4.2.1.

Figure 74 shows a timethread describing the *Sliding Window with Go Back* protocol using a receive window equals to one. When the first unexpected packet (out of sequence) arrives, the receiver does not send an acknowledgment but simply waits for a retransmission, all subsequent packets are discarded. The transmitter's timer eventually expires causing the retransmission of all packets in the Transmit window.

Figure 74 Sliding Window Go Back Protocol - RX Window Size Equals One

Figure 75 shows a timethread describing the *Sliding Window with Selective Repeats*. When the first unexpected packet arrives, the receiver sends a negative acknowledgment. Any subsequent packets are stored in the Receive window (if within the bound). When the transmitter receives the negative acknowledgment, it only retransmits the bad packet. When the expected packet finally arrives, the receiver has a series of consecutive packets which may be processed. At that point, it sends an acknowledgment with a packet number corresponding to the upper end of the Receive window.

Figure 75 Sliding Window with Selective Repeats Protocol

B.2 Abstracting Framing Policies

This section contains two commonly used framing policies. These framing policies are expressed in terms of Finite State Machines to help creating an abstract class for framing policy objects. Refer to Section 4.2.2.

Figure 76 illustrates the *Character Counting* framing policy. The *Character Counting* framing policy uses a field in the frame header to specify the number of characters in the frame. When the frame arrives at destination, the count character is used to determine the end of the frame. This is a very simple framing strategy but it is not reliable because count character may be garbled by a transmission error [51]. On the transmit side, the framing policy object (aFramingPolicy) defines four methods namely: *Get*, *reset*, *size*, and *read*. Similarly, on the receive side, the object called aDeFramingPolicy defines *Put*, *store*, and *reset*.

Figure 76 Framing Policy- Character Count

Figure 77 illustrates the *Starting and Ending Flags with Bit Stuffing* framing policy. With bit stuffing, the boundary between two frames can be unambiguously recognized by the flag pattern (01111110). If the receiver loses track of where it is, all it has to do is scan the input for flag sequences, since they can only occur at frame boundaries and never within the data [51]. Whenever the sender encounters five consecutive bits equal to one in the data, it stuffs a 0 bit into the outgoing bit stream, thus, insuring the flag patterns from appearing in the data. On the transmit side, the framing policy object (aFramingPolicy) defines four

methods namely: *Get*, *reset*, *readBit* (converting characters to bits), and *read*. Similarly, on the receive side, the object called aDeFramingPolicy defines *Put*, *store*, and *reset*.

Figure 77 Framing Policy - Simplified Bit Stuffing

END

15-1 2-93

FIN