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

Your file Votre référence

Our file Notre référence

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.

Using Timethreads to Relate Performance and Design Models

by

Yan Tang, B. C. Sc.

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

**Master of Science
in Information and System Science**

**Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada
January 31, 1995**

**© copyright
1995 Yan Tang**

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

Votre file - Votre référence

Cher file - Notre référence

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 ACCORDE UNE LICENCE
IRREVOCABLE ET NON EXCLUSIVE
PERMETTANT A LA BIBLIOTHEQUE
NATIONALE DU CANADA DE
REPRODUIRE, PRETER, DISTRIBUER
OU VENDRE DES COPIES DE SA
THESE DE QUELQUE MANIERE ET
SOUS QUELQUE FORME QUE CE SOIT
POUR METTRE DES EXEMPLAIRES DE
CETTE THESE A LA DISPOSITION DES
PERSONNE INTERESSEES.

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 PROPRIETE
DU DROIT D'AUTEUR QUI PROTEGE
SA THESE. NI LA THESE NI DES
EXTRAITS SUBSTANTIELS DE CELLE-
CI NE DOIVENT ETRE IMPRIMES OU
AUTREMENT REPRODUITS SANS SON
AUTORISATION.

ISBN 0-612-03009-1

Canada

Name

YAIN TANG

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.

Psychology

SUBJECT TERM

1790

SUBJECT CODE

U·M·I

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture 0729
Art History 0377
Cinema 0900
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 0727
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 0998
Physical 0520

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 0679
General 0289
Ancient 0290
Linguistics 0291
Modern 0401
Literature 0294
General 0295
Classical 0297
Comparative 0298
Medieval 0316
Modern 0591
African 0305
American 0352
Asian 0355
Canadian (English) 0593
Canadian (French) 0311
English 0312
Germanic 0315
Latin American 0313
Middle Eastern 0314
Romance 0314
Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy 0422
Religion 0318
General 0321
Biblical Studies 0319
Clergy 0320
History of 0322
Philosophy of 0469
Theology 0323

SOCIAL SCIENCES

American Studies 0323
Anthropology 0324
Archaeology 0326
Cultural 0327
Physical 0310
Business Administration 0272
General 0770
Accounting 0454
Banking 0338
Management 0385
Marketing 0501
Canadian Studies 0503
Economics 0505
General 0508
Agricultural 0509
Commerce-Business 0510
Finance 0511
History 0358
Labor 0366
Theory 0351
Folklore 0578
Geography 0366
Gerontology 0351
History 0578
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 0615
General 0616
International Law and Relations 0617
Public Administration 0814
Recreation 0452
Social Work 0626
Sociology 0627
General 0938
Criminology and Penology 0631
Demography 0628
Ethnic and Racial Studies 0629
Individual and Family Studies 0630
Industrial and Labor Relations 0700
Public and Social Welfare 0344
Social Structure and Development 0709
Theory and Methods 0999
Transportation 0453
Urban and Regional Planning 0453
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture 0473
General 0285
Agronomy 0475
Animal Culture and Nutrition 0476
Animal Pathology 0359
Food Science and Technology 0478
Forestry and Wildlife 0479
Plant Culture 0480
Plant Pathology 0817
Plant Physiology 0777
Range Management 0746
Wood Technology 0306
Biology 0287
General 0308
Anatomy 0309
Biostatistics 0379
Botany 0329
Cell 0353
Ecology 0369
Entomology 0793
Genetics 0410
Limnology 0307
Microbiology 0317
Molecular 0416
Neuroscience 0433
Oceanography 0821
Physiology 0778
Radiation 0472
Veterinary Science 0786
Zoology 0760
Biophysics 0786
General 0760
Medical 0760

EARTH SCIENCES

Biogeochemistry 0425
Geochemistry 0996

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

Speech Pathology 0460
Toxicology 0383
Home Economics 0386

PHYSICAL SCIENCES

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

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

**Using Timethreads to Relate
Performance and Design Models**

**submitted by Yan Tang, B. C. Sc.
in partial fulfillment of the requirements for
the degree of Master of Science in Information and System Science**

Thesis Supervisor

**Chair, Department of
Systems and Computer Engineering**

**Department of System and Computer Engineering
Faculty of Engineering
Carleton University
January 31, 1995**

Abstract

The thesis describes two techniques that use a timethreads model of a system behaviour as a means of relating activity diagrams to design models based on MachineCharts. The activity diagrams can then be mapped onto performance models using standard techniques. Timethreads and activity diagrams both describe the sequencing of activities over time; thus they are equivalent representations of sequencing information. Timethreads also show the flow of activity over space, thus they can be used to explain architectural design models, such as MachineCharts. The first technique manually maintains concordance between performance and design by associating the activities in an activity diagram, with an equivalent set of timethreads. The timethreads are then modified into a final design, and as a result showing how the activities have been mapped to a design model. The second technique uses a programming model of executable timethreads to control an application (design) model, and thus mechanically recording the concordance through the formality of a programming language. In both cases, the end result is a relation between the activity sequence and the design model that allows for easier construction of performance experiments on the design model (via a prototype or an actual implementation). A case study of a Layered Communications System is performed to evaluate the techniques.

Acknowledgements

I would like to take this opportunity to thank my thesis supervisor Prof. Gerald M. Karam, for his continuous support and encouragement, his patience and help, his spirit and generosity.

I would like to thank all of the professors, staff, students, and researchers of the Department of Systems and Computer Engineering at Carleton for their daily help during my school years.

I would also like to thank my family, especially my husband for his support and baby-sitting; and his ride to school every day. Finally, I thank my wonderful son, Nicholas.

Financial support was provided by NSERC and Carleton University.

Dedicated to my father

Table of Contents

Abstract	iii
Acknowledgements	iv
List of Tables	viii
List of Figures	ix
1 Introduction	1
1.1 Background and Motivation	1
1.2 Thesis Overview	7
1.3 Thesis Outline	8
2 Background	10
2.1 Issues in software life cycles	10
2.2 Data Flow Oriented Approach.	12
2.3 Petri Nets Oriented Approach	15
2.4 Timethreads Oriented Approach	17
2.4.1 Timethreads Notation	18
2.4.2 Activity Diagram.	20
2.4.3 MachineChart Notation.	23
2.5 MLog Programming Language	25
2.5.1 OOPS in MLog	25
2.5.2 Concurrency in MLog.	26
2.5.3 Prolog in MLog.	27
2.5.4 Prototyping Environment in MLog.	27
3 Tracking Design Changes Using Timethreads	29
3.1 Behavior Reasoning with Timethreads	29
3.2 Tracking Changes in Activity Diagram.	32
3.3 Designing with Timethreads	34
3.3.1 Timethreads Design Reasoning.	34
3.3.2 Components Discovery	38
3.3.3 Design Component Collaborations	40
3.3.4 Controlthreads Reasoning.	43
3.4 Module Design and Activities Mapping	45

4 Executable Timethreads	49
4.1 Programming Environment for Executable Timethreads	49
4.2 Introduction to Executable Timethreads	52
4.3 Executable Timethreads Programming Model	54
4.4 Proto-Control Model	68
4.5 Timethreads Control Model	77
5 Case Study	82
5.1 Executable Timethreads for Layered Communications System	82
5.2 Implementation Issues for the Layered Communications System	87
5.3 Timethreads Control Model for Layered Communications System	91
6 Conclusion	97
6.1 A Research Summary	97
6.2 Future Research	99
Bibliography	100
A Appendix: Notation	105
A.1 TimeThread Notation	105
A.2 MachineChart Notation	106
A.3 Activity Diagram Notation	107
A.4 TimeLine Tool	108
B Appendix: MLog Code	110

List of Tables

Table 3-1: Mapping between activities and modules 47

List of Figures

Figure 1-1: Software Development Process based on the Waterfall Life Cycle Model. . .	2
Figure 2-1: Data Flow Diagram of Layered Communications System	13
Figure 2-2: Petri Nets Diagram of Layered Communications System.	17
Figure 2-3: Basic Thread Notation	18
Figure 2-4: Major Timethreads for a Layered Communications System	19
Figure 2-5: Major Activities for a Layered Communications System	21
Figure 2-6: Basic Architecture for a Layered Communications System	24
Figure 3-1: Refinement #1 of Timethreads for Layered Communications System	30
Figure 3-2: Activity Diagram for the Layered Communications System.	33
Figure 3-3: Refinement #1 Overlaid of Timethreads for Layered communications System	35
Figure 3-4: Refinement of Activity Diagram for the Layered Communications System.	37
Figure 3-5: Allocation of Refinement #1 Timethreads to Tasking Elements.	39
Figure 3-6: MachineCharts Design of Layered Communications System	40
Figure 3-7: Detailed Design of Task TX	42
Figure 3-8: MachineChart of Tx and tickTask	43
Figure 3-9: Controlthread Notation	44
Figure 3-10: Controlthread Behaviour for TX Task.	45
Figure 3-11: Completed Design	46
Figure 4-1: The MLog Rendezvous	51
Figure 4-2: Executable Timethreads Control Architecture.	53
Figure 4-3: Concurrent Timethreads.	55
Figure 4-4: Sequences of Timethreads	56
Figure 4-5: Spawning Timethreads.	56
Figure 4-6: And-Join Timethreads	57
Figure 4-7: And-Fork Timethreads.	58
Figure 4-8: Or-Join Timethreads	59
Figure 4-9: Or-Fork Timethreads	60
Figure 4-10: Trigger Timethreads.	61
Figure 4-11: Trigger a Timethreads	62
Figure 4-12: Conditional timed Wait Timethreads.	63

Figure 4-13: Producer-Consumer Timethreads	65
Figure 4-14: TimeLine Tool output for the Producer-Consumer Timethreads	67
Figure 4-15: Class Inheritance Hierarchy.....	69
Figure 4-16: Nested Rendezvous Mechanism	69
Figure 4-17: Producer-Consumer Architecture	73
Figure 4-18: TimeLine Tool output for the Proto-Control of Producer-Consumer example	75
Figure 4-19: TimeLine Tool output of the Rendezvous for Producer-Consumer example .	76
Figure 4-20: Nested Rendezvous Mechanism in Timethreads Control Model	78
Figure 4-21: TimeLine Tool output for Timethreads Control of Producer-Consumer exam- ple	80
Figure 5-1: Timethreads Execution on the Design Picture.....	83
Figure 5-2: TimeLine Tool output of the Timethreads Execution	86
Figure 5-3: MachineChart of the Link Layer Architecture.....	88
Figure 5-4: System Initialization of the Layered Communications System.....	89
Figure 5-5: Runtime Execution of the Layered Communications System.....	90

1

Introduction

1.1 Background and Motivation

Software engineering is a very broad area which involves many aspects. In Schach's book, he defines that software engineering is a discipline whose aim is the production of quality software, that is delivered on time and within budget, and that satisfies its requirements [46]. Furthermore, Jacobson adds that it also should ease its long term maintenance, alteration and extension [30]. People either in industry or research have made many efforts to improve software development efficiency and reliability. Especially, in recent years, software quality control has gained a lot of attention. It has become a key issue to business success.

Software performance engineering (SPE) plays a very important role in the software development process. Smith states in her book [48] that meeting performance requirements (such as responsiveness or throughput) is as vital as meeting requirements of functionality, reliability, and maintainability, yet previous software engineering research has largely ignored performance issues. *Performance* refers to system responsiveness: either the time required to respond to specific events, or the number of events processed in a given time interval [49]. For information systems, performance considerations are often associated with usability issues. In real-time systems, performance concerns can become correctness

issues. Especially for the hard real-time systems, failure to meet deadlines or throughput requirements is equivalent to producing incorrect results [36] [49]. As Ferrari [23] has noted, performance is one of the three fundamental categories of attributes that are indispensable for the viability of any technical system, the other two being *functionality* and *economicity*. Computer systems are no exception to these rules.

Performance models of computer systems often describe behavior of functions and devices using mathematical notations and models [40] [45]. Within the typical software development lifecycle, performance analysis often occurs in either the requirements phase or the implementation and integration phases (as shown in Figure 1-1).

During the implementation and integration phases, performance models are constructed from the implementation and experimental data in order to identify performance problems and to perform implementation tuning [29]. This approach is the more prevalent and follows the typical “build it first and fix it later” attitude of many software development efforts. However, it is commonly conceded that this is often a much

more expensive route to constructing a satisfactory system than establishing correct requirements and designs. When software is delivered to customers, it may not meet the customer's needs, for example, the response time may be slower than expected, or it may cause system failures because of some kind of performance behavior. However, at this point, it will be more difficult and expensive to correct an error. System tuning may not improve the performance dramatically and to remedy problems, sometimes, the whole system needs redesign. The earlier a system flaw can be detected in the life cycle, the lower the overall cost of correcting the error [44].

In the requirements phase, specific functions and their interrelationships are overlaid with available device requirements (such as disk, processors, and communications channels), to yield a performance model, such as *queueing networks* [35]. The resulting model can be used to evaluate and predict performance characteristics. As the development of a system progresses from requirements to design and implementation, decisions are made which are either performance-neutral or performance-sensitive. In the latter case, such decisions typically have introduced devices, or software bottlenecks. However, at this stage, no explicit performance model is usually constructed, and no performance examination is done until integration testing takes place. This also delays the detection of performance errors and degrades the design quality.

Designed-in performance is preferable to tuned-in performance [48]. Thus, the focus of this thesis is how to embed performance analysis in a piecemeal fashion into the software development process. Smith views SPE as an integral part of software development [48]:

“Software performance engineering is a method for constructing software systems to meet performance objectives. The process begins early in the software lifecycle and use quantitative methods to identify satisfactory designs and to eliminate those that are likely to have unacceptable performance, before developers invest significant time in implementation. SPE continues through the detailed design, coding, and testing stages to predict and manage the performance of the evolving software and to monitor and report actual performance against specifications and predictions.”

There are three commonly used performance evaluation techniques: *analytical*

modeling, simulation, and measurement [29]. Analytical modeling and simulation can be used at early software life cycle stage, where measurement is not possible. Measurements are possibly used only if the system already exists (or a prototype exists). Since analytical modeling requires so many simplifications and assumptions, the accuracy could be very low. Simulations take a long time to build the model and to execute the model in order to get a moderate result. Although measurements sound like the real thing, they may not give accurate results simply because of many of the environmental parameters, such as system configuration, type of workload, and time of the measurement. SPE unifies and augments these diverse techniques, and meanwhile, integrates with design methods to achieve the best solution.

The evaluation of computer systems performance has developed in substantial isolation with respect to software engineering [23]. In recent years, a number of software performance engineering approaches have been introduced to find performance problems as early as possible. But those methods lack the direct linkage between the performance model and design model in a more detailed level, especially for real-time systems. Ferrari states that most of the models and analysis techniques used are of a highly abstract nature with little mention on how they may be integrated into real design environments or applied on working systems [23].

The approaches in [21] [42] [51] are based on very high level module precedence relationships in distributed real-time systems. Typically, descriptions or models of the software are constructed using a formal mathematical notation. Assumptions have to be made in order to analyze the system. Although resource contentions and interprocess communication delays are considered, the software design complexity impact on the performance is ignored; software complexity may significantly change the performance. These approaches are similar to the traditional analytical performance-dominated approach. It provides the initial assessment for SPE.

The approaches in [57] [39] [47] are based on performance modeling tools solved with simulation. They help to advance the ability to apply SPE during software design, but still require considerable supplemental work to create and analyze performance models. TimeBench [18] [19] [57] is a prototype CAD tool featuring the ability to experiment with

temporal behavior and performance of Ada systems at the design level before any implementation steps are taken. The performance model used is called *Stochastic Rendezvous Networks* (SRVNs) [58] [27], a form of queueing network adapted for concurrent software entities. The SRVNs is an abstract model. When details of control flow are added, they have to be approximated and one may require a more detailed modeling method. SES/workbench [47] is a design specification, modeling and simulation tool. It also focuses on higher level system architectural issues. Design/CPN [39] is based on the principles of *Colored Petri Nets* which embody mathematical and graphical approaches to system behavior. It consists of an editor for modeling the system and a simulator for executing the model. Smith argues that the time required for solution of complex Petri net models makes them inconvenient for design trade-off studies. The SPE philosophy suggests using simpler models early in development to identify suitable designs. As the software evolves, model extensions analyze contention delays, synchronization and communication behavior. During implementation, the SPE models can produce Petri net models that permit extensive, detailed analysis of timing, synchronization and communication [49]. Furthermore, the animation of typical execution paths is an important feature for the temporal behavior patterns and performance evaluation. There is no such function provided in the simulation tools.

Smith has developed SPE methodology for single sequential programs [48] [49]. But there is no sound SPE method for concurrent systems. It is increasingly clear that more integrated approaches are needed throughout systems development. And particularly in concurrent systems, which may be either distributed and/or parallel, the challenges are greater and the pitfalls are deeper due to greater behaviour complexity of the software and the hardware [56].

However, software development phases are normally separated. Each phase has self-contained notations and methodology. Expertise and information tend to be compartmentalized between experts in these different domains. Furthermore, as software artifacts may be introduced during software development progression, it may be difficult to continually reconcile the performance model and the design model.

As a goal, a designer should be able to work with either model as one can feed

information to the other. The design model, if executable, as in the case of a prototype, can enhance the quality of performance parameters (e.g., choice probabilities or times). The performance model can give estimates and suggest experiments that help select one design alternative over another. Thus, there is considerable value in being able to relate a performance model to a design model at any point in the development.

Timethreads [16] [17] are a notation for specifying end-to-end behaviour of computer-systems; i.e., a timethread describes the sequence of events from a stimulus to a response. A performance model *activity diagram* [56] conveys equivalent information -- it defines the allowable sequences of functions in a computer system. The key difference between the two notations is that timethreads enumerate the various sequences that can occur in a system, and an activity diagram expresses the constraints on functions that imply the various sequences.

Timethreads are useful for describing any aspect of time behaviour in a system at any level of detail -- from the flow of information through subsystems and high level functions, to the detailed call-return paths of a complete software design. In this latter case, the timethread describes a sequence of changes in the locus of control of software objects (as limited by the scope of a single task's thread of control). This type of timethread, which expresses the finest detail of activity, is called a *controlthread*. Refinement techniques are used by a designer to move from coarse to fine levels of detail. As finer detail is revealed, the software design approaches its final form.

As a mechanism for relating performance and design models, timethreads are suitable for two reasons: (1) there is an equivalence between a set of timethreads and a performance model activity diagram; and (2) timethread refinements, which represent progressively greater levels of design detail, can be used to relate how activities are subsequently mapped onto a final design. Given this relationship, a designer can construct a performance experiment in the activity domain, and trace this to a specific experiment that can be conducted on the design model (such as a prototype). Furthermore, the decomposition or allocation of activities can be traced back through the timethreads so that the activity diagram could be refined to correctly portray the current design (and thus account for software bottlenecks).

1.2 Thesis Overview

The prime objective of this thesis is to integrate software performance analysis into the design process to improve software development. In pursuit of this objective, the sub-objectives in this thesis are:

1. To study the software analysis modelling and design modelling techniques to explore the software development transition process and its traceability with respect to performance.
2. To use timethreads as a mechanism for relating performance and design models to achieve manual documentation.
3. To prototype executable timethreads in the MLog environment (multitasking, object-oriented Prolog) [32] [33] to achieve the executable design and control over the execution path that is useful to conduct performance experiments in the timethreads model.
4. To construct simple experiments using a layered communications system as an example, to evaluate timethreads execution in the MLog environment.

The main research of the thesis is to use timethreads as a mechanism for relating performance activity diagram and design models. The thesis describes two techniques that use a timethreads model of a system behaviour as a means of relating performance models based on activity diagrams to design models based on *MachineCharts* [12].

The first technique employs timethreads in a manual fashion: each design refinement, from the requirements to the final design, reports the changes in timethreads. Thus there is a trail to follow from the performance model activity diagram to the software design, and the changes to activities can be tracked along the refinements. The software design could be realized as a high level prototype or portions of an actual implementation, and performance experiments could be conducted accordingly. Also, the activity model could be manually refined to correspond to some point along the design. The problem with this approach is that concordance between the timethreads and the software design must be maintained manually.

The second technique uses a programming model of executable timethreads to control

an application (design) model, and thus mechanically recording the concordance through the formality of a programming language. As timethreads and software objects are refined, the designer modifies the descriptions of the timethreads and software objects and their relation. Execution of the design model becomes execution of the timethreads to control the activities embodied by the software objects. The designer can construct performance experiments such as branching probabilities based on the design. Since the timethreads are the objects that are executing (or at least controlling execution) it is easier to relate the experiments to the objects, and perform measurements. The disadvantage of this approach is the need for a specialized execution environment to support timethread execution.

In both cases, the end result is a relation between the activity sequence and the design model that allows for easier construction of performance experiments on the design model (via a prototyping or an actual implementation). However, in either way, the concordance between timethreads and activity diagram is still maintained manually.

The programming model of executable timethreads in the thesis is based on the MLog environment developed by Carleton University. MLog [33] is a design prototyping language which combines: (1) the packaging features of object-oriented programming systems, (2) the computation model of Prolog, and (3) concurrency based on communicating sequential processes and both rendezvous and asynchronous message passing for interprocess communications. Its primary application is prototyping embedded, concurrent systems. The MLog environment provides tools for debugging and exploring prototype behavior.

A layered communications system has been implemented in the MLog environment. In this case study, simple experiments have been conducted and evaluated based on the design and the programming model of executable timethreads.

1.3 Thesis Outline

The remainder of the thesis is as follows: chapter 2 describes the background of the thesis and the timethread and activity diagram notations and related concepts; chapter 3 shows how timethreads can be used manually to relate performance models to design models; chapter 4 introduces the notion of executable timethreads; chapter 5 documents the case

study of the layered communications system using executable timethreads; and chapter 6 provides conclusions.

2

Background

This chapter provides an overview of software analysis and design methodologies which have been widely used, and summarizes the key issues and concepts relevant to this thesis.

Section 2.1 discusses issues in the software lifecycle. Sections 2.2 and 2.3 review some commonly used software analysis modelling techniques, such as Data Flow analysis and Petri Nets analysis. Section 2.4 introduces the concept of the Timethreads analysis method and gives a brief introduction to timethreads visual notation, activity diagrams, and MachineCharts. Finally, section 2.5 describes the MLog programming language.

2.1 Issues in software life cycles

Software development is carried out in a number of phases, as shown in Figure 1-1, each of which consists of a more detailed and concrete development over the previous phase. Each phase generates its model descriptions. A modeling technique in general is described by means of syntax, semantics and pragmatics [30]. The requirements and specification phase generate an analysis model, which converts requirements into a totally problem-oriented (frequently operational) specification. Based on the analysis model, a performance model is constructed and the performance evaluation is involved. The design phase generates a design model, which contains architecture and module design. At this point, the performance model can be extended to reflect the design reality. The implementation phase generates the machine dependent programming codes. And finally the integration testing model is generated, which contains functional and robustness testing, and performance measurements.

Thus, software development can be viewed as a gradual transformation of a sequence

of models. It is an evolutionary, complex process. In practice, the software life cycle is not a purely linear process. Each phase may be revisited, but software development still accomplishes these phases. The aim in the software development process is to make the transitions between the different models seamless and traceable. That is, the model in one phase can be readily related to the model in the next or previous phase. At present, no sound technique exists. In Jacobson's book, he states [30]:

"The success of the system is highly dependent upon whether it has been possible to capture and formulate the user's requirements in such a way that the requirements can be formalized and transformed into working programs."

In particular, the transition from the analysis model to the design model can be challenging because the mapping is not truly isomorphic. While these two activities are conceptually distinct, in practice the line between analysis and design is frequently blurred. Thus, the transition difficulty obstructs the evolvement of performance modeling.

Alford [2] [3], in SREM (Software Requirements Engineering Method), discusses testability and traceability issues not only on functional requirements, but on performance requirements as well. From the performance engineering perspective, SPE provides coverage throughout the development process to help ensure that the performance requirements are achieved. At the initial stage, analytic performance models using early estimates of data, can help in guiding software design, in assessing the need for distribution of data and functionality, and in early hardware planning. As software development evolves, the design model can help in adding to the accuracy to the performance model. The overhead execution caused by the design and delays due to resource contention can be taken into account. The performance requirements can be further examined. Finally, during the integration phase, performance measurements complete and validate the software model.

The evolution of the performance model progresses parallel to the software development process. There are three analysis strategies that guide the performance model formulation [49]: (1) Adapt-to-precision - to use models that can be constructed quickly, enable rapid evaluation of alternatives, and produce results that distinguish suitable plans from those that are unlikely to meet performance objectives; (2) Simple-to-realistic - to

postpone modeling details of execution, then later as developers make specific design and implementation decisions, these simple models can be extended to more precisely represent execution behavior; (3) Best-and-worst-case - to examine bounds on system performance. This thesis adopts the simple-to-realistic strategy; i.e., as software development evolves, the performance model is extended and reconciled with the software model at each stage.

Furthermore, Smith summarizes five types of data needed for constructing and evaluating the performance models [49] [48]: performance requirements; behavior patterns and intensity; software descriptions; execution environment; and resource usage estimates. She also covers performance data collection and predictions strategies in her book [48]. Those are very important aspects for the performance evaluation but, are beyond the scope of this thesis.

The construction of an accurate performance model before the implementation is not easy. The software performance may depend upon the environment and the target implementation language. Sometimes, it is difficult to emulate system dynamic behavior and key execution paths before implementation. Rapid prototyping is a very useful solution. Prototyping is used to assess risks associated with proposed solutions, algorithms, expected performance, and coding strategies [5]. It can also efficiently provide measurement data for critical software components. Balzer notes that an important purpose of prototyping is answering questions concerning the dynamics of a system, relating to both its behavior and performance. There are existing examples where performance information can be provided by a prototype [37] [60].

2.2 Data Flow Oriented Approach

Modeling techniques pave the way from system analysis and design, to programming. Good and clear diagrams play an essential part in the progress. The data flow oriented approach is to use data flow diagram as a modelling tool to analyze the system. During the design phase, functional decompositions take place. This method has been popular since the 1970s, and is currently used with an object-oriented methodology. A data flow diagram is a tool for top-down analysis. It provides both a high-level and more detailed view of a system.

The *data flow diagram* (DFD) is a graphical technique that depicts information flow and the transforms that are applied as data moves from input to output [44]. The DFD is built from four basic components: the data flow, the process, the data store, and the terminator. The data flow traces the flow of data through a system of processes. The direction of data flow is indicated by an arrow and the data is identified by a label alongside its corresponding arrow. The process is represented by a single *bubble* operating on data through the process. Each process could be refined into a series of bubbles to provide more detailed information. A data store is connected to a process box. A terminator shows a data source or sink; normally they lie outside the DFD.

There are two kind of popular data flow diagrams, one is developed by DeMarco [22], the other by Gane and Sarsen [24]. Both methods are similar in many ways and are equally good. A DeMarco's data flow diagram for a layered communications system is shown in Figure 2-1. In this system, there are three layers of a communications sub-system. Each bubble represents a layer -- the network layer, the link layer, and the physical layer. There is a point-to-point physical link between any two nodes. A packet is send from one node to another on the basis of "best effort delivery", thus this packet is not acknowledged by the network layer at the receiving node. Figure 2-1 shows two instances of the three layers, and a channel connecting the two physical layers. In the network layer, there is a bounded queue. The Alternating Bit Protocol [52] has been used in the link layer. At the sending side,

the network layer transforms incoming messages into packets. The link layer takes the incoming packets and outputs data frames. Then the physical layer converts the data frames into bits and sends them over the cable. At the receiving side, the physical layer converts the bits into data frames. The link layer checks the data frames, packages them, and sends the Ack frame to the physical layer. Finally, the network layer sends the packets to the destination.

After the data flow analysis model is established, the transition from information flow to structure is accomplished as a transform-centered design or a transaction-centered design [59]. These design strategies achieve the module structure by continuously decomposing the bubbles. The control flow structure implements the data flow semantics. Yourdon gives a detailed discussion on these methods. The data flow oriented approach advances the modules high cohesion and low coupling, but, moving from a data flow diagram to design-structure charts needs a significant shift in perspective.

Data flow diagrams are useful for showing the flow of information through functional processes (analogous to activities). Their control semantics are implicit; they only show what happens, not how it happens. The design architectural diagrams show the control semantics. There is no straightforward transition from data flow analysis diagram to design structure to represent system control semantics. Data Flow diagrams do not relate to architectural diagrams -- their only component is the functional process which represents an activity, which does not necessarily have a one-to-one relationship with an architectural structure. While the performance model is built based on the data flow analysis using queuing network techniques, as the software development progresses, the design complexity is hard to reflect on the data flow diagram. This increases the difficulty in feeding information between the performance model and design model.

Furthermore, timing considerations are not handled in DFD. However, timing is a major concern in real-time systems. It can cause synchronization problems, such as critical races and dead-lock. That is to say, if we failed to properly model the specification, it could result in a poor design or a faulty implementation. The Timed Data Flow diagram [53] has been developed to enhance Data Flow diagrams so that it can represent control and timing. However, it is not widely used because its complexity and similarity to Petri Nets.

Besides data flow oriented methods, Jackson's data structure method is very popular too. The method starts from data structure analysis and its main application is information systems. The object-oriented approach is a combination of these two methods. One aspect of an object is that it is a data structure, together with the processes that operate on that data structure [46]. However, for the real-time systems, none of these methods has reached the level that can well handle time constraints and synchronization issues. Most real-time design methods are either the extensions of nonreal-time methods (s.t., extension of structured systems, SDRTS [54]) or Ada-specific real-time methods (s.t., PAMELA [20], and MachineChart [12]). Object-oriented methods have acquired an Ada-oriented flavor because of the efforts by Booch [8]. MachineChart is a Ada-based graphical notation for designing real-time systems. This thesis uses it as a generic architecture design notation.

2.3 Petri Nets Oriented Approach

The Petri Nets oriented approach is to use Petri Nets as an analysis modeling and design modeling tool to help designing system architecture and software modules.

Petri Nets [43] [41] are a graphical and mathematical modelling tool which provide a formal method for specifying the behaviour of concurrent systems. They consist of three fundamental concepts: *tokens*, *places*, and *transitions*. Tokens represent instances of concurrent behaviour that flow through a network of places and transitions.

A Petri Net is a particular kind of directed, weighted, bipartite graph consisting of two kind of nodes, called *places* (drawn as *circles*) and *transitions* (drawn as *boxes*), where *arcs* are either from a place to a transition or from a transition to a place. A *marking (state)* assigns each place a nonnegative integer. If a marking assigns place p a nonnegative integer k , we say that p is marked with k tokens. Pictorially, we place k black dots (tokens) in place p . Tokens await processing in places. When all of the places that are incoming to a transition have tokens, then the transition fires, causing one token to be removed from each of the input places, and one token to be placed in each of the output places.

Conditions and events are the concepts used in the modeling technique. Here, places represent conditions, and transitions represent events. A transition (an event) has a certain number of input and output places representing the pre-conditions and post-conditions of

the event, respectively. The presence of a token in a place is interpreted as holding the truth of the condition associated with place.

Conditions (or predicates) [25] can be attached to transitions to further refine the firing mechanism, and time periods representing activities can be associated with the transitions -- these represent variations on the basic Petri Net concept. In graphical representation, a filled box shows a timed transition. The transition condition is the time duration [1]. Another useful variation is the Coloured Petri Net [31] in which tokens are coloured and are therefore distinguishable from one another. Moreover each place and each transition has attached a set of colours. A transition can fire with respect to each of its colours. Coloured tokens are usually employed to represent individual classes of behaviour within the system, thus the transitions can be made to respond differently to different token colours.

A coloured Petri Net model of the layered communications system is shown in Figure 2-2. The upper part of the picture presents the sending side, the lower part presents the receiving side. There is one link layer and physical layer for each physical connection in each node. Therefore, the coloured tokens indicate the packet's transmission route. Number n represents the queue size in network layer, and m represents the sliding window size in link layer. Time period τ shows the time-out duration. The shaded area is the transmission channel.

As can be seen from Figure 2.2 the Petri Net can describe behaviour in a system, but does not actually relate it to the architectural components. In fact, it represents the essence of the behaviour with architectural components removed. However, Petri Nets are amenable to a wide variety of analysis and in particular, a basis for a performance analysis model.

As a specification modeling tool Petri Nets can precisely describe a system's behaviour, as a design tool it can analyze the design problems, and as a performance evaluation tool it can evaluate the performance of the system [39] [11]. PROT nets [11] as a prototyping tool can even automatically translate the Petri Nets into Ada program structures. There are also some performance analytic models which combines Queueing networks or Markov models with Petri Nets to analysis the software synchronization phenomena [4] [1]. However, the complexity involved in modelling a large real-time system

is substantial. Smith suggests to use such detailed models at the later stages of the software development lifecycle.

2.4 Timethreads Oriented Approach

“Timethreads” is a visual notation for describing system behaviour scenarios, which are used to do requirements analysis and preliminary design. During the system specification phase, timethreads can be used for component discovery and behaviour reasoning. During the design phase, timethreads can be used to design control flow patterns, to verify overall behaviour, and to develop component interfaces. The timethreads-oriented approach provides a seamless technique for describing both pure behaviour and behaviour linked to components [16]. Sections 2.4.1 to 2.4.3 introduce the basic concept of Timethreads notation, the MachineChart notation and Activity diagram notation. These concepts are

the foundation for the following chapters.

2.4.1 Timethreads Notation

The timethreads method is a notation for representing the sequences and coordination of activities in a system. Competing with formalisms like Petri Nets and Data flow diagrams, timethreads differ chiefly in that they add sequence and coordination to architectural diagrams. Otherwise, it is generally possible to describe timethreads using Petri Nets. In essence, timethreads illustrate the tracks of tokens flowing over the Petri-Nets.

A basic timethread is shown in Figure 2-3. Each timethread has a *thread start*, *path*, and a *thread end*. A timethread shows the sequence of activities along the path, where the thread begins at the thread start, and terminates at the thread end. The activities which occur along the path are simply annotated. When overlaid on an architectural diagram (such as a MachineChart diagram), the components¹ of the diagram give a context for the timethread's behaviour. For example, if a timethread starts in Component A, then the timethread is initiated from Component A; if the timethread's path crosses over Component B, then the timethread carries out some activity in Component B; if the thread end appears in Component C, then the timethread terminates in Component C. A timethread example for a layered communications system is shown in Figure 2-4.

1. A component is represented in a black box, which presents programming constructs such as modules, subsystems, or layers. The detailed descriptions follow in section 2.4.3.

Figure 2-4: Major Timethreads for a Layered Communications System

In this system, there are six architectural components shown, two instances of the three layers of the communication system, each with three components (N -- network layer, L -- link layer, P -- physical layer). Four timethreads are overlaid on the architectural components, and they collectively illustrate the behaviour that occurs when a message is transmitted from one of the systems (SYSTEM 1, in this case). Timethread **black** shows the behaviour initiated from outside of the system, passing through the network layer, and into the link layer where it initiates timethread **dark grey**, waits for a synchronization from timethread **medium grey** at a *waiting place* (a large dot), and then terminates.

Timethread **dark grey** represents the transmission of the message (**Data**) from the link layer, via the physical layer, over the media to the physical layer of the target system

(SYSTEM 2, in this case), where it terminates and simultaneously initiates timethread **medium grey**, to handle processing in the target system. Assuming that the message is passed error-free, timethread **light grey** is initiated to deliver the message to the network layer, and meanwhile timethread **medium grey** generates an acknowledgment in the link layer, sends it through the physical layer, over the media, and into the physical layer of the originating system. Once in the originating system the timethread **medium grey** continues into the link layer where it signals the waiting place on timethread **black**, and terminates. Wherever a timethread path crosses over an architectural entity, there may be potential activities. Labels **a**, **b**, **c**, and **d** in Figure 2-4 represent activities. These labeled activities may be characterized by specific parameters such as time, memory, algorithms, et cetera. Timethreads do not suggest any limitations.

A subset of the timethreads notation is provided in appendix A.1. The complete notation can be found in [16] [17] [14]. The different timethread constructs can be combined as necessary to achieve desired semantics. For example, the And-Join and the And-Fork are unified to form an “And-Join-And-Fork” which initiates several outgoing timethreads concurrently, only after all of the incoming timethreads have terminated.

Timethreads express the sequencing of activities in relation to the functional components. A system with logical concurrence may have multiple threads proceeding through it simultaneously that may interact and compete for limited resources. However, in the system specification phase, timethreads are used express end-to-end behavior patterns spanning the system as a whole, or many of its components, in a way that does not require early commitment to architectural details [14].

2.4.2 Activity Diagram

An Activity Diagram [56] is another notation for describing behaviour and is closely related to Petri Nets in concept. It differs markedly in that it describes many behavioural operations more succinctly than the Petri Net (that often is quite verbose for even simple behaviours) and places greater emphasis on the explicit representation of activities. The main purpose of the activity diagram is to provide a framework for a performance model, possibly implemented by a Petri Net or a queueing network model.

Activity Diagrams describe the ordering and causality of the system activities. They can express not only the partial order of the activities, but also the concurrency and synchronization relationships among the activities. The Activity Diagrams only model paths that are key to the performance. The diagrams are the extension of the performance task graph [42] [21] [51] [48], and are closely related to the design model. An example activity diagram is depicted in Figure 2-5. This diagram represents the behaviour of the layered communications system, at the same level of detail as illustrated by the timethread model in Figure 2-4.

The activity boxes represented in Figure 2-5 are:

Box a: send packets and wait for acknowledgment.

Box b: transmit data through the channel.

Box c: receive data and send acknowledgment back.

Box d: sent packets to its network layer.

When a packet arrives at the **event ready** point in SYSTEM 1, **box a** may accept this packet after a period of time. It then sends the packet to **box b** and waits for the acknowledgment at the **waiting place** (large black dot). After **box b** finishes its work, the data packet is ready for **box c** at the **event ready** point in SYSTEM 2. Then, **box c**

processes the data packet and starts box d, and meanwhile synchronizes with the waiting event. The packet transmission completes after box d finishing its activity at the event end point in SYSTEM 2. The acknowledgment completes at the event end point in SYSTEM 1.

The complete Activity Diagram notation is summarized in appendix A.3. The Activity Diagrams use *Race line* and *Inhibitor arcs* to express the competition of the execution path. An inhibitor arc *Disables* flow along an event path if it is “unfilled”. If it is “filled” then events may flow along the path. A transition with inhibitor arcs is enabled if there is no unfilled circle and all input places are active. The annotations on activity boxes identify the textual descriptions of activities that are symbolized by activity boxes. As with data flow diagrams, the activity boxes do not necessarily represent architectural components.

Timethreads diagrams and Activity diagrams are, to a large extent, equivalent, but they describe the system requirements and behaviors from different view points. The two diagrams have different perspectives; i.e., the timethreads diagram is from the functional analysis perspective and the Activity diagram is from the performance analysis perspective. The evolution of the Activity diagram continually reconciles the evolution of the timethreads diagram. This consistency ease: the forwards transition and backwards tracing between the analysis model and design model.

Activity diagrams, similar to the execution graph given by Smith [48], are used to model concurrent software. In the early stage of software development, a simple model is constructed to capture essential software performance behaviour. Then the performance parameters can be derived by the model with insufficient data using performance analysis equation and queueing network techniques [48] [35] [40]. Interactively, this provides initial feedback on whether or not the planned software is likely to meet performance goals.

The resource contention delays and some execution overheads may not be considered in this early stage. Later, when design progresses, more accurate performance data can be extract from the model. Quantifiable performance targets provide the basis for selecting appropriate design alternatives. Moreover, design decisions affect performance, thus the performance model should completely reflect the design; i.e., any changes in design must be reflected on performance model.

The important execution paths are essential to understanding and improving performance [48]. The response time and throughput of the execution path are two important parameters. Path response time is the time from an invocation of the path until the completion of its execution. It normally contains the execution times for activities along a path, delays due to resource contention (which include hardware resources such as CPU, I/O, etc., and software resources such as data files, semaphores, mailboxes, etc.), and overhead introduced by interprocess communications and context switches. These factors are affected by the module precedence relationships, module synchronizations, task assignments, resource availability and utilization, and processor scheduling policies. However, at the software analysis stage, many of the factors are hidden along the system execution path and will be affected by later design decisions. Hence it is very important to analyze the system performance of the design.

2.4.3 MachineChart Notation

MachineCharts [12] is an architectural design notation. A configuration of operational components is called an *Architecture*. MachineCharts express modularity, concurrency, waiting, sequencing, and so forth. They have the ability to express inter-object connections in a way that captures the essential temporal semantics of practical communications, active messengers, mailboxes, or semaphores where the representations of these communications mechanisms are uncommitted to any particular implementation technologies [19]. The basic MachineChart notation is defined in appendix A.2, and a complete notation can be found in [12].

An architecture diagram of a layered communications system is given in Figure 2-6, which shows a system of two nodes and one channel. Each node has three components -- N (network layer), L (link layer), and P (physical layer). The channel is also an component. The components interact through the *interface buttons*. *Data flows* represent the packets passing across the components. Figure 2-6 is a very high level architecture diagram for a layered communications system, corresponding to the high level timethreads pattern in Figure 2-4. A timethread that overlays an architectural diagram, identifies components that have responsibilities along the path. The act of drawing a timethread across components in a diagram like Figure 2-6 amounts to assigning responsibilities to the components for

converting causes into effects [16].

The architecture diagram shows how the components operate together as a whole to accomplish the overall purposes of the system. A component is like a black box hiding many details. It is good to start the transformation from specification to design at this point. When timethreads are factored, the detailed structure of the components will become apparent.

There are three concepts used in timethread-with-architecture design, that is *teams*, *carriers*, and *workers* [16]. A team is composed of carriers and workers. In relation to timethreads, its responsibility is to work like a team to advance a segment of a large-span timethread. As showed in Figure 2-6, component L is a team that carries a packet through the link layer to the physical layer. A carrier is an active entity responsible for carrying messages in a team. A worker is an entity responsible for doing detailed work in a team. The use of carriers and workers is hidden in Figure 2-6. Chapter 3 will illustrate these entities.

2.5 MLog Programming Language

MLog [32] [33] is a multitasking, object-oriented Prolog for which the primary application is prototyping embedded, concurrent systems. MLog combines three programming paradigms: (1) *object-oriented programming* (OOP) in a manner very similar to Smalltalk [26], (2) *concurrent programming* using rendezvous and asynchronous message-passing mechanisms for interprocess communications, and (3) *logic programming* based on the computational model of Prolog. The MLog environment provides tools for debugging and exploring prototype behavior [33].

2.5.1 OOPS in MLog

MLog supports the OOPS concepts of class hierarchies, such as object, class, and meta-class. Inheritance is the relationship among the classes. An object has state, behaviour, and identity; the structure and behaviour of similar objects are defined in their common class [10]. An instance of a class is an object, which is encapsulated as data and operational procedures on data. Additionally, MLog distinguishes objects as passive objects, which are sequential and for which there is no protection for mutual exclusion, and active objects, which are sequential and autonomous. MLog active objects represent large-grain communicating sequential processes. They are drawn as an actor in a MachineChart diagram. The basic execution units in MLog are objects, which can be created using the *new* method with a private copy of the declared instance variables and the methods of the class. The new method also determines the nature of the object, active or passive. When any active object is created, it automatically sends itself a *begin/0* message; by default, this will cause the object to continuously accept a message and process it. Once an active object accepts a message, it queues all the other incoming messages while the message is processed. The other incoming messages are stored in the object's message queue. The active object gets the next message from its message queue after finishing the previous message. If there is no message in the queue, it waits. Active objects have mutually exclusive access to their instance variables. Passive objects will execute only upon receipt of a message, and will not impose mutual exclusion on sender objects.

Code reusability is derived from the class notion combined with inheritance. MLog

uses a single inheritance scheme. A class inherits all of the methods and instance variable declarations of its superclass. A subclass may override the methods of a superclass, and add new methods, and provide new instance variable declarations. MLog inheritance when combined with its rendezvous mechanism provides a good facility for the *nested rendezvous*. This is discussed in chapter 4.

Unlike Smalltalk, elements in MLog such as symbols, numbers, and lists are not treated as objects, and MLog does not support class variables.

2.5.2 Concurrency in MLog

The creation of objects and the communication between objects supports concurrency in MLog. The semantics of message-passing depends on the type of target object. If the target is a passive object, sending a message is equivalent to a procedure call. If the target is an active object, then a rendezvous takes place. In this case, both the sender and the receiver must agree to communicate. If the target chooses not to communicate, then the sender has to wait for the rendezvous happens. Multiple messages from different senders to a given target are queued FIFO (First In First Out). A secondary form of interprocess communications is asynchronous message-passing in which the sender is not blocked.

MLog uses a message counter to simulate a clock rather than using a physical mechanism. That is, each time a message is passed to an active or passive object, the counter is incremented to represent another “clock tick”. Each build-in method counts for one tick. Users define how many ticks constitute one second in MLog. The smallest time resolution provided by MLog is one second corresponding to one clock tick. This mechanism poses difficulty for performing simulation since there is no way to get accurate times. The performance experiments based on the case study is limited to the examination of the execution overhead.

MLog contains two logical queues: (1) the *application ready-to-run queue* for active objects which belong to user’s application, and (2) the *system ready-to-run queue* for active system objects in MLog. The MLog virtual clock only counts for the execution of active objects in the application queue; the system active objects are ignored.

2.5.3 Prolog in MLog

Computation in MLog is based on the execution paradigm of conventional Prolog, where the methods are actually Prolog procedures, and message passing in MLog equates to solving goals in Prolog. If a failure occurs within a method body, then normal backtracking will take place, including exploration of alternate method clauses. If no method clause can succeed then the goal evaluation fails.

MLog's backtrackable communications mechanism allows the message receiver to choose the "right" message using whatever information is present in the message and in the state of execution of the target object. In this way, the programmer simply has to declare the proper conditions for communication and the MLog search mechanism does the work.

Prolog programs have precise operational meaning. They are the logic consequences of the axioms [50]. The MLog computation model inherited from Prolog and its IPC declarative representation make the language powerful to represent a complex things in an abstract way. This power will be applied in Chapter 4 for executable timethreads.

All reference to variables except instance variables refer to the Prolog variables.

2.5.4 Prototyping Environment in MLog

The MLog prototyping language can be used to study the behavior of the systems to determine: (1) the validity of the design, (2) errors due to non-determinism, and (3) the appropriateness of modularization decisions. Also, rapid design prototyping encourages the exploration of design alternatives, which generally leads to a better final design choice [33].

The MLog development environment provides a framework for constructing the wide variety of tools and software to help in design prototyping. The DMLog debugger tool provides a three-level debugging model: interprocess communication level; class and object level; and method level. It addresses the concurrency, object-oriented, and logic programming aspects of an application. The ProtoTool is an operational prototyping environment that gives software developers execution control and feedback during the evaluation of concurrent system designs. The TimeLine Tool is a *time line* (execution thread) display tool that graphically illustrates the temporal order of concurrent system

events. The time line display provides instant feedback to developers. Using ProtoTool, the application can be stopped, its time lines can be examined, and any necessary program or environment changes can be made. In the time line: (1) the vertical arrows denote active object creation and destruction, (2) the thin horizontal line shows an active object that is ready-to-run, (3) the thick horizontal line shows an active object is running, (4) the thin dashed line shows an active object blocked waiting for a message to be received, and (5) a thin vertical line below an active object's activity time-line is used to show that other active objects are waiting to rendezvous with it. The graphical display is shown in Appendix A4.

Because of its features and availability, MLog is employed in this thesis to prototype the layered communication system, to implement the executable timethreads, to control the application's execution thread, and to provide an experimental environment to examine the design.

3

Tracking Design Changes Using Timethreads

This chapter presents the maintenance of concordance between performance and design by associating the activities in an activity diagram with an equivalent set of timethreads. The timethreads are used as a behavior reasoning and components discovering tool to help the design process, and in the end, it shows how the activities have been mapped to a design model (MachineCharts).

3.1 Behavior Reasoning with Timethreads

Using timethreads as a system behavior reasoning tool, the process starts from discovering end-to-end timethreads, then factors end-to-end timethreads into more detailed small-span timethreads, and then combines those smaller-span timethreads into large-span timethreads to represent the whole system behaviour.

Timethreads can be refined by substituting them with more detailed timethreads that convey conforming semantics. Specifically, any timethread represents a sequence of activities occurring in an architecture, thus any activity which is known to terminate can be replaced by another set of activities performing the function that are also known to terminate. Any other semantics, such as waiting places must also be preserved.

As an example, consider the simple timethreads model of the layered communications system in Figure 2-4. There are three timethreads organized by sequencing, spawning and triggering at a waiting place. A possible refinement of this system is shown in Figure 3-1,

where the **black** and **medium grey** timethreads have been decomposed into smaller timethreads. These timethreads are conforming as follows:

medium grey -- the first new timethread, **slotted black** is initiated from the arrival of a message at the physical layer, as before. It transfers this message to the link layer for processing, at which it spawns two timethreads at an AND-fork, and then itself terminates. The timethread, **light grey**, takes the packet and transfers it through the network layer to the end user -- this is unchanged from the earlier design. The second timethread, (new) **medium grey**, carries the acknowledgment back across the physical layer, through the channel, and into the sender side link layer. Thus, **slotted black** and new **medium grey**, when combined, exhibit the same behaviour as the original **medium grey** timethread; the

new timethreads constitute a simple refinement of the original timethread.

Black -- six smaller timethreads are refined: timethreads labelled 0 and 1 are the original timethread entering the network layer, getting a packet and passing this to the link layer for processing; timethreads 2 and 3/4 (3/4 is one timethread) are link layer processing, timethread 3/4 spawns the physical transmission timethread (**dark grey**), and waits for acknowledgment; timethreads 8 and 9 define new or more detailed activities that were not expressed in the original timethread -- they do not contradict it, they just add detail as follows:

timethread 7, triggered by black timethread 4, in conjunction with the acknowledgment thread (**medium grey**) represents a conditional timed wait, in which if the acknowledgement does not reach the link after some time period, then a timeout occurs. The original black timethread simply did not address this case as it considered only the case where the timeout did not occur.

timethread 8 represents the re-transmission of a timed-out frame. It did not appear on the original black timethread as the time-out case did not exist. It is connected by an Or-Join with the original timethread 2, so that a frame is sent if it is new, or if it is a re-transmission.

timethread 9 supports flow control of packet processing by the link layer. It was not present in the original black timethread as flow control was not considered. In this case, a successfully transmitted frame, should allow the processing of another packet; this is defined by the condition attached to the Or-Join with timethread 1 -- if there is room for a packet to be send, timethread 1 is chosen (and causes timethread 2 to start), otherwise it is not selected -- if there is no room and timethread 9 arrives at the Or-Join, then the waiting packet is processed (timethread 2 is started), otherwise timethread 9 terminates.

Timethreads in Figure 3-1 demonstrate the end-to-end behaviors of layered communications system spanning over three layer components. At this point, each end user only represents one role (either as a sender or a receiver).

3.2 Tracking Changes in Activity Diagram

The activity diagram in Figure 2-5 corresponds to the timethreads in Figure 2-4. After refining the timethreads shown in Figure 3-1, the activity diagram should illustrate the changes as well. The activity diagram shows the activities and their coordination, but does not place the activities in an architectural context; the timethreads model provides this, as well as describing the coordination.

Timethreads factoring actually corresponds to activities decomposition. Figure 3-2 is the refinement of Figure 2-5, which corresponds to the refinements of timethreads for the layered communication system shown in Figure 3-1. The labels on the timethreads are the activity boxes in Figure 3-2. The actual activities are defined below:

activity 0: transfer an outgoing message to the network layer for processing.

activity 1: look up the route, create a packet, and transfer the packet to the link layer

activity 2: create outgoing data frames from an input packet.

activity 3: get a data frame for transmission (set sequence bits).

activity 4: set up the timer for re-transmission time-out.

activity 5: transmit the data frame over the physical media.

activity 6: receive a frame (an acknowledgment in this case) from the physical media.

activity 7: time out.

activity 8: invoke a retransmission.

activity 9: prepare to sent the next frame.

activity R0: assemble the frame at the receiver side.

activity R1: transfer an incoming frame to the link layer for processing.

activity R2: transfer an incoming packet to the Network layer, de-packetize it, and transfer to the host.

activity R3: create an acknowledgment frame and send over the physical media.

Comparing Figure 2-5 with Figure 3-2: activity a in Figure 2-5 is decomposed into four activities 0, 1, 2, and 3 in Figure 3-2; activity c is decomposed into three activities R0, R1, and R3; activity 5 is equivalent to activity b and activity R2 to activity d; activities 4, 6, 7, 8, and 9 are all represented by a large black dot (waiting place) in Figure 2-5 since there was no timeout mechanism introduced before.

Figure 3-2 in a sense is closer to the Petri Nets diagram shown in Figure 2-2. It represents the system behaviours and explicitly shows the ordering and causality of the activities. This provides a good model for performance evaluation based on design. Using the activities precedence relationships in Figure 3-2, performance analysis techniques can approximate the path response time and frequency.

3.3 Designing with Timethreads

The transformation from an analysis model to a design model is often a complex task. However, timethreads can be a very good tool to help walk through this process. Buhr summarizes the design processes with timethreads [16]: discover end-to-end timethreads and components (skeleton architecture); factor end-to-end timethreads and compound components; map timethreads to components in order to decide local responsibilities, overall controls, and collaborations; and design component logic. The previous sections have discussed using timethreads to discover end-to-end system behavior patterns and components. This section discusses how to propagate timethreads into a design process and how to decompose the compound components to map the timethreads so that the components can share the responsibilities for the timethreads execution.

3.3.1 Timethreads Design Reasoning

A further refinement of the timethread model overlays the acknowledgment and data reception activities into a single system (which is the real case). The simpler diagrams given so far (in Figure 2-4 and Figure 3-1) show these behaviours split over two systems, where actually both behaviors exist in each system. This refinement is shown in Figure 3-3, and includes the following changes (with respect to a transmitting system):

- Timethread 15 carries data or acknowledgments and terminates in the physical layer; thus, the original timethread 6 was decomposed into a sequence of timethreads 6, 10, and 15.

- Timethread 10 transfers the incoming frame to the link layer where it Or-Forks into timethreads 6 and 11; timethread 6 represents the acknowledgment frame and timethread 11 represents the incoming data. Thus, the original R1 timethread is now decomposed into timethreads 10 and 11 in sequence.

- Timethread 11 reaches the incoming data And-Fork that creates timethreads 12 and 13. Timethread 12 represents the outgoing acknowledgment frame and transfers it to the physical layer; in the physical layer timethread 14 from the Or-Join, transfers the frames over the physical media. In this case, the original R3 timethread is decomposed into

timethreads 12 and 14.

-- Timethread 13 converts the data frame to a packet and transfers it to the network layer. Finally timethread 16 depacketizes the message and transfers it to the host. Thus, timethread R2 has been decomposed into two timethreads 13 and 16.

Figure 3-3: Refinement #1 Overlaid of Timethreads for Layered communications System

At this point the refinements to the timethread model are complete, however the activities from the activity diagram are no longer correlated with the activities on the timethread model. The performance view of the original activity diagram has been refined

in the new timethread model; in some cases activities have been only decomposed, but in some cases they have been decomposed and factored into common activities. The changes are summarized below:

activity R0: simply renamed to 15.

activity R1: decomposed into 10 and 11, where 10 is a factored activity.

activity 6: decomposed into 15, 10, and 6, where 15 and 10 are factored activities.

activity R3: decomposed into 12 and 14, where 14 is a factored activity.

activity 5: decomposed into 5 and 14, where 14 is a factored activity.

activity R2: decomposed into 13 and 16.

The new (or refined) activities are described as follows:

activity 5: transfer the frame to the physical layer for transmission.

activity 6: process an acknowledgment frame that was received from the physical layer.

activity 10: transfer an (acknowledgment or data) frame from the physical layer to the link layer.

activity 11: process a data frame that was received from the physical layer.

activity 12: transfer an acknowledgment frame to the physical layer for transmission.

activity 13: transfer an incoming packet from the link layer to the network layer.

activity 14: transfer a frame over the physical media.

activity 15: assemble an incoming frame from the physical media.

activity 16: depacketize incoming packet and transfer message to the host.

Figure 3-4: Refinement of Activity Diagram for the Layered Communications System

Now, the performance modeler could change the activity diagram shown in Figure 3-4 so that it is again in one-to-one correspondence with the timethread model, or simply keep track of how the changes will affect the interpretation of the performance results, or how measurements from a prototype will need to be interpreted in the context of the performance model.

3.3.2 Components Discovery

The next stage of design is refining the architectural components by mapping the timethreads and activities to actors. Once this stage is complete, actor interconnections are added which represent those portions of timethreads that reside between actors. MachineCharts [12] is used here as a design visual notation that gives behavioral cues to help in understanding how components are intended to operate and cooperate. The actor allocation is illustrated in Figure 3-5. The mapping process has resulted in the following assignments:

- activities 0 and 16 are allocated to the network layer task;

- activity 1 is allocated to the network-to-link layer transporter actor;

- activity 13 is allocated to the link-to-network layer transporter actor;

- activities 2, 3, 4, 5, 7, 8, and 9 are allocated to the TX actor in the link layer;

- activities 6, 11 and the Or-Fork for incoming frames are allocated to the LinkMgr actor;

- activities 14 and the Or-Join are allocated to actor 14 in the physical layer for sending frame to the channel;

- activity 15 is allocated to actor 15 in the physical layer for receiving frame from the channel.

- activity 12 and the And-Fork for data frame processing are allocated to the RX actor in the link layer.

As can be seen, timethreads 1, 10, and 13 are the threads responsible for carrying messages from one component to another. Thus, three carrier actors 1, 10, and 13 are assigned. Actors TX, RX, and LinkMgr are workers which team up together with carrier 13 to perform the functions of the link layer. Actor 0/16 is a worker (a network layer engine) which combine carrier 1 to carry out the functions of the network layer. Actors 14 and 15 are also workers together with carrier 10 to achieve the functions of the physical layer.

3.3.3 Design Component Collaborations

The next design stage is to add the interconnections. The resulting design is shown in Figure 3-6. The various fingers, gates, and waiting conditions are added to enforce the timethread semantics as follows:

The Transporter actor for activity 1 blocks when there is no outgoing packet from the network layer UserMgr actor; this is analogous to waiting for the trigger from the preceding

timethread.

The Transporter actor for activity 1 blocks at the TX actor in the link layer when there is no room for incoming packets and thus satisfies the Or-Join of activities 1, 2, and 9. Activity 9 controls the blocking of the Transporter actor by altering the waiting condition.

The Transporter actor for activity 13 blocks at the RX actor in the link layer when there are no outgoing packets to the network layer; this corresponds to the timethread waiting at the And-Fork.

The LinkMgr actor directly invokes either the TX or RX actors whenever a frame arrives. As each frame is either an acknowledgment or a data frame, each timethread of the Or-Fork is accommodated by exclusive-or treatment by the LinkMgr; furthermore, incoming frames cannot trigger the Or-Fork unless the previous frame has been handled; messages to the LinkMgr actor are queued while it is engaged in other processing.

The TX and RX actors start the timethreads of activities 5 and 12, respectively, when outgoing frames are present. These timethreads invoke a common gate in PhyDown actor of the physical layer to implement the Or-Join.

The Transporter actor for activity 10 is invoked when the PhyUp actor in physical layer receives an incoming frame. The Transporter waits at the gate of PhyUp actor until a frame arrives, thus it correctly implements the start of its timethread. When the LinkMgr actor is ready to process another incoming frame, it accepts the message from the Transporter actor, and then invokes one of the two outgoing timethreads of its Or-Fork. The detailed design of the TX actor is presented in Figure 3-7. Timethreads arrive at TX via gates: one representing incoming packets from the network layer and the other representing incoming acknowledgment frames that have been sorted by the LinkMgr actor. These timethreads are serialized by the TX engine through the pull on the gates and processed. All timethreads that arise from an incoming packet are processed in sequence: (a) the packet is converted into (one or more) data frames for transmission (activity 2 -- this occurs in a gate); (b) a data frame is selected for transmission and a sequence bit is set (activity 3 -- this occurs in a button "sendData"); (c) the data frame is transferred to the physical layer for transmission (activity 5 -- this is invoked from the button "sendData"); (d) the timer on

transmission is enabled (activity 4 -- this occurs in the “timer object”, which may be realized in a variety of ways in actual implementations); (e) the acknowledgment (activity 6 -- which invokes a gate) or timeout (activity 7 -- caused by the “timer object”) is processed; and (f) a retransmission occurs (activity 8 -- in the actor engine) or new transmission is started (activity 9 -- handled in a gate). With this mapping of activities to design objects, the performance model can locate the activities in the detailed design space.

Figure 3-7: Detailed Design of Task TX

The timer object can be represented as an actor (named tickTask) as well. The explicit relationship between tickTask and TX is shown in Figure 3-8. In this implementation, the TX engine controls three gates: putPacket, putAckFrame, and timeOut.

When activity 1 invokes gate putPacket, the engine will pull the gate and check if there is room in the buffer for the packet. If there is room, activity 2 is performed, and if TX is not at “ackWait” status, the new transmission will start, that is, activity 3 and 5 will execute. Meanwhile, activity 4 will execute as well, which invokes gate startTimer of TickTask. The TX status is set to “ackWait”.

Figure 3-8: MachineChart of Tx and tickTask

When activity 6 invokes gate `putAckFrame`, the engine will pull the gate and check the sequence bit. If it is correct, activity 9 is performed, which invokes the gate `stopTimer` of `TickTask` and sets `TX` status to “establish”, indicating that it is ready for a new transmission. At this point, if there is a data frame in the buffer waiting for transmission, the new transmission will start.

When a timeout occurs, activity 7 invokes the gate `timeOut`. The `TX` engine will pull the gate and retransmit the data frame; thus, activities 3, 5 and 4 will execute again.

3.3.4 Controlthreads Reasoning

Controlthreads [17] represent the final refinement of the timethread model as they define the explicit changes of control activities in a design object, where the controlthread is limited to the execution scope of a single object; i.e., a controlthread is equivalent to a “thread of control”. The documentation of a controlthread is the same as a timethread with the exception of the different starting place annotation, as shown in Figure 3-9. As can be seen from Figure 3-9, recording a controlthread with the conventional timethread notation can lead to a cluttered diagram, thus a timeline notation [17] is used instead to state the same information, one timeline for each design object. This displays the time-driven interactions among the objects, at the expense of not showing physical interconnections. The start of the

controlthread is shown at the top of the timeline diagram, and each transfer of control between design objects is shown by vertical movement between the lines representing each of the design objects. Horizontal movements along a timethread show the execution of activities.

Figure 3-9: Controlthread Notation

The actual controlthread for the TX actor is depicted in Figure 3-10. The activities associated with an actor are either associated with specific design objects (such as the gates, button, or timer) or with segments of the engine's timeline. Synchronization with external controlthreads is shown by the annotated solid circles with intersecting lines -- those with small arrows leading to a timeline define triggering controlthreads, and those that touch timelines are triggered controlthreads. The controlthreads cannot express parallelism (with the exception of timers) as they define a single thread of control, thus different scenarios of execution must be shown explicitly. For example, the timeline of Figure 3-10 shows two scenarios: a timeout occurs, followed by a retransmission, and then a successful acknowledgment received. For the performance modeler, the controlthreads show the exact start and end of activities by allowing the designer to identify when control is transferred

to and from an activity.

3.4 Module Design and Activities Mapping

By completing the timethread development process (detailed design and controlthreads) for the other tasks, the activities from the original activity diagram of Figure 3-4 can be mapped to the design objects, including their invocation and completion points within the design. This is shown in Table 3-1 using the remaining detailed design shown in Figure 3-11.

Activity	Design Object	Start	Stop
0	UserMgr	Host calls gate User.txTo to sent packets	messages are stored
1	userTransporter	triggers gate User.getFor to get a packet	carries the packet to L, returns from TX.putPacket
2	TX	Tx engine accepts the packet from gate TX.putPacket, if possible	stores the packet
3	TX	Tx engine takes a data frame, sets the sequence bit	triggers activities 4 and 5
4	TX	TX engine triggers gate tickTask.startTimer	waiting
5	TX	TX engine sends frame to P through gate rxFromLink	returns from PhyDown.send-Down
6	LinkMgr	checks frame, and calls TX.putAckFrame	call accepted on TX.putAckFrame
7	tickTask	timeout occurs	triggers gate TX.timeOut
8	TX	TX engine receives timeout message from TX.timeOut	ready for retransmission
9	TX	TX engine receives Ack from TX.putAckFrame	ready for new transmission
10	frameTransporter	triggers gate PhyUp.read to get a frame	carries the frame to L, returns from LinkMgr.putFrame
11	LinkMgr	checks frame, and calls RX.putDataFrame	call accepted on RX.putDataFrame
12	RX	sends an Ack to P through gate rxFromLink	returns from PhyDown.send-Down
13	packetTransporter	triggers gate RX.getPacket to get a packet	carries the packet to N, ends at UserMgr.putFrom
14	PhyDown	receives frame from gate PhyDown.seadDown	sends to channel
15	PhyUp	receives frame from channel	stores frame through gate PhyUp.write

Table 3-1: Mapping between activities and modules

Activity	Design Object	Start	Stop
16	UserMgr	receives a packet from gate UserMgr.putFrom	sends to Host

Table 3-1: Mapping between activities and modules

Using this table, a performance engineer can now instrument a prototype for performance experiments and then correlate the results with the performance model based on the activity diagram. The experiments, based on paths through the activity diagram, can be mapped to specific points on the design for data injection and measurement probes. Thus, the timethread approach has provided a mechanism (albeit manual) for linking the performance model with the design model.

For example, the average waiting time that a packet spends between arriving at the buffer of the network layer and getting into the buffer of the link layer is the time between the end of activity 0 and the end of activity 2. This can be calculated by inserting a probe at these two points of the program. The performance model and design model are connected in such a simple way which eases the performance testing.

4

Executable Timethreads

This chapter introduces the technique of using a programming model of executable timethreads to control an application (design) model, and thus mechanically recording the concordance through the formality of a programming language (MLog). Section 4.1 discusses the environment and semantics of the MLog prototyping language. Section 4.2 introduces the main idea of executable timethreads. The executable timethreads programming model and control model are demonstrated in sections 4.3 to 4.5.

4.1 Programming Environment for Executable Timethreads

Prototyping executable timethreads is part of a process of prototyping executable specification and design. It provides a means of increasing the probability that the output of the specification phase are the requirements desired by users and that the requirements are feasible. It also ensures that the design correctly implements the requirements and is sound, so that the output of the coding phase will faithfully reproduce the system's requirements and design [55]. Moreover, it ensures testability of the performance requirements and traceability of the performance requirements to the design [2].

MLog is used in the thesis as a programming environment since it has an efficient declarative computation model, a highly abstract structure, and the ability to express concurrency. MLog can express timethreads in a textual way and the relationships and constraints among timethreads can be defined explicitly. The activities along a timethread can be presented as operational methods (which can be viewed as Prolog procedures, or OOPS methods). Each timethread can be created as an active object. The timethreads can

communicate with other objects in the system, and compete for system resources.

MLog supports a declarative inter-process communication mechanism. Synchronous communication in MLog uses the *rendezvous*. In a rendezvous, the sender object must send a message and wait until it is processed by a target object; the target object must decide when to accept messages from a sender, and then given that a message can be accepted, the target will process the message. A first in, first out queue of messages is used to buffer incoming messages that cannot be processed by a target object.

When an object wishes to send a message, it specifies the message pattern in either an object identifier followed by a “-” and a message body, or just a message body. In the latter case, the object identifier is assumed to have been *self*, i.e., the object that sent the message. When an object wishes to accept a message, it specifies a message pattern in one of two ways: (1) its method name, and arguments, meaning a specific message, or (2) a single uninstantiated variable, meaning any message.

A rendezvous receive operation is performed when a target object sends a *waitToAccept()* message to itself. Starting with the first message in the message queue, the *waitToAccept()* method applies the two steps of an MLog rendezvous: *eligibility* and *evaluation* [32]. Evaluations occurs only if eligibility is successful and both steps must be successful for a message to be accepted and for *waitToAccept()* to terminate successfully.

To pass the eligibility step, an incoming message must unify with the pattern defined in the *waitToAccept()* argument. Once this unification has succeeded, the method specified in the message is invoked by the target object, and the evaluation step begins.

In the evaluation step, the method head is unified with the result of the eligibility step and the corresponding method body is evaluated. If this is successful, then both active objects resume execution with the unification of arguments resulting from the two rendezvous steps; that is, the sender’s message “succeeds”, the sender continues to the next message with the resulting unifications, the target’s message *waitToAccept()* succeeds with the resulting unifications, and the target continues execution. If a failure occurs within a method body, then normal backtracking will take place, including exploration of alternate

method clauses (as appropriate). If no method clause can succeed then the evaluation step fails.

If either eligibility or evaluation fails, MLog leaves the message in its queue position and tries the next message in the queue. If no message is successful then the target object waits until a new message arrives. Figure 4-2 shows the steps of an MLog rendezvous.

MLog's backtrackable communications mechanism allows the message receiver to choose the "right" message using whatever information is present in the message and in the state of execution of the target object. In this way, the programmer simply has to declare the proper conditions for communication and the MLog search mechanism does the work.

MLog has the flexibility to let application objects select one of two types of ready-to-run queues: the *system queue* and the *application queue*. This helps the performance evaluation for the timethread model. The system queue holds system objects which are ready to run. If the object at the front of the system queue has the same priority as the application object chosen to run, the system object will be selected to execute [32]. System objects cannot be interrupted and do not consume "time" (i.e. the MLog clock does not run while system objects execute). The application queue is used to queue the active application objects that are ready to run. The "time" concept applies to the application processes; i.e.,

times-out can happen to an application process and the MLog clock ticks as application object execute. The distinction between system objects suppressing the clock and application objects enabling the clock, is used when timethreads are controlling the actual prototype; i.e., the timethreads run as system objects and do not introduce time overhead and the prototype objects execute as application objects and cause time to elapse.

4.2 Introduction to Executable Timethreads

Executable timethreads represent a programming mechanism for developing timethread models, studying them, and binding them to design models. This constitutes an alternative approach to the manual technique described in the previous chapter. The generalized model of the programming environment is shown in Figure 4-2 and is based on the MLog programming language [32] [33].

The environment is composed of three types of objects: *timethread objects*, *control objects*, and *application objects*. The timethread objects represent instances of timethreads, thus in principle, each is capable of autonomous behaviour, and communicates with other timethreads via synchronous message passing in order to implement the timethread synchronization mechanisms. The application objects are the actual concurrent entities in the design model; i.e., they are the actors. They can define the actual design, or merely components of a design in progress (e.g., activities), that have yet to be composed into a final design. The control objects exercise control over the message reception of the application objects; i.e., which messages can be accepted and in what order. There is one control object for each application active object. A control object affects the behaviour of an application object through an indirect nested rendezvous that occurs between the application object and its peer control object every time the application object invokes a receive operation.

Timethreads sequence the behaviour of activities in application objects by sending appropriate messages to the corresponding control objects. Multiple timethreads compete to control the application objects, where permitted by the timethread semantics. This is done by messages from different timethreads to the same control object. Where competition exists, messages are processed first-come first-serve. However, at some point, timethreads cannot control the execution of application objects; i.e., in the time out scenario. This is discussed in detail later in sections 4.5 and 5.3.

For the performance modeler, this approach has two main uses. First, if only the executable timethreads are used, and the application objects are either not used, or do not as yet exist, experiments can be conducted on the timethread model to explore interaction among timethreads and to gain a better understanding of the impact of design choices. Second, once the application objects are integrated, the timethreads will explicitly represent and in fact control the execution of activities. In this manner it is easy to track the start of

activities (and possibly the end of activities) for use in instrumentation for experiments. The cost of this control can be substantial, thus it is limited for the moment to those activities that are initiated from a message to an actor (a push). Unavoidably, timethreads that are completely hidden within an actor are not revealed using the suggested implementation. It is possible to control such intra-actor timethreads, but inefficient to do, in general.

4.3 Executable Timethreads Programming Model

As discussed in the previous chapters, “Timethreads” is a visual notation for describing system behaviour scenarios, which are used to do requirements analysis and preliminary design. To implement the timethreads execution in a programming model is to emulate the behaviour of a system at an abstract level. Thus, the requirements can be operationally validated in the early stage and the contention among the timethreads can be explored which help the designer to gain a better understanding of the impact of design choices. This process advocates executable specification.

A timethread represents a part of the system behaviour. A scenario of the behaviour can be viewed as an instance of the timethread. The autonomous execution unit in MLog is an active object. Timethreads are defined as timethread class objects in MLog. Hence, an instance of a timethread is an active object of the timethread class which is created using the MLog *new/3* method. Thus, each timethread object is capable of autonomous behaviour, and communicates with other timethreads via synchronous message passing. MLog can explicitly express the synchronizations between the timethreads. The MLog run-time system provides processor-sharing for all of the active objects. Multiple timethreads objects exist as the concurrent timethreads, thus they compete for and share the system resources.

The activities along the timethread can be discovered through the software development process. The activities are attached to the timethread object as an instance variable list when the timethread object is created. The activities can be decomposed as design progresses. Finally, each activity corresponds to a function or a section of the real program implementation. A timethread can cover a large span of control of the activities or

can be decomposed into a several of small timethreads, where each timethread controls a few activities.

The rest of this section discusses the executable timethreads description in MLog for different kinds of timethread scenarios such as concurrent timethreads, sequences of timethreads, spawning timethreads, And-Join/Fork timethreads, Or-Join/fork timethreads, trigger and waiting on a timer. Note that activities are shown as messages in an MLog method clause rather than in a list as mentioned earlier. The explicit representation as messages is simpler to illustrate the timethread semantics, however it is replaced later with a list and an activity interpreter.

The following is a timethread initialization description in MLog. The function of the program is to create the instance timethread objects **a** and **b**. Suppose the timethread **a** has two activities: **activity1** and **activity2**; the timethread **b** has one activity: **activity3**.

```
class: main of object.
begin :-
    threadA-new(a,active,[]),
    threadB-new(b,active,[]).
```

Figure 4-3 shows the MLog implementation of the timethread **threadA** and **threadB**. The instance object **a** of **threadA** and the instance object **b** of **threadB** are running in the system concurrently.

Figure 4-4 shows the sequences of timethread **a** and timethread **b**. Upon termination of timethread **a**, timethread **b** is initiated. It is implemented by sending a message from object **a** to object **b**.

Figure 4-5 shows the spawning of timethreads. The timethread **b** is spawned as an activity along the path of timethread **a**. The implementation of this case is also through the message passing.

The following program creates the timethread objects **a**, **b** and **c**. Suppose that the timethread **a** has two activities: **activity1** and **activity2**. The timethreads **b** and **c** each have one activity **activity3** and **activity 4**, respectively.

```

class: main of object.
begin :-
    threadA-new(a,active,[]),
  
```

```
threadB-new(b,active,[]),
threadC-new(c,active,[]).
```

Figure 4-6 shows the And-Join timethreads. Timethread **a** and **b** run concurrently and terminate at a point that initiates the timethread **c**. The condition here is that both timethread **a** and **b** have to finish before timethread **c** starts. It is implemented by sending message to timethread **c** after timethread **a** and **b** finish their work, and timethread **c** waits for both messages from **a** and **b** before starting its execution.

Figure 4-7 demonstrates And-Fork timethreads. Upon termination of timethread **a**, timethreads **b** and **c** are initiated. It is implemented by sending messages to timethreads **b** and **c** from timethread **a**.

Figure 4-8 shows the Or-Join timethreads. Upon termination of one of timethreads **a** and **b**, the timethread **c** is initiated. It is implemented by letting timethread **c** receive a message either from timethread **a** or timethread **b**. As long as timethread **c** gets the message from **a** or **b**, it starts its execution.

Figure 4-9 shows the Or-Fork timethreads. Upon termination of timethread **a**, exactly one of timethreads **b** and **c** will be initiated. The termination condition decides whether to invoke timethread **b** or timethread **c**. In this implementation, it assumes that if **activity1** is executed, it then initiates timethread **b**; if **activity2** is executed, it then initiates timethread **c**.

Figure 4-10 shows trigger timethreads. The timethread **b** triggers another timethread **a** that is suspended at a waiting place (a large dark dot) which represents a location along the path where one timethread must synchronize with another timethread. It is implemented by the message passing mechanism.

To illustrate the case of a timer thread, the **main** program creates two active objects, one is timethread **a**, the other one is **timer**.

```
class: main of object.
begin :-
    threadA-new(a, active, []),
    thread-new(timer, active, []).
```

The **timer** thread is initiated as an activity along the path of the timethread **a**. The **timer** thread starts with a timer object and performs the timer thread behaviour when an elapsed time period has passed. The implementation is shown in Figure 4-11. When the Timer starts, it is either stopped by another timethread or times-out (invokes the `timeOut` method).

The following **main** program creates three active objects: the timethread **a** object, the timethread **b** object, and the **timer** object.

```

class: main o. object.
begin :-
    threadA-new(a, active, []),
    threadB-new(b, active, []),
    thread-new(timer, active, []).
  
```

Figure 4-12 shows conditional timed wait timethreads. Timethread **a** triggers **timer** thread and then waits at an Or-Join position for either the termination of **timer** thread, or the termination of timethread **b**. Actually, timethread **b** and **timer** thread compete and the result affects the execution of timethread **a**. If timethread **a** is triggered by the timethread **b**, it will do `activity2`. If timethread **a** is invoked by the **timer** thread, it will do `timeout_handler`.

In the real implementation of executable timethreads, the activities of a timethread are stored in an activity list represented as an instance variable in MLog. The activity list records all the activities and their order. The instance variable name for the activity list is called *activity*. There is another instance variable *numOfActs* that represents the number of activities in a timethread. A thread interpreter gets an element of the activity list and translates it into an operation. The interpreter program is shown below. It provides the operations on the list and interpretes any element of the list as an activity operation. The

mainControl function loops to get all the activities in the list.

```

begin :-                                     % loop through all the threads
    theValueOf(numOfActs,NumOfActs),
    mainControl(0,NumOfActs).

mainControl(Y,Y).
mainControl(Y,Z) :-                         % check the type of activity
    getAt(L,Y),
    getAt(Felmt,0,0,L),
    deleteFrom(Felmt,L,NewL),
    doDecision(Felmt,NewL),
    Index is Y + 1,
    mainControl(Index,Z).

doDecision(do,JobL) :-                      % do method
    getAt(Msg,0,0,JobL),
    deleteFrom(Msg,JobL,LL),
    Msg.

doDecision(send,JobL) :-                   % send method
    getAt(Msg,0,0,JobL),
    deleteFrom(Msg,JobL,LL),
    getAt(Info,0,0,LL),
    Msg-send(Info).

doDecision(receive,JobL) :-                % receive method
    getAt(Msg,0,0,JobL),
    deleteFrom(Msg,JobL,LL),
    waitToAccept(send(Msg)).

deleteFrom(Elmt,[Elmt|T],T).               % remove an element from the
deleteFrom(Elmt,[HIT1],[HIT2]) :-          % activity list
    deleteFrom(Elmt,T1,T2).

getAt(Elmt,Position) :-                   % search for an element
    theValueOf(activity,Activity),
    getAt(Elmt,0,Position,Activity).

getAt([],X,X,[]).
getAt(Elmt,X,X,[Elmt|_]).
getAt(Elmt,X,Y,[HIT]) :-
    NewX is X + 1,
    getAt(Elmt,NewX,Y,T).

```

Basically, there are three kinds of activities: **do**, **send**, and **receive**. The **do** activity is the work that a timethread does along the path. The **send** and **receive** activities are the communication operations among the timethreads. The **do** activity plays the **worker** role

in a timethread; the **send** and **receive** activities play a carrier role.

The Producer-Consumer problem is used in this chapter as an simple example to demonstrate the concept of executable timethread programming model. In the Producer-Consumer problem, a producer generates items that must be consumed by an autonomous consumer. A shared FIFO buffer is used to exchange items. The timethreads diagram for the Producer-Consumer problem is illustrated in Figure 4-13. There are three timethreads: **producerthread** with two activities, **consumerthread** with two activities and **commonthread** with two activities as well. The **producerthread** generates items and stores them into a shared buffer. The **consumerthread** retrieves items from the shared buffer and consumes them. There could be multiple producers and consumers timethreads existing at the same time. They compete for the limited resource “shared buffer”. The **commonthread** waits for **producerthread** or **consumerthread** to trigger it. Then, it performs the “store” function or the “retrieve” function and goes back to the waiting state. This provides the mutually exclusive access to the shared buffer.

Figure 4-13: Producer-Consumer Timethreads

The MLog version of the executable timethread description is shown below. The **producerthread** first generates an item, and then passes this item to the **commonthread**. These two activities are kept in the **producerthread**'s activity list. The **consumerthread** first requests an item to the **commonthread**, and then consumes it. The two activities are kept in the **consumerthread**'s activity list. The **commonthread** is triggered by either **producerthread** or **consumerthread**. If the **commonthread** receives a message from a

producerthread, it performs *store(Item)* function. Otherwise, it does *retrieve(Item)*. The **commonthread** maintains a private buffer.

class: main of object.

begin :-

```
thread-new(pt,active,[activity=[do,produce(_)],send,common,store(_)],
            numOfActs=2),
thread-new(ct,active,[activity=[send,common,retrieve(_)],do,consume(_)],
            numOfActs=2),
thread-new(common,active,[activity=[receive,_],numOfActs=1]).
```

class: thread of object.

ivars: [activity=[],numOfActs,myBuffer=Q/Q].

methods: class.

methods: instance.

begin :-

```
%%%%%%%%%
```

```
% interpreter code shown earlier %
```

```
%%%%%%%%%
```

send(store(_)) :-

```
theValueOf(myBuffer,B),
enqueueDifferenceList(DATA,B,B2),
makeTheValueOf(myBuffer,B2).
```

send(retrieve(_)) :-

```
size(S),S > 0,
theValueOf(myBuffer,B),
dequeueDifferenceList(DATA,B,B2),
makeTheValueOf(myBuffer,B2).
```

size(S) :-

```
theValueOf(myBuffer,B/[]),
listLength(B,S).
```

```
enqueueDifferenceList(X,Q/[X/T],Q/T).
```

```
dequeueDifferenceList(X,[X/T]/T2,T/T2).
```

```
send().
```

```
produce(_).
```

```
consume(_).
```

The main program creates three active timethreads objects; they are **pt**, **ct**, and **common**. The *produce(_)* and *consume(_)* activities are considered as **do** activities. The request to *store(_)* and *retrieve(_)* activities to the **common** are considered as **send** activity, which send a request to the **common** to store or retrieve an item. The **common**

object always performs the receive operation, and then based on the message it receives to do *store()* or *retrieve()*. The detailed implementations of *produce()*, *consume()*, *store()*, and *retrieve()* are not important. The executable timethreads emphasize the coordination of the timethreads; i.e., how the producer timethreads and consumer timethreads interact with the common timethread. The producer timethread and consumer timethread compete for communicating with the common timethread.

Figure 4-14 shows the TimeLine Tool output for the execution of Producer-Consumer timethreads. The graphical notation is explained in Appendix A4 and [32]. Figure 4-14 shows the execution time lines of the four objects in 12 seconds. The **main** object initiates a producerthread - **pt**, a consumerthread - **ct**, and a commonthread - **common**. After that, the **main** object terminates; the **pt**, **ct**, and **common** objects are ready to run. The **pt** object

starts running first to produce an item and then sends a message *store()* to **common** object that waits for the interaction with **common**. The **ct** object gets a chance to run. It sends a message to **common** to retrieve an item and then waits for the interaction with **common**. Finally, the **common** object starts running. It first processes the message *store()* and then releases the **pt** object. Now **pt** object is ready to run. The **common** object continues to process the *retrieve()* message and releases the **ct** object as well. After the **common** object finishes its operations, the **pt** and then **ct** get their fair chance to run again. The executable timethreads model can be used to analyze the system performance characteristics, such as buffer size. For example, if the producer generates the items more than the consumer needs, the buffer will overflow; if the items the producer generated can not meet the consumer's need, the buffer might be frequently empty. These problems can be examined before the final design and the implementation. The execution speed of the timethreads can be simulated by inserting some time delay.

4.4 Proto-Control Model

The proto-control model defines control objects that exercise control over the message reception of the application objects. When control objects and application objects stand alone, this model can be used as a monitoring tool to record the execution sequences. When used with timethreads, the timethreads sequence the behaviour of activities in application objects by sending appropriate messages to the control objects. This can be used as a testing tool or for performance experiments.

The technique used in the proto-control model is based on the class inheritance hierarchy structure and the nested rendezvous mechanism. For each active object, there is a corresponding control object created by redefining the class method *new/3* in its parent class *protoObject*. Figure 4-15 shows the class inheritance hierarchy. Application class is a subclass of *protoObject* class, and the *protoObject* class is a subclass of *object*. Each subclass inherits methods and instance variables from its parent classes. It can also redefine or add methods and instance variables to its class.

The nested rendezvous mechanism is achieved by redefining the instance method *waitToAccept/1* in class *protoObject*. Each time an application object invokes a receive operation, it must first wait for its control object to send a message. This message controls the application object by determining which messages can be received from other objects. The nested rendezvous mechanism is shown in Figure 4-16.

The *waitToAccept(X)* method in the application object is redefined in its parent class *protoObject*. When executed by the application object, *waitToAccept(X)* invokes *waitToAccept(waitToAcceptControl(X))*. Now program control moves to its control object since only the control objects can send a *waitToAcceptControl(Msg)* message. After receiving the message from the control object, the *waitToAccept(X)* method invokes the *super-waitToAccept(X)* method which is the regular *waitToAccept(X)* method with the instantiated value *msg* for *X*. This achieves the message control mechanism through the control objects.

The MLog programming model is demonstrated in *protoObject* class and *protoControl* class. The *protoToolManager* object records the execution sequences. Whenever a control object is resumed, it reports to the *protoToolManager* who keeps track of activity. The implementation of the Proto-Control programming model in MLog are in *protoObject* class and *protoControl* class.

```

class: protoTool of object.
methods: class.
initialize :-
    new(protoToolManager,active,[]).
methods: instance.
begin :-
    mainConsole-(write("*** ProtoTool Trace Active ***"),nl(2)).
    repeat,
    waitToAccept(reportMessage(Controlld,ProxyId,Message)),
    fail.

reportMessage(Controlld,ProxyId,Message) :-
    mainConsole-(write(Controlld),writeSpaces(3),write(ProxyId),
    writeSpaces(3),writeln(Message)).

class: protoControl of object.
ivars: [proxyObject].
begin :-
    myself(MyId),
    theValueOf(proxyObject,ProxyId),
    repeat,
    processMessage(MyId,ProxyId),
    fail.
processMessage(MyId,ProxyId) :-

```

```

ProxyId-waitToAcceptControl(Message),
protoToolManager-reportMessage(MyId,ProxyId,Message).

```

```

class: protoObject of object.
methods: class.
new(X,active,Y) :-
    super-new(X,active,Y),
    protoControl-new(Z,active,[proxyObject=X]),
    mainConsole-(write("active object create! ").write(X),
                  write("control object create! ").writeln(Z)).
new(X,passive,Y) :-
    super-new(X,passive,Y).
methods: instance.
waitToAccept(X) :-
    super-waitToAccept(waitToAcceptControl(X)).
waitToAcceptControl(X) :-
    super-waitToAccept(X).

```

The above model provides the basic functions for the control process. Since the purpose of control object is to control the system run-time execution; it is not necessary to control the execution of system initialization. Thus, the control object must have the ability to distinguish the system initialization stage and execution stage. The methods in protoControl object have been modified as follow.

```

class:protoControl of object.
ivars:[proxyObject].
begin :-
    myself(MyId),
    theValueOf(proxyObject.ProxyId),
    protoNames-regist(ProxyId,MyId),
    phaseInit(MyId,ProxyId),
    repeat,
    processMessage(MyId,ProxyId),
    fail.

phaseInit(MyId,ProxyId) :-
    repeat,
    ProxyId-waitToAcceptControl(Message.P),
    protoToolManager-reportMessage(MyId,ProxyId,Message),
    Message == init.

processMessage(MyId,ProxyId) :-

```

```

ProxyId-waitToAcceptControl(Message,X).
protoToolManager-reportMessage(MyId,ProxyId,Message).

```

For each active object, after it finishes its initialization process, it sends a message to its ProtoObject to inform its initialization is done. Thus, the control object can proceed to the next procedure which enables execution control of the object.

MLog provides the timed receiver operation *waitToAccept/6*. This interaction should also be controlled by the control object, therefore, the instance method *waitToAccept/6* needs to be redefined in class *protoObject* so that the *protoControl* object can handle the timed receiver operations. The modification of the program model is as follows and the complete Proto-Control program is provided in Appendix A5.

```

class: protoObject of object.
methods: class.
new(X,active,Y) :-
    super-new(X,active,Y),
    protoControl-new(Z,active,[proxyObject=X]),
    mainConsole-(write("active object create! ").write(X).
                  write("control object create! ").writeln(Z)).
new(Z,passive,Y) :-
    super-new(X,passive,Y).

methods: instance.
initDone :-
    waitToAccept(init).
waitToAccept(X) :-
    super-waitToAccept(waitToAcceptControl(X,[])).

waitToAccept(Message,A,B,C,D,E) :-
    super-waitToAccept(waitToAcceptControl(Message,A,B,C,D,E)).

waitToAcceptControl(X,_) :-
    X isA nonvar,
    X == init.

waitToAcceptControl(X,[]) :-
    super-waitToAccept(X).

waitToAcceptControl(X,[A,B,C,D,E]) :-
    super-waitToAccept(X,A,B,C,D,E).

```

Figure 4-17 shows the Producer-Consumer timethreads overlaid on the architecture diagram. There are three kinds of objects: Producer objects (instances of class **Producer**), Consumer objects (instances of class **Consumer**), and a shared buffer called common object (instance of class **buffer**). The producer object is responsible for the producer timethread. The consumer object is responsible for the consumer object. And the common object is responsible for the common timethread. The MLog implementation of the Producer-Consumer program embedded in the proto-control model is shown below.

Figure 4-17: Producer-Consumer Architecture

```
class:protoObject of object.
class:main of object.
begin :-
    buffer-new(common,active,[]),
    producer-new(P,active,[]),
    consumer-new(C,active,[]),
    mainConsole-writeln("All tasks initialized").
```

```
class:buffer of protoObject.
ivars:[myBuffer=Q/Q].
store(DATA) :-
```

```

        theValueOf(myBuffer,B),
        enqueueDifferenceList(DATA,B,B2),
        makeTheValueOf(myBuffer,B2).
retrieve(DATA) :-
    size(S),S > 0,
    theValueOf(myBuffer,B),
    dequeueDifferenceList(DATA,B,B2),
    makeTheValueOf(myBuffer,B2).
size(S) :-
    theValueOf(myBuffer,B/[]),
    listLength(B,S).
enqueueDifferenceList(X,Q/[X|T],Q/T).
dequeueDifferenceList(X,[X|T]/T2,T/T2).

class:producer of protoObject.
begin :-
    buffer-new(X,passive,[myBuffer={apples,oranges,pears,kiwi|T}/T)),
    producerLoop(X).
producerLoop(X) :-
    X-retrieve(D),
    common-store(D),
    producerLoop(X).
producerLoop(X).

class:consumer of protoObject.
begin :-
    buffer-new(X,passive,[]),
    consumerLoop(X).
consumerLoop(X) :-
    common-retrieve(D),
    X-store(D),
    X-size(S),
    consumerLoop(X).

```

The program creates one active producer object, one active consumer object, one active buffer (**common**) object, and one passive buffer object. The items that the producer will produce are put into a list (named **myBuffer**, which is an instance variable of the passive object. The **common** object has the same instance variable name but with empty value initially). The class name description defines the class inheritance hierarchy. The result in Figure 4-18 is produced by the control objects and application objects. There is no interaction message displayed with producer and consumer objects since the proto-control

model only shows the received messages. In the result, the control object does not restrict the received messages, it only plays an auditing role.

The MLog console in Figure 4-18 shows text output from the application. The time-line display illustrates the start-up phase of behavior over the first 18 seconds of existence. Figure 4-19 shows the part of the time-lines in Figure 4-18 that illustrate the execution of the rendezvous. In Figure 4-19, we can see that the interactions of the **producer** object and **consumer** object with **common** object are indirect. The running of the **common** object is

transferred to the **protocontrol** object (#272) of the **common** object and the rendezvous starts here. After the object #272 receives the message from the producer object (#273) and the consumer object (#275), the execution goes back to the **common** object. The **common** object processes the *store(_)* message and completes the rendezvous with the producer object. Then the producer object reports to the **protoToolmanager**. The rendezvous with consumer object starts afterwards.

Figure 4-19: TimeLine Tool output of the Rendezvous for Producer-Consumer example

If the control object restricts the message received by the **common** object, e.g. the **common** object can only receive the *restore(X)* message, the result is shown below; i.e., the consumer object becomes suspended.

*** ProtoTool Trace Active ***

```
active object create! common control object create! #223
active object create! #224 control object create! #225
active object create! #226 control object create! #227
All tasks initialized
#223 common store(apples)
#223 common store(orange)
#223 common store(pears)
#223 common store(kiwi)
```

Control is limited in the proto-control model. For example, the proto-control model cannot handle the case that to allow the **producer** object to receive *store(apple)* message only and to allow the **consumer** object to receive *retrieve(apple)* message only because the *Message* in *ProxyId-waitToAcceptControl(Message)* applies to all the proto objects. It cannot be defined individually. However, the proto-control model is a good tool for auditing the system execution without affecting the source code structure. It provides a simple way to do software monitoring at a high level (Angio Trace provides the lower level of software monitoring [27]).

4.5 Timethreads Control Model

As shown in Figure 4-2, the timethread control model contains three parts: the executable timethreads model, the proto-control model, and the application model. Timethread control over the execution of application systems through control objects is called the timethreads control model.

Before sending a message to an application object, the control object must wait for the message from timethread object shown in Figure 4-20. Assume the active object A invokes a receive operation. The execution control goes up to its control object. Before sending the *waitToAcceptControl(X)* message, the control object must invoke another receive operation. The message that the control object is waiting for is *defineMsg(X)*. Now the execution control moves to the timethreads objects since only the timethreads objects can send out this message. In this way, the timethreads control the execution of the applications.

The execution of timethreads synchronizes the execution of the application program. Once an activity is invoked by a timethread, it sends the message to the corresponding

control object so that the control object tells the application object do the activity. This is done by sending a *defineMsg(X)* to the control object. Timethreads execution simulates the application execution, thus multiple timethreads compete to control the application objects.

Before sending a message *waitToAcceptControl(Message)* to the application object, the protoControl object must wait for the message *waitToAccept(defineMsg(Message))* from the corresponding timethread object. Therefore the MLog implementation of the protoControl object is changed as follows.

```
class: protoControl of object.
ivars: [proxyObject].
begin :-
    myself(MyId),
    theValueOf(proxyObject,ProxyId),
    repeat,
    processMessage(MyId.ProxyId),
    fail.
processMessage(MyId.ProxyId) :-
    waitToAccept(defineMsg(Message)),
    ProxyId-waitToAcceptControl(Message).
```

protoToolManager-reportMessage(MyId,ProxyId,Message).

The Producer-Consumer timethreads have been modified to respond to the changes of **protoControl** object. After the producer timethread or the consumer timethread send a message to the common timethread, they must send a message to the control object of the common object so the execution of the application objects synchronizes with the timethreads execution. The modified code is shown below.

```
doDecision(do,JobL) :-
    getAt(Msg,0,0,JobL),
    deleteFrom(Msg,JobL,LL),
    Msg.
doDecision(send,JobL) :-
    getAt(Msg,0,0,JobL),
    deleteFrom(Msg,JobL,LL),
    getAt(Info,0,0,LL),
    Msg-send(Info),
doDecision(receive,JobL) :-
    getAt(Msg,0,0,JobL),
    deleteFrom(Msg,JobL,LL),
    protoNames-lookup(common,ProtoName),
    waitToAccept(send(Msg)),
    ProtoName1-defineMsg(Msg).
send(store()) :-
    theValueOf(myBuffer,B),
    enqueueDifferenceList(DATA,B,B2),
    makeTheValueOf(myBuffer,B2).
send(retrieve()) :-
    size(S),S > 0,
    theValueOf(myBuffer,B),
    dequeueDifferenceList(DATA,B,B2),
    makeTheValueOf(myBuffer,B2).
```

Figure 4-21 shows the execution result of the timethreads control model of the Producer-Consumer problem. The **protoNames** object maintains a table of all pairs of an active application object and its control object. From Figure 4-21, the **protoControl** object ID of the **common** object is #230.

1. ProtoName is an object ID number which can be found in an object table. The implementation of the table is shown in chapter 5.

All the interactions between the objects along the time line can be seen explicitly in Figure 4-21. The **pcmain** object initiates the **producer**, **consumer**, and **common** objects. The **tthread** object initiates the **pt**, **ct**, and **common** timethreads (**pt** represents the producer timethread, **ct** represents the consumer timethread, and **common** represents the commonthread). The **common** object gets executed first. Its receive message is transferred to its **protoControl** object (#230). The receive operation in **protoControl** object is

redefined to nest another receive operation. This is called the nested rendezvous. The **producer** object (#231) starts the first rendezvous with the **common** object. Then the **common** object begins the nested rendezvous with the object #230. The **common** object sends the message *store(X)* received from the **pt** object to the object #230. After the nested rendezvous completes, the control goes back to the **common** object, which perform the real *store(X)* operation. The *retrieve(X)* operation does exactly the same interaction sequences.

Timethread control has some limitations. For example, in a time-out scenario, the timethread object can not force the application object to time-out or not time-out. This behavior can be decided only by the execution scenario of the application object. In this case, we want to disable the timethread control, but let the control object control the timethread's execution so that the timethread objects and application objects can still be running simultaneously. A detailed discussion and example of this limitation is given in chapter 5.

5

Case Study

This chapter describes the prototypes of the executable timethreads for the layered communications system to achieve the executable design. The implementation issues of the layered communications system in the MLog environment are discussed. Simple performance tests have been conducted and evaluated based on the design and the programming model of executable timethreads.

Section 5.1 shows the executable timethreads of the layered communications system. Section 5.2 discusses the functional implementation of the layered communications system. And section 5.3 illustrates the execution of the timethread control model over the application system and the experimental results.

5.1 Executable Timethreads for Layered Communications System

As discussed in chapter 3, Figure 3.3 shows the timethreads manual description for the layered communication system, and Figure 3.6 shows the design picture of the system. The mapping from Figure 3.3 to Figure 3.6 is a design process that assigns responsibilities to the timethreads. Figure 5.1 shows an end-to-end timethread overlaid on the design architecture. The end-to-end timethread is represented by the composition of several executable timethreads in Figure 3.1. The timethreads in Figure 5.1 illustrate only one direction of data flows, i.e. from **node1** to **node2**.

Figure 5-1: Timethreads Execution on the Design Picture

Timethread **t1** is responsible for receiving a packet from the network layer of the **node1** and carrying the packets to the link layer. On its way, timethread **t1** triggers timethreads **t2** and **t3**. Timethread **t2** is responsible for the *tickTask*. Timethread **t3** is responsible for receiving the packets from the link layer and sending the packets through the channel to the other end of the physical layer (which is **node2**). Timethread **t4** is triggered by timethread **t3**. Timethread **t4** is responsible for carrying the packets to the link layer of **node2**. At the end of timethread **t4**, it forks two timethreads **t5** and **t6**. Timethread **t5** is responsible for passing the data to **node2**'s network layer. Timethread **t6** is responsible for sending the acknowledgment back to the sender. Timethread **t6** would wake up timethread **t1** that would then stop the timer, or if a time-out happens first, **t3** would be executed again.

The interface buttons on the actors through which a timethread passes are the activities of the timethread. These activities are put into the timethread activity list implemented as an instance variable. The activities are categorized into three groups: **do**, **send**, and **receive**.

To make timethreads executable in the MLog programming environment, the timethreads diagram must be translated into the MLog script. Each timethread has been created as an instance object of the *thread* class. The instance variable *activity* explicitly describes the activities of the timethread. The instance variable *numOfActs* records the number of activities of a timethread. The MLog script of timethread **t1** is created as follows:

```
thread-new(t1,active,[activity=[[do,txTo(_,_),"node1",userMgr],
                             [do,getFor(_,_),"node1",userMgr],
                             [do,putPacket(_),"node1","channel1",tx],
                             [send,t2,startTimer(_,_)], [send,t3,sendDown(_,_)],
                             [receive,_,"node1","channel1",tx]],
                             numOfActs=6]),
```

The timethread **t1** has six activities. The first three activities are **do** activities. The fourth and fifth activities are **send**. The last activity is **receive**. In each activity list, there are some parameters that specify the contents of the activity. For example, the first activity is “**do** *txTo*(_,_) on *userMgr* object of *node1*,” the fourth activity is “**send** a *startTimer*(_,_) message to the timethread **t2**”; the last activity is “**receive** a message from *channel1* to *tx*”

2 of/de 2

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

PRECISIONSM RESOLUTION TARGETS

object of *node1*."

The operations of the timethreads and the synchronization among the timethreads are implemented using the techniques discussed in chapter 4. There is an interpreter to solve the activity list. The synchronization among the timethreads in the layered communications system is more complex than the producer/consumer threads. For example, the timed message passing is used in the layered communications system as shown below. The MLog code shows only the interactions (send, receive) between the timethreads. The complete timethreads implementation is given in appendix A7.

```
doDecision(doJobL) :-                                % do method
    getAt(Msg,0,0,JobL),
    deleteFrom(Msg,JobL,LL),
    do(Msg).
doDecision(sendJobL) :-                             % send method
    getAt(Msg,0,0,JobL),
    deleteFrom(Msg,JobL,LL),
    getAt(Info,0,0,LL),
    Msg-send(Info).
doDecision(receiveJobL) :-                          % receive method
    getAt(Msg,0,0,JobL),
    deleteFrom(Msg,JobL,LL),
    waitToAccept(send(Msg)),
    postRendezvous(send(Msg)).
postRendezvous(send(timeOut)) :-                    % timeout handler
    theValueOf(timeOutMark,TimeOutMark),
    N is TimeOutMark + 1,
    makeTheValueOf(timeOutMark,N).
postRendezvous(send(putAckFrame(_))) :-             % method for receiving ACK
    t2-send(stopTimer).
postRendezvous(send(startTimer(_))) :-              % method for starting timer
    waitToAccept(send(stopTimer).relative,[0,00,15],Duration,timeOut,Result).
postRendezvous(send(Msg)).
send(_).
timeOut :-
    t1-send(timeOut).
```

The execution results can be seen in Figure 5.2. The timelines show the interactions of the six timethreads in the first one minute and fifteen seconds. **tthread** initializes threads **t1** to **t6**. **t1** gets the first chance to run. It triggers **t2**, **t3**, and then suspends itself. **t2** starts the

timer. **t3** sends a packet to **node2** and triggers **t4**. **t4** runs and triggers **t5** and **t6**. **t5** is responsible for sending the packet to the receiver. **t6** is in charge of sending the acknowledgment to the sender. Once the ACK gets the sender's link layer, it wakes up **t1** that tells **t2** to stop the timer. By now, one round has completed and **t1** is ready to send the next packet.

Figure 5-2: TimeLine Tool output of the Timethreads Execution

The timethreads in Figure 5.1 represent one scenario of the timethreads execution. For different kinds of scenarios, we use different MLog scripts to show the situations. For example, timethread **t3** can be terminated at the channel because the transmission channel lost the data frame. After a while, the time-out must happen and timethread **t3** has to be executed again. In another scenario, the timethread could be terminated at the *Rx* of **node2**

because of an incorrect sequence bit. Different scenarios have to be described separately. For example, the timethread to move data from **node2** to **node1** would required separate specification by a set of MLog scripts. However, the instance methods of the thread are generic so no new MLog code needs to be written.

5.2 Implementation Issues for the Layered Communications System

The MLog implementation of the Layered Communications System is based on the work completed as a course project by Greg Bond. This thesis included analysis, restructuring, and enhancement of the original design - it has undergone substantial revision.

The design diagram is shown in Figure 3.11. The actors such as TX actor have been implemented as the active objects. The components such as layers have been implemented as passive objects. The interface buttons such as **putPacket** have been implemented as the object methods, which provide the interface to communicate with other objects. The data stores are implemented as buffer objects. The engine in TX and RX is implemented as the message flow control for the active objects.

Though the actors in each component are not implemented as the subclass of the component class, the component object controls these active objects. That means the component object creates these active objects and provides the intermediate service to the other layers so that the objects can communicate with each other. This achieves the code packaging of the layers. Figure 5.3 shows the component L that contains actors tickTask, packetTransporter, linkMgr, TX, and RX. The component L provides two interfaces, *rxfromuser* for communicating with the network layer, *rxfromphysical* for communicating with the physical layer. linkMgr controls the data flow to TX or RX, i.e., data or ack.

Figure 5-3: MachineChart of the Link Layer Architecture

The system can create multiple nodes and channels as shown in Figure 5.4. There is a node window for each node and a channel window for each channel. A node window shows the activities on that node; i.e. sending packets. A channel window shows the activities on that channel; i.e. a packet arrives at the channel, or a transmission error occurs. A channel is an end-to-end point connection between any two nodes. Therefore, there could be multiple link layers and physical layers for all of the end-to-end connections. The UserMgr actors in each node is responsible for routing packets. The channel is implemented as an object that has two ports (in and out) and a transmitter called *disturbanceTask* that is implemented as a passive object. The disturbance task can randomly generate four types of message status; i.e., OK, ChkSumError, LoseFrame, and SynchError.

After the system set up, the Done button in the Initialize Network System window finishes the initialization stage so that the system is ready to run. Each node has all the pop up windows for the connections it has with other nodes. The pop up window is the window for sending messages to the other node. Figure 5.4 shows the initialization stage of the system.

Figure 5.5 shows two way messages being sent between any two nodes. Each node can randomly generate packets and the transmission rate can be controlled. Each channel can randomly generate errors and the error rate can be controlled as well. The message passing activities are displayed on the corresponding windows. The MLog console displays the execution results. Figure 5-5 shows the run time execution of the Layered Communications system.

The interactions between TX and tickTask are demonstrated below. As shown in Figure 3-8, the TX trigger the timer and releases itself. The tickTask, after receiving the *startTimer* message, then waits for the *stopTimer* message or if the time-out happened, it sends the *timeout* message to the TX. The MLog implementation is shown below.

```

class: tickTask of protoObject.
ivars: [txTask].
methods: class.
methods: instance.
begin :-
    initDone.
    repeat.
    waitToAccept(startTimer(TxName,Packet)).
    makeTheValueOf(txTask,TxName).
    waitToAccept(stopTimer.relative,[0.59,20],Duration,timeOut,Result).
    fail.
startTimer(_).
stopTimer.
timeOut :-
    theValueOf(txTask.TxTask).
    TxTask-timeOut.

```

The TX object waits for the *Ack* or the *timeout* message after it sends out a message. If it receives the *Ack* message, the TX sends *stopTimer* message to the **tickTask**. Otherwise, the **tickTask** sends *timeOut* message to the TX and the TX retransmits the data and starts the timer again.

5.3 Timethreads Control Model for Layered Communications System

Once the layered communications system is implemented, the executable timethreads is used to control the execution of the actual system.

The timethreads control mechanism is achieved through the protoControl objects as discussed in chapter 4. In order to communicate between the timethread objects and application objects, the timethread objects must know the proto object name of the application object with which it wants to communicate. A table is maintained that keeps all pairs of the active application object and its control object, and all the passive objects and their binding names. The operations on the table are shown below. When an active object and its control object are created, they are registered in the table. If a timethread object wants to control an application object, it first searches the table to find the corresponding control object and send the message to the control object.

```

class: nameserver of object.
ivars: {namelist=[]}.
methods: class.
initialize :-
    new(protoNames,active,[]).
methods: instance.
regist(ProxyId,ProtoId) :-                                % regist method
    theValueOf(namelist,X),
    appendTo((ProxyId,ProtoId),X,Newlist),
    makeTheValueOf(namelist,Newlist).
lookup(ProxyId,ProtoId) :-                                % look up method
    theValueOf(namelist,X),
    findIn((ProxyId,ProtoId),X).
appendTo(Elmt,[],[Elmt]).                                  % append to the table
appendTo(Elmt,[HT1],[HT2]) :-
    appendTo(Elmt,T1,T2).
findIn(Elmt,[Elmt_]).                                     % search for an object from the table
findIn(Elmt,[_T]) :-
    findIn(Elmt,T).

```

As discussed in section 5.1, the timethreads execution carries out the activities in each timethread. Those activities are actual implemented as the instance methods in the real system. Since the system can create multiple nodes and channels, there are a link layer and a physical layer for each connection. The **UserMgr** object in each node is responsible for routing the packets.

In order to distinguish the object's family and their connections, a lookup table is maintained. The table sets up all the relationships between the objects and their binding names. This helps the timethreads objects to find the right application objects to control. A simple example is shown below.

```

class: tableserver of object.
ivars: {tablelist=[]}.
methods: class.
initialize :-
    new(tablename,active,[]).
methods: instance.
begin :-
    objectNames-lookup(("node1",User1)),
    objectNames-lookup((User1,UserMgr1,UserTrans1)),

```

```

    regist([UserMgr1,"node1".userMgr]),
    regist([UserTrans1,"node1".userTransport]).

```

The following result shows the execution of the layered communications system under the normal condition; i.e., a packet is transmitted successfully. The time duration from starting the timer to stopping the timer is 26 seconds.

```

MLog (C) Copyright 1989, 1990, 1991, 1992 Carleton University
*** ProtoTool Trace Active ***
#243 #242 txTo(node2,1103527590)
#243 #242 getFor(1103527590,#311)
#304 #303 startTimer(#305,1103527590)
#295 #294 sendDown(#277,#343)
#306 #305 putPacket(1103527590)
#318 #317 write(#343)
#275 #274 txToAt(#343,#278,node2)
#318 #317 read(#343)
#335 #334 putFrame(#343)
#322 #321 sendDown(#278,#344)
#327 #326 putDataFrame(#343)
#291 #290 write(#344)
#275 #274 txToAt(#344,#277,node1)
#327 #326 getPacket(1103527590)
#261 #260 putFrom(1103527590,#338)
#291 #290 read(#344)
#308 #307 putFrame(#344)
Duration:[0,0,26]Result:message Packet:1103527590
#306 #305 putAckFrame(#344)
#304 #303 stopTimer

```

The execution of the timethreads control over the layered communications system is demonstrated below. The time duration from starting the timer to stopping the timer is tremendously increased because of the interactions with the timethreads. The timethreads are running concurrently with the application objects.

```

MLog (C) Copyright 1989, 1990, 1991, 1992 Carleton University

```

```

*** ProtoTool Trace Active ***
#t1 (0 6, do [txTo(_),node1,userMgr]
#t2 (0 1, receive [startTimer(_),node1,channel1,tickTask]
#t3 (0 4, receive [sendDown(_),node1,channel1,downTransport]
#t4 (0 5, receive [read(_),node2,channel1,bufferTask]

```

```
#t5 (0 2, receive [getPacket(_),node2,channel1,rx]
#t6 (0 6, receive [sendDown(_),node2,channel1,downTransport]
#t1 (1 6, do [getFor(_),node1,userMgr]
#243 #242 txTo(node2,1103527590)
#t1 (2 6, do [putPacket(_),node1,channel1,tx]
#243 #242 getFor(1103527590,#311)
#t1 (3 6, send [t2,startTimer(_)]
#t1 (4 6, send [t3,sendDown(_)]
#t1 (5 6, receive [_ ,node1,channel1,tx]
#t3 (1 4, do [txToAt(_,_),channel1,disturbanceTask]
#304 #303 startTimer(#305,1103527590)
#295 #294 sendDown(#277,#343)
#306 #305 putPacket(1103527590)
#t3 (2 4, do [write(_),node2,channel1,bufferTask]
#t3 (3 4, send [t4,read(_)]
#t3 finished
#318 #317 write(#343)
#275 #274 txToAt(#343,#278,node2)
#t4 (1 5, do [putFrame(_),node2,channel1,linkMgr]
#318 #317 read(#343)
#t4 (2 5, do [putDataFrame(_),node2,channel1,rx]
#335 #334 putFrame(#343)
#t4 (3 5, send [t5,getPacket(_)]
#t4 (4 5, send [t6,sendDown(_)]
#t4 finished
#t6 (1 6, do [txToAt(_,_),channel1,disturbanceTask]
#322 #321 sendDown(#278,#344)
#327 #326 putDataFrame(#343)
#t5 (1 2, do [putFrom(_),node2,userMgr]
#t6 (2 6, do [write(_),node1,channel1,bufferTask]
#327 #326 getPacket(1103527590)
#t5 finished
#t6 (3 6, do [read(_),node1,channel1,bufferTask]
#261 #260 putFrom(1103527590,#338)
#291 #290 write(#344)
#275 #274 txToAt(#344,#277,node1)
#t6 (4 6, do [putFrame(_),node1,channel1,linkMgr]
#291 #290 read(#344)
#t6 (5 6, send [t1,putAckFrame(_)]
#t6 finished
#308 #307 putFrame(#344)
#t1 finished
#306 #305 putAckFrame(#344)
#t2 finished
```

Duration:[0,1,7]Result:message Packet:1103527590
#304 #303 stopTimer

In order to get rid of the overhead that the timethreads cost, the timethread objects can be switched to the system queue so that the execution time of the timethreads are excluded. The *protect(X)* and *unprotect(X)* are the two functions to switch the application objects that are created in between these two functions into the system queue. For example, in the following piece of code, the timethread *t1* is put on the system ready-to-run queue.

```
...
    protect(X),
    thread-new(t1,active,[activity=[[do,txTo(_,_),"node1",userMgr],
                                [do,getFor(_,_),"node1",userMgr],
                                [do,putPacket(_),"node1","channel1",tx],
                                [send,t2,startTimer(_,_)],[send,t3,sendDown(_,_)],
                                [receive,_,"node1","channel1",tx]],
                                numOfActs=6)),
    unprotect(X),
...
```

After assigning the timethreads to the system queue, the time duration is decreased from 1 minute and 7 seconds to 53 seconds. However, it is still greater than the time spent without timethread control (which is 26 seconds). This is due to the interactions with timethreads, but not specifically the timethread object code.

The tests with the timethreads control model are based on the normal execution of the system (which means no time-out or transmission errors occurred). A limitation of the timethreads control model is that the timethreads cannot control the time-out scenario of the application objects. That is to say the timethread objects cannot force the application object to time-out or not time-out. This behavior can be decided only by the execution scenarios of the application objects. In this case, we want to disable the timethread control, but let the control object to control the timethreads execution so that the timethreads objects and application objects can still be running synchronously. That is to say, if a timeout happens in the application system, it could inform the timethreads so that the timethreads could behave the same way as application objects, instead of sending the messages to the

application objects which conflict with the behaviour of the application objects and cause the deadlock. In this way, the timethreads have to be both the managing objects and managed objects. This challenging problem is not addressed in the thesis due to time constraints.

6

Conclusion

6.1 A Research Summary

This thesis described the use of timethreads as a basis for bridging activity diagram (used to construct the performance model) and design models of software systems. Two approaches were described: (1) a manual technique for mapping activities in a performance model to the changes in a design, as modelled by timethreads and MachineCharts, and (2) executable timethreads, which provide a language-enforced concordance between the timethreads and changes in the software design.

The first technique employs timethreads in a manual fashion: each design refinement, from the requirements to the final design, reports the changes in timethreads. Thus there is a trail to follow from the performance model activity diagram to the software design, and the changes to activities can be tracked along the refinements. The software design could be realized as a high level prototype or portions of an actual implementation, and performance experiments could be conducted accordingly. Also, the activity model could be manually refined to correspond to some point along the design. The problem with this approach is that concordance between the timethreads and the software design must be maintained manually.

The second technique uses a programming model of executable timethreads to control an application (design) model, and thus mechanically recording the concordance through the formality of a programming language. As timethreads and software objects are refined, the designer modifies the descriptions of the timethreads and software objects and their

relation. Execution of the design model becomes execution of the timethreads to control the activities embodied by the software objects. The designer can construct performance experiments such as branching probabilities based on the design. Since the timethreads are the objects that are executing (or at least controlling execution) it is easier to relate the experiments to the objects, and perform measurements. The disadvantage of this approach is the need for a specialized execution environment to support timethread execution. There are also limitations in the executable timethreads control model; the executable timethreads can control the normal execution of the application, but for the abnormal situation (e.g., a transmission problem causes a time-out), the executable timethreads cannot project this situation and force the application to respond.

The thesis used a layered communications system example to illustrate the manual and executable methods. The manual example provided sufficient complexity to demonstrate that timethreads can be used to relate the activity diagram and design models. It also showed the difficulty of keeping track of many small details. The timethreads of the layered communications system are executable so that the design of the system can be evaluated at very early stage, and later on, the executable timethreads can control the execution path of the real system under normal conditions (i.e., a packet is transmitted successfully) and record the execution data mechanically. However, the case study also discovered a drawback of the executable timethreads control model - the executable timethreads model can not control the execution of the system under abnormal situations (i.e., a packet has to be retransmitted due to a time-out) because of the way the executable timethreads model is constructed. This was revealed when trying to force the time-out case in the applications by the timethreads. The application times out when it sees a problem in the external transmission of packets, not because a timethread wants to direct the application into an error. The timing measurements with and without timethread controlling the application also showed a significant overhead due to timethread control, even when the execution of the timethread object themselves were not counted.

6.2 Future Research

Future research can build on what has been learned from the executable timethreads model to develop a method for constructing executable timethreads, testing and using them to control applications under a greater variety of running conditions.

The executable timethreads model can be extended to automatically instrument as well as (or instead of) controlling the application; e.g., to direct the gathering of performance data (like *Angio Trace* [27]) along the execution path.

The executable timethreads model can be redeveloped to function in a well known environment such as Unix and C++ to reduce overhead and conduct performance experiments.

Further case studies particularly with performance experiments need to be performed to examine and evaluate the techniques.

Bibliography

- [1] M. Ajmone-Marsan, G. Balbo, and G. Conte, "A Class of Generalized stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems," *ACM Transactions on Computer Systems*, Vol. 2, No. 2, May 1984, pp. 93.
- [2] M. W. Alford, "A Requirements Engineering Methodology for Real-Time Processint Requirements," *IEEE Transactions on Software Engineering*, Vol. 3, No. 1, January 1977, pp. 60-69.
- [3] M. W. Alford, "SREM at the Age of Eight; The Distributed Computing Design System," *IEEE Computer*, Vol. 18, No. 4, April 1985, pp. 36-46.
- [4] G. Balbo, S. C. Bruell, and S. Ghanta, "Combining Queueing Network and Generalized Stochastic Petri Net Models for the Analysis of Some Software Blocking Phenomena," *IEEE Transactions on Software Engineering*, Vol. 12, No. 4, April 1986, pp. 561-576.
- [5] R. Balzer, "Draft Report on Requirements for a Common Prototyping System," *SIGPLAN Notices*, Vol. 24, No. 3, Nov. 1988, pp. 93-114.
- [6] R. Budde, K. Kautz, K. Kuhlenkamp and H. Zullighoven, *Prototyping: An Approach to Evolutionary System Development*, New York, NY:Springer-Verlag, 1992.
- [7] B. Berthomieu and M. Diaz, "Modeling and Verification of Time Dependent Systems Using Time Petri Nets," *IEEE Transactions on Software Engineering*, Vol. 17, No. 3, March 1991, pp. 259-273.
- [8] G. Booch, *Software Engineering with Ada*, Second Edition, Menlo Park, Redwood City, CA:Benjamin/Cummings, 1983.
- [9] G. Booch, "Object-Oriented Development," *IEEE Transactions on Software Engineering*, Vol. 12, No. 2, Feb. 1986, pp. 211.
- [10] G. Booch, *Object Oriented Design*, Menlo Park, Redwood City, CA:Benjamin/Cummings, 1992.

- [11] G. Bruno and G. Marchetto, "Process-Translatable Petri Nets for the Rapid Prototyping of Process Control Systems," *IEEE Transactions on Software Engineering*, Vol. 12, No. 2, March 1986, pp. 346-357.
- [12] R. J. A. Buhr, *Practical Visual Techniques in System Design (with Applications to Ada)*, Englewood Cliffs, NJ: Prentice-Hall, 1990.
- [13] R. J. A. Buhr, *System Design with Ada*, Englewood Cliffs, NJ: Prentice-Hall, 1984.
- [14] R. J. A. Buhr, "Pictures that Play: Design Notations for Real-Time and Distributed Systems," *Software Practice and Experience*, Vol. 23., No. 8, 1993, pp. 895.
- [15] R. J. A. Buhr and R. C. Casselman, "Architectures with Pictures," *Proceedings of OOPSLA'92, ACM/SIGPLAN*, Vancouver, Canada, October 1992, pp. 466-483.
- [16] R. J. A. Buhr and R. C. Casselman, *Designing with Timethreads*, Technical Report SCE-93-05, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada, May 1993.
- [17] R. J. A. Buhr and R. C. Casselman, *Notation for Threads*, Technical Report SCE-92-07, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada, May 1992.
- [18] R. J. A. Buhr, G. M. Karam, C. J. Hayes and C. M. Woodside, "Software CAD: A Revolutionary Approach," *IEEE Transactions on Software Engineering*, Vol. 15, No. 3, March 1989, pp. 235-249.
- [19] R. J. A. Buhr, G. M. Karam, C. M. Woodside, et. al., "TimeBench: a CAD Tool for Real-Time system System Design," in *Proceedings of IEEE/ACM 2nd Intl. Symposium on Environments and Tools for Ada (SETA2)*, Washington D. C., January 1991, pp. 3-15.
- [20] G. Cherry, *The PAMELA Designer's Handbook*, Thought Tools, Reston, VA, 1986.
- [21] W. W. Chu, C. Sit, and K. K. Leung, "Task Response Time for Real-Time Distributed Systems with Resource Contentions," *IEEE Transactions on Software Engineering*, Vol. 17, No. 10, October 1991, pp. 1076-1092.
- [22] T. DeMarco, *Structured Analysis and System Specification*, New York, NY: Yourdon Press, 1978.
- [23] D. Ferrari, "Considerations on the Insularity of Performance Evaluation," *IEEE Transactions on Software Engineering*, Vol. 12, No. 6, June 1986, pp 678-683.

- [24] C. Gane and T. Sarsen, *Structured Systems Analysis: Tools and Techniques*, Englewood Cliffs, NJ:Prentice-Hall, 1979.
- [25] H. J. Genrich and K. Lautenbach, "System Modelling with High-Level Petri Nets," *Theoretical Computer Science*, No. 13, 1981, pp. 109-136.
- [26] A. Goldberg and D. Robson, *Smalltalk-80: The Language And Its Implementation*, MASS: Addison-Wesley Publishing Company, 1985.
- [27] C. Hrischuk, J. Rolia, and C. M. Woodside, "Automatic Generation of a Software Performance Model Using an Object-Oriented Prototype", International Workshop on Modeling and Simulation, Analysis, and Simulation of Computer and Telecommunication Systems, 1995.
- [28] C. Hrischuk and C. M. Woodside, *Prototyping for Performance Measures*, Technical Report SCE-93-16, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada, 1993.
- [29] R. Jain, *The Art of Computer Systems Performance Analysis*, New York, NY: John Wiley & Sons, 1991.
- [30] I. Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach*, MASS: Addison-Wesley, 1992.
- [31] K. Jensen, "Coloured Petri Nets and The Invariant-Method," *Theoretical Computer Science*, No. 14, 1981, pp. 317-336.
- [32] G. M. Karam, et al., *The MLog User Guide*, Technical Report SCE-89-15, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada, 1992.
- [33] G. M. Karam and J. T. Wilson, "MLog: A Language for Prototyping Concurrent Systems," *Canadian Conference on Electrical and Computer Engineering*, Ottawa, Canada, 1990, pp. 43.2.1-43.2.6.
- [34] R. Kowalski, "Algorithm = Logic + Control," *Communications of the ACM*, Vol. 22, No. 7, July 1979, pp. 424-436.
- [35] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*, Englewood Cliffs, NJ: Prentice-Hall, 1984.
- [36] S. Levi and A. K. Agrawala, *Real-Time System Design*, New York, NY: McGraw-Hill Publishing Company, 1990.

- [37] Luqi, "Computer Aided Prototyping for a Command and Control System Using Caps," *IEEE Software*, Jan. 1992, pp. 66-72.
- [38] Luqi and V. Berzins, "Rapidly Prototyping Real-Time Systems," *IEEE Software*, Sept. 1988, pp. 25-36.
- [39] Meta Software Corporation, *Design/CPN: A Reference Manual*, Version 1.9, April 1992.
- [40] M. K. Molloy, *Fundamentals of Performance Modeling*, New York: Macmillan, 1989.
- [41] T. Murata, "Petri Nets: Properties, Analysis, and Applications," *Proceedings of the IEEE*, vol. 77, No. 4, April 1989, pp. 541-580.
- [42] D. Peng and K. G. Shin, "Modeling of Concurrent Task Execution in a Distributed System for Real-Time Control," *IEEE Transactions on Computers*, Vol. 36, No. 4, April 1987, pp. 500-516.
- [43] J. L. Peterson, *Petri Net Theory and the Modelling of Systems*, Englewood Cliffs, NJ: Prentice-Hall, 1981.
- [44] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, New York, NY: McGraw-Hill, 1987.
- [45] C. H. Sauer and K. M. Chandy, *Computer Systems Performance Modeling*, Englewood Cliffs, NJ: Prentice-Hall, 1981.
- [46] S. R. Schach, *Software Engineering*, II., Holmwood: Irwin and Aksen Associates, 1990.
- [47] Scientific and Engineering Software Inc., *SES/workbench: User's Manual*, Release 2.1, February 1992.
- [48] C. U. Smith, *Performance Engineering of Software Systems*, Reading, Mass: Addison-Wesley, 1990.
- [49] C. U. Smith and Lloyd G. Williams, "Software Performance Engineering: A Case Study Including Performance Comparison with Design Alternatives," *IEEE Transactions on Software Engineering*, Vol. 19, No. 7, July 1993, pp. 720-741.
- [50] L. Sterling and E. Shapiro, *The Art of Prolog: Advanced Programming Techniques*, Cambridge: The MIT Press, 1986.

- [51] A. Thomasian and P. F. Bay, "Analytic Queuing Network Models for Parallel Processing of Task Systems," *IEEE Transactions on Computers*, Vol. 35, No. 12, December 1986, pp. 1045-1054.
- [52] J. Walrand, *Communication Networks: A First Course*, II., Holmwood: Irwin and Aksen Associates, 1991.
- [53] P. T. Ward, "The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing," *IEEE Transactions on Software Engineering*, Vol. 12, No. 2, February 1986, pp. 198-210.
- [54] P. T. Ward and S. J. Mellor, *Structured Development for Real-Time Systems, Volumes 1, 2, and 3*, New York, NY: Yourdon Press, 1985.
- [55] J. T. Wilson, *A Prototyping Environment For Concurrent Systems*, Master Thesis, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada, 1990.
- [56] C. M. Woodside, *A Three-View Model for Performance Engineering*, Technical Report SCE-93-19, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada, 1993.
- [57] C. M. Woodside, E. M. Hagos, E. Neron, and R. J. A. Buhr, "The CAEDE Performance Analysis Tool," *Ada Letters*, Vol. XI, No. 3, Spring 1991, pp. 125-136.
- [58] C. M. Woodside, J. E. Neilson, D. C. Petriu, and S. Majumdar, *The Rendezvous Network Model for Performance Synchronous Multi-Tasking Distributed Software*, Technical Report SCE-89-8, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada, March 1989.
- [59] E. Yourdon and L. L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Englewood Cliffs, NJ: Prentice-Hall, 1979.
- [60] P. Zave and W. Schell, "Salient Features of an Executable Specification Language and its Environment," *IEEE Transactions on Software Engineering*, Vol. 12, No. 2, February 1986, pp. 312-325.

Appendix: Notation

A.1 TimeThread Notation

Concurrent Threads

Independent timethreads represent concurrent behaviors in a system

Sequences of Threads

Upon termination of an incoming timethread, an outgoing timethread is initiated

Spawning Threads

A new (spawned) timethread is initiated as an activity along the path of a timethread.

And-Join

An outgoing timethread is initiated immediately after all of the incoming timethreads terminate.

And-Fork

Upon termination of an incoming timethread, several outgoing timethreads are initiated.

Or-Join

Upon termination of one of a set of incoming timethreads, an outgoing timethread is initiated.

Or-Fork

Upon termination of an incoming timethread, exactly one of several outgoing timethreads will be initiated. The selection condition is annotated.

Stub thread

A stub thread represents local detail hidden

Trigger Thread

A timethread triggers another timethread that is suspended at a waiting place (a large dark dot) which represents a location along the path where one timethread must synchronize with another timethread.

Trigger a Timer

A timer thread is initiated as an activity along the path of a thread. A timer thread starts with a timer object rather a thread start. The timer object starts the timer thread behaviour when an elapsed time period has passed.

Waiting on a Timer

A thread triggers a timer thread and then suspends at a waiting place for the timer thread to trigger it. The timer thread starts when the time period elapses and then proceeds to trigger the waiting place, and terminates.

Conditional Timed Wait

A thread triggers a timer thread and then waits at an Or-Join for either the termination of the timer thread, or the termination of one or more other threads.

A.2 MachineChart Notation

Box

Represents 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

Active boxes. They are autonomous, self-directing components. They map to tasks in Ada or tasks as supported by real-time executives.

Reactor

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		Represents procedural abstractions, and map to re-entrant procedures or functions in programming languages. Buttons may be combined with boxes.
Gate		Represents procedural abstractions but impose synchronization among actors visiting them.
Store		Place for keeping data. They map to variables in programming languages.
Finger		Attached to buttons or gates.
Waiting Place		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.
Data Flow		Indicates the transfer of data between components.

A.3 Activity Diagram Notation

Event Ready		A starting place for an event. events flow through an activity diagram like tokens in a Petri Net.
Event End		A termination place for an event.

AND Fork		An event splits into multiple events and travels down multiple, concurrent event paths.
AND Join		When an event arrives on each of the input event paths, a single event emerges from the output event path.
OR Fork		An event travels along one and only one of several outgoing event paths. The selection condition may be stochastic (for performance modelling) or deterministic.
OR Join		An event along one of the incoming event paths is routed along the single outgoing event path.
Race Line		Coordinates the event flow along competing concurrent event paths. An event flow entering a race line, leaves along its outgoing path so long as it was not disabled by an inhibitor arc (see below).
Inhibitor arc		Disables flow along an event path if it is "unfilled". If it is "filled" then events may flow along the path. An active input place will unfill the circle.
Activity Box		The annotations on activity boxes identify the textual descriptions of activities.

A.4 TimeLine Tool

Task_Created		The specified task has been created
Task_Removed		The specified task has terminated

Task_Running		The specified task or ISR is running
Task_Ready		The specified task or ISR is ready
Task_Waiting		The specified task or ISR is waiting for an interaction
Sync_interaction Initiated		A rendezvous has been initiated
Sync_interaction Established		The receiver of the message has begun to process it
Sync_interaction Failed		The receiver of the message put it back on the message queue
Sync_interaction Completed		The rendezvous has completed successfully

Appendix: MLog Code

B.1 Proto-Control Program

```
class: protoTool2 of object.
methods: class.
initialize :-
    new(protoToolManager,active,[]).
methods: instance.
begin :-
    mainConsole-(write("**** ProtoTool Trace Active ****"),nl(2)),
    repeat,
    waitToAccept(reportMessage(ControllId,ProxyId,Message)),
    fail.
reportMessage(ControllId,ProxyId,Message) :-
    mainConsole-(
        write(ControllId),writeSpaces(3),write(ProxyId),
        writeSpaces(3),writeLn(Message)).

class:protoControl of object.
ivars:[proxyObject].
begin :-
    myself(MyId),
    theValueOf(proxyObject,ProxyId),
    protoNames-regist(ProxyId,MyId),
    phaseInit(MyId,ProxyId),
    repeat,
    processMessage(MyId,ProxyId),
    fail.
phaseInit(MyId,ProxyId) :-
    repeat,
    ProxyId-waitToAcceptControl(Message),
    protoToolManager-reportMessage(MyId,ProxyId,Message),
    Message == init.
processMessage(MyId,ProxyId) :-
    % waitToAccept(defineMsg(Message)),
    ProxyId-waitToAcceptControl(Message,_),
    protoToolManager-reportMessage(MyId,ProxyId,Message).
```

```

defineMsg(_).
class: protoObject of object.
methods: class.
new(X,active,Y) :-
    super-new(X,active,Y).
    protoControl-new(Z,active,[proxyObject=X]),
    mainConsole-(write("active create! "),writeln(Z)).
new(X,passive,Y) :-
    super-new(X,passive,Y).
methods: instance.
initDone :-
    waitToAccept(init).
waitToAccept(X) :-
    super-waitToAccept(waitToAcceptControl(X,[])).
waitToAccept(X,A,B,C,D,E) :-
    super-waitToAccept(waitToAcceptControl(X,[A,B,C,D,E])).
waitToAcceptControl(X,_):-
    X isA nonvar,
    X == init.
waitToAcceptControl(X,[]) :-
    super-waitToAccept(X).
waitToAcceptControl(X,[A,B,C,D,E]) :-
    super-waitToAccept(X,A,B,C,D,E).

```

B.2 Producer-Consumer Program

```

class:protoObject of object.
class:main of object.
begin :-
    buffer-new(common,active,[]),
    producer-new(P,active,[]),
    consumer-new(C,active,[]),
    mainConsole-writeln("All tasks initialized").

class:buffer of protoObject.
ivars:[myBuffer=Q/Q].
begin :-
    repeat,
    waitToAccept(X),
    postRendezvous(X),
    fail.

show.
postRendezvous(show) :-

```

```

        mainConsole-writeln("Buffer active"),
        initDone,
        postRendezvous(_).
store(DATA) :-
    theValueOf(myBuffer,B),
    enqueueDifferenceList(DATA,B,B2),
    makeTheValueOf(myBuffer,B2).
retrieve(DATA) :-
    size(S),S > 0,
    theValueOf(myBuffer,B),
    dequeueDifferenceList(DATA,B,B2),
    makeTheValueOf(myBuffer,B2).
size(S) :-
    theValueOf(myBuffer,B/[]),
    listLength(B,S).
enqueueDifferenceList(X,Q/[X|T],Q/T).
dequeueDifferenceList(X,[X|T]/T2,T/T2).

class:producer of protoObject.
begin :-
    buffer-new(X,passive,[myBuffer=[apples,oranges,pears,kiwi|T]/T]),
    initDone,
    common-show,
    producerLoop(X).
producerLoop(X) :-
    X-retrieve(D),
    mainConsole-(write("sending "),writeln(D)),
    common-store(D),
    producerLoop(X).
producerLoop(X).

class:consumer of protoObject.
begin :-
    buffer-new(X,passive,[]),
    initDone,
    consumerLoop(X).
consumerLoop(X) :-
    mainConsole-writeln("waiting to receive"),
    common-retrieve(D),
    X-store(D),
    X-size(S),
    mainConsole-(write(D),write(" received"),nl),
    mainConsole-(write(S),write(" elements received"),nl),
    consumerLoop(X).

```

B.3 Executable Timethreads of the Layered Communications System

```

class: thread of object.
ivars: [activity=[],numOfActs,index=0,timeOutMark=0].
methods: class.
methods: instance.
begin :-
    loop.
loop :-
    theValueOf(index,Index),
    theValueOf(numOfActs,NumOfActs),
    theValueOf(timeOutMark,TimeOutMark),
    mainControl(Index,NumOfActs,TimeOutMark),
    myself(MyId),
    mainConsole-(write(MyId),writeln(" finished")),
    loop.
mainControl(Y,Y,0).
mainControl(Y,Y,1) :-
    NewIndex is Y - 3,
    theValueOf(timeOutMark,TimeOutMark),
    N is TimeOutMark - 1,
    makeTheValueOf(timeOutMark,N),
    mainControl(NewIndex,Y,N).
mainControl(Y,Z,0) :-
    myself(MyId),
    mainConsole-(write(MyId),write("("),write(Y),write(" ").writeln(Z)),
    getAt(L,Y),
    getAt(Felmt,0,0,L),
    deleteFrom(Felmt,L,NewL),
    mainConsole-(write(Felmt),write(" "),writeln(NewL)),
    doDecision(Felmt,NewL),
    NewIndex is Y + 1,
    theValueOf(timeOutMark,TimeOutMark),
    mainControl(NewIndex,Z,TimeOutMark).
doDecision(do,JobL) :-
    getAt(Msg,0,0,JobL),
    deleteFrom(Msg,JobL,LL),
    tablename-search([LL],X),
    protoNames-lookup(X,ProtoName),
    ProtoName-defineMsg(Msg),

```



```

    getTime(T),
    mainConsole-(write(Msg),write("starts at "),writeln(T)).

doDecision(send,JobL):-
    getAt(Msg,0,0,JobL),
    deleteFrom(Msg,JobL,LL),
    getAt(Info,0,0,LL),
    Msg-send(Info).
doDecision(receive,JobL):-
    getAt(Msg,0,0,JobL),
    deleteFrom(Msg,JobL,LL),
    tablename-search([LL],X),
    protoNames-lookup(X,ProtoName),
    waitToAccept(send(Msg)),
    postRendezvous(send(Msg),ProtoName).
postRendezvous(send(timeOut),ProtoName):-
    ProtoName-defineMsg(timeOut),
    getTime(T),
    mainConsole-(write("timeOut"),write("starts at "),writeln(T)),
    theValueOf(timeOutMark,TimeOutMark),
    N is TimeOutMark + 1,
    makeTheValueOf(timeOutMark,N).
postRendezvous(send(putAckFrame(_)),ProtoName):-
    ProtoName-defineMsg(putAckFrame(_)),
    getTime(T),
    mainConsole-(write("putAckFrame"),write("starts at "),writeln(T)),
    t2-send(stopTimer).
postRendezvous(send(startTimer(_)),ProtoName):-
    ProtoName-defineMsg(startTimer(_)),
    getTime(T),
    mainConsole-(write("startTimer"),write("starts at "),writeln(T)),
    waitToAccept(send(stopTimer),relative,[0,60,15],Duration,timeOut,
    Result),
    timeChoice(Result,ProtoName).
timeChoice(message,ProtoName):-
    mainConsole-(writeln("message")),
    ProtoName-defineMsg(stopTimer),
    getTime(T),
    mainConsole-(write("stopTimer"),write("starts at "),writeln(T)).
timeChoice(timeout,ProtoName):-
    mainConsole-(writeln("timeout")).
postRendezvous(send(Msg),ProtoName):-
    ProtoName-defineMsg(Msg),
    getTime(T),

```

```

        mainConsole-(write(Msg).write("starts at ").writeln(T)).
send(_).

timeOut :-
    getTime(X),
    mainConsole-(write("timeout at ").writeln(X)),
    t1-send(timeOut).
appendTo(Elmt,[],(Elmt)).
appendTo(Elmt,[HIT1],[HIT2]) :-
    appendTo(Elmt,T1,T2).
deleteFrom(Elmt,[ElmtfT],T).
deleteFrom(Elmt,[HIT1],[HIT2]) :-
    deleteFrom(Elmt,T1,T2).
getAt(Elmt,Position) :-
    theValueOf(activity,Activity),
    getAt(Elmt,0,Position,Activity).
getAt([],X,X,[]).
getAt(Elmt,X,X,[Elmtf_]).
getAt(Elmt,X,Y,[HIT]) :-
    NewX is X + 1,
    getAt(Elmt,NewX,Y,T).

class: timethreads of object.
methods: class.
initialize :-
    new(tthreads,active,[]).
methods: instance.
begin :-
    thread-new(t1,active,[activity=[(do,txTo(_,_),"node1",userMgr),
        (do,getFor(_,_),"node1",userMgr),
        (do,putPacket(_),"node1","channel1",tx),
        (send,t2,startTimer(_,_)),(send,t3,sendDown(_,_)),
        (receive_,_,"node1","channel1",tx)],
        numOfActs=6)),
    thread-new(t2,active,[activity=[(receive,startTimer(_,_),"node1",
        "channel1",tickTask)],numOfActs=1]),
    thread-new(t3,active,[activity=[(receive,sendDown(_,_),"node1",
        "channel1",downTransport),
        (do,txToAt(_,_),"channel1",disturbanceTask),
        (do,write(_),"node2","channel1",bufferTask),
        (send,t4,read(_))],numOfActs=4]),
    thread-new(t4,active,[activity=[(receive,read(_),"node2","channel1",bufferTask),
        (do,putFrame(_),"node2","channel1",linkMgr),
        (do,putDataFrame(_),"node2","channel1",rx)].

```

```
[send,t5.getPacket(_)],[send,t6.sendDown(_)]],
  numOfActs=5)),
thread-new(t5,active,[activity=[[receive,getPacket(_),"node2","channel1",rx],
  [do,putFrom(_),"node2",userMgr]],
  numOfActs=2)),
thread-new(t6,active,[activity=[[receive,sendDown(_),"node2",
  "channel1",downTransport],
  [do,txToAt(_,_),"channel1",disturbanceTask],
  [do,write(_),"node1","channel1",bufferTask],
  [do,read(_),"node1","channel1",bufferTask],
  [do,putFrame(_),"node1","channel1",linkMgr],
  [send,t1.putAckFrame(_)]],numOfActs=6)),
mainConsole-(writeln("All timethreads initialized")).
```

END

1 6-1 1-9 5

FIN