Chapter 2

Background

This chapter summarizes the key issues and concepts relevant to this thesis. In the first section, key OO system concepts and characteristics are described, followed by a discussion about OO development process. Then, Concurrent OO Systems are discussed as natural extension of OO systems. The text focuses on clarifying various types of concurrency and their distinctive behaviors. Finally, issues related to software performance modelling are examined, several important performance models and their merits are summarized. Various design and performance modelling notations are used in this chapter, detailed descriptions of these notations can be found in Appendix A. Also since the chapter covers more than one field in computer science, for some terms and concepts used in Computer System Performance and Operating Systems, the chapter simply gives references to the bibliography instead of giving detailed definitions.

2.1 Object-Oriented System

Object-Oriented systems have become very popular nowadays because of their reusability and ease of the maintenance. They behave differently from traditional systems and are characterized by terms like objects, classes, encapsulation, inheritance, polymorphism, message passing and patterns. Accordingly, OO system development process have different requirements and procedures. All these key concepts, characteristics, requirements and procedures are discussed in the following subsections.

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.

Canada

PERFORMANCE ANALYSIS OF CONCURRENT OBJECT- ORIENTED SYSTEMS

by

Jennifer J. Liu

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

Ottawa-Carleton Institute for Electrical Engineering
Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University,
Ottawa, Ontario, Canada

December 9, 1994

© copyright
1994, Jennifer J. Liu

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

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 0-315-98604-2

Canada

Name JENNIFER J. LIU

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.

Computer Science

SUBJECT TERM

0984

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 0272
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 0523

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

LANGUAGE, LITERATURE AND LINGUISTICS

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

PHILOSOPHY, RELIGION AND THEOLOGY

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

SOCIAL SCIENCES

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

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
Ethnic and Racial Studies 0628
Individual and Family Studies 0629
Industrial and Labor Relations 0630
Public and Social Welfare 0700
Social Structure and Development 0344
Theory and Methods 0709
Transportation 0999
Urban and Regional Planning 0453
Women's Studies

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

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

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
Atmospheric Science 0798
Atomic 0759
Electronics and Electricity 0609
Elementary Particles and High Energy 0752
Fluid and Plasma 0611
Molecular 0756
Nuclear 0463
Optics 0346
Radiation 0984
Solid State 0984
Statistics

Applied Sciences

Applied Mechanics 0346
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

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

*Performance Analysis of Concurrent Object-
Oriented Systems*

submitted by Jennifer J. Liu, B.A.
in partial fulfillment of the requirements
for the degree of
Master of Information and System Science

Dr. D. C. Petriu, Thesis Co-Supervisor

Dr. J. P. Corriveau, Thesis Co-Supervisor

Chair, Department of Systems and Computer Engineering

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
December 9, 1994

Abstract

The focus of this research is on exploring the performance of *Concurrent Object Oriented Systems*. First the thesis investigates the performance characteristics of Object-Oriented systems and analyzes the key OO factors that have great impact on system performance. Then a set of guidelines and techniques that can improve OO system performance in general are proposed to offer designers guidance for better system performance.

An example simulation model for a real-time OO system is built which includes the competition between various objects for hardware and software resources. Special attention was given to represent in the simulation model the OO system behavior, functionalities, workload requirements, various OO related overheads, and to capture OO performance statistics

The thesis also identifies and analyzes communication patterns and mechanisms between objects in order to incorporate them into an analytical performance model. Since many object interactions are of a client-server type with nested synchronized requests, the Stochastic Rendezvous Network (SRVN) model is the starting point for building a performance model tailored for OO systems. Such a model is constructed in the thesis for the real-time OO system used as a simulation case-study in the earlier chapters. The model uses objects as modelling building blocks and covers the object behaviors, concurrency among objects, contention requirements as well as multiple types of messages. Notations for different message types are introduced and ways to include OO related performance overheads into the model parameters are explored.

*To my parents, especially my mother,
for the valuable training and education
given to me during my childhood.*

Acknowledgements

First, I would like to express my deep gratitude to my supervisor, Professor Dorina Petriu for the invaluable support and guidance she has given me during the past two years. Her patience, understanding and encouragement have given me enduring inspiration throughout the evenings and weekends in completing this thesis on a part-time basis. I am sure her motivation will have a long-lasting impact on my future endeavour too.

I would also like to thank my co-supervisor Professor Jean-Pierre Corriveau from School of Computer Science for his timely guidance and suggestions throughout the research.

Special thanks to my husband, Patrick, for his understanding and encouragement he has shown during this period of time. Without his consistent support, I could have never gone so far and been so strong.

Table of Contents

Abstract.....	iii
Acknowledgements.....	v
Table of Contents.....	vi
List of Figures.....	x
List of Tables.....	xiv
1. Introduction.....	1
1.1 Motivations.....	1
1.2 Content Outline.....	4
2. Background.....	6
2.1 Object-Oriented System.....	6
2.1.1 Key Concepts.....	7
2.1.2 OO System Characteristics.....	8
2.1.3 Object-Oriented Development Process.....	11
2.1.3.1 OO Analysis.....	11
2.1.3.2 OO Design.....	13
2.1.3.3 OO Implementation, Verification and Testing.....	13
2.2 Concurrent Object-Oriented System (COOS)	14
2.2.1 Introduction.....	14
2.2.2 Types of Concurrency.....	15
2.2.3 Identifying Concurrency in OO System	18
2.3 Software System Performance Modelling.....	19

2.3.1 Queueing Network Model	21
2.3.2 Petri-Net Model	22
2.3.3 SRVN Model.....	24
2.3.4 Software Performance Engineering Model	25
2.3.5 Three View Models.....	27
2.4 Summary.....	29
3. Performance Study of Object-Oriented System.....	30
3.1 Performance Characteristics of OO Systems.....	30
3.2 Key Factors and the Implementation.....	32
3.2.1 Inheritance	33
3.2.2 Polymorphic Binding.....	36
3.2.3 Context Switching	38
3.2.4 Messaging.....	40
3.2.5 Other Factors	43
3.3 Techniques for Improving OO System Performance.....	44
3.3.1 OO Incremental Performance Monitoring Plan.....	44
3.3.2 Guidelines for Better Performance	45
3.4 Summary.....	54
4. Case Study on a Real-Time Concurrent OO System.....	55
4.1 Automatic Conference Call System (ACS) Description	55
4.2 OO Analysis and Design of ACS Application	57
4.2.1 Call Processing Activities.....	57
4.2.2 The ACS Object Model	59
4.2.3 Mainline Use Cases and Their Execution Scenarios	62
4.3 Summary.....	67

5. ACS Simulation Model	69
5.1 About the Simulation.....	69
5.1.1 Introduction.....	69
5.1.2 Motivation for ACS Simulator	70
5.2 Using PARASOL to Model OO System.....	70
5.3 ACS Simulation Model	73
5.3.1 ACS Model Component Description and Behavior	73
5.3.2 Assumptions	80
5.3.3 Delay and Overhead Representation	81
5.3.4 Performance Statistics Capturing and Description.....	83
5.4 Detailed Simulation Model Behavior	85
5.4.1 Object Creation.....	85
5.4.2 Object Interaction Diagram	87
5.4.3 Execution Sequences	91
5.5 Performance Analysis.....	99
5.5.1 Response Time Behaviors	99
5.5.2 Software Bottleneck.....	101
5.5.3 OO Related Delay Behaviors.....	105
5.6 Summary.....	107
6. Abstract Performance Modelling of Concurrent OO System	108
6.1 Performance Modelling Requirements	109
6.1.1 Object Behaviors	109
6.1.2 Concurrency.....	111
6.1.3 Resource Contention Among Objects	111
6.1.4 Message Request Types.....	112

6.2 SRVN Modelling Capabilities	118
6.3 OO Performance Model.....	120
6.3.1 Model Description	120
6.3.2 OO Performance Model for ACS Application	125
6.3.3 OO Overhead Representation.....	127
6.4 Summary.....	129
7. Conclusions.....	130
7.1 Research Summary and Contributions	130
7.2 Future Work	131
References.....	133
Appendix Notations.....	139
A.1 TimeThreads (Buhr)	139
A.2 OMT Notation (Rumbaugh)	140
A.3 Context Diagram	140
A.4 State Transition Diagram	141
A.5 Message Sequence Charts (MSCs)	141
A.6 Execution Graph Notation (Connie Smith)	142

List of Figures

Figure 2.1. An Object's Internal State	7
Figure 2.2. Asynchronous Message Passing.....	9
Figure 2.3. Typical Steps in Object-Oriented Analysis	12
Figure 2.4. Threads Among Objects.....	16
Figure 2.5. Object State Concurrency.....	17
Figure 2.6. Control Changes Within an Object.....	18
Figure 2.7. Data Flow in Performance Analysis.....	20
Figure 2.8. Closed QN Model.....	22
Figure 2.9. Precedence Graph of a Concurrent System... ..	23
Figure 2.10. Typical Petri-Net System.....	23
Figure 2.11. Example of SRVN Model.....	25
Figure 2.12. Example Execution Graph for Entering Marks Scenario	26
Figure 2.13. MPR Framework	27
Figure 2.14. Different Views in the MPR Model.....	28
Figure 3.1. Object Implementation in C++.....	34
Figure 3.2. Inheritance Implementation in C++	35
Figure 3.3. Polymorphism Implementation in C++.....	37
Figure 3.4. Objects Mapped to Operating System Processes	39
Figure 3.5. Context Switch Due to Messaging	41
Figure 3.6. Messaging Between Objects Residing on Different CPU	42
Figure 3.7. Messaging Between Objects Residing on Same CPU.....	43
Figure 3.8. OO Incremental Performance Monitoring Plan	46

Figure 3.9. Add Direct Object Association.....	48
Figure 3.10. Reduce Code Inheritance Level	49
Figure 3.11. Rearrange Execution Order	50
Figure 3.12. Localization of Information.....	51
Figure 3.13. Cloning an Object.....	53
Figure 4.1. Call Processing Sequence Chart.....	58
Figure 4.2. Inheritance Hierarchy of ACS	60
Figure 4.3. Object Model of ACS Application	63
Figure 4.4. Message Sequence Chart for Conference Call.....	65
Figure 4.5. Message Sequence Chart for Regular Call.....	66
Figure 4.6. Conference Call State Transition Diagram.....	67
Figure 5.1. Object Model of ACS Application	74
Figure 5.2. Implemented Call Processing Components.....	75
Figure 5.3. Attributes for User and Call classes	77
Figure 5.4. Attributes for Server class	78
Figure 5.5. Attributes for Resource Class.....	79
Figure 5.6. Object Creation Process	86
Figure 5.7. Message Passing During INIT & ADD in Conference Call	88
Figure 5.8. Message Passing During TALK in Conference Call.....	89
Figure 5.9. Message Passing During TERM in Conference Call	90
Figure 5.10. Message Passing During INIT in Regular Call	92
Figure 5.11. Message Passing During TALK in Regular Call.....	93
Figure 5.12. Message Passing During TERM of Regular Call.....	94
Figure 5.13. Timethread for INIT & ADD in Conference Call	95

Figure 5.14. Timethread for TERM Scenario in Conference Call.....	96
Figure 5.15. Timethread for INIT & ADD Scenarios in Regular Call	97
Figure 5.16. Timethread for TERM Scenario in Regular Call.....	98
Figure 5.17. Response Time Behavior for Conference Call Client	100
Figure 5.18. Response Time Behavior for Regular Call Client	101
Figure 5.19. Software Bottleneck	102
Figure 5.20. Software Bottleneck Behavior in Conference Call Scenario	102
Figure 5.21. Software Bottleneck Behavior in Regular Call Scenario	103
Figure 5.22. Utiliz. & Response Time Change with Cc_server Cloning Level.....	104
Figure 5.23. Utilization Change with Cp_server Cloning Level	105
Figure 5.24. Response Time Change under OO Performance Overheads.....	106
Figure 6.1. Example Object Interaction Diagram.....	109
Figure 6.2. Object Contention.....	112
Figure 6.3. Request Type I: Synchronous Rendezvous	113
Figure 6.4. Request Type II: Synchronous Rendezvous Phase I	114
Figure 6.5. Request Type III: Non-blocking Message Sending Request.....	115
Figure 6.6. Request Type IV: Message Forwarding.....	116
Figure 6.7. Request Type V: Tentative Command Message	117
Figure 6.8. A Mapped SRVN Model	119
Figure 6.9. Notation for Synchronous Rendezvous Phase I Message Type	122
Figure 6.10. Notation for Synchronous Rendezvous Message Type	123
Figure 6.11. Notation for Non-blocking Message Sending Request Type	123
Figure 6.12. Notation for Message Forwarding Request Type	124
Figure 6.13. Notation for Tentative Command Message Request Type	124

Figure 6.14. OO Performance Model of ACS Application	126
Figure 6.15. Represent OO related overheads in OO Performance Model	128
Figure A.1. Timethread Notations	139
Figure A.2. Object Model	140
Figure A.3. Context Diagram	140
Figure A.4. Dynamic Model	141
Figure A.5. Message Sequence Chart Notations	141
Figure A.6. Execution Graph	142

List of Tables

Table 1: Example OO Languages and Their Characteristics	31
Table 2: PARASOL Hardware Components.....	71
Table 3: Performance Overhead Statistics Collected from ACS Simulation.....	106

Chapter 1

Introduction

1.1 Motivations

The Object-Oriented approach in developing software system is widely spread in the software community because of its notable advantages in design and implementation of computer software. Among the most important advantages are: comprehensibility, maintainability and reusability. Many Object-Oriented Systems are under construction and many Object-Oriented books, papers and journals have been published recently. These materials cover many areas of the Object-Oriented System development, such as analysis, design, implementation, testing and OO languages, but often neglect one important yet difficult issue: system performance.

With the increased popularity of OO systems, more and more designers have become aware of its downside as well. The OO system's execution has often been intrinsically inefficient due to excessive overhead. This problem of performance degradation has become the major obstacle in widespreading the OO technology in industry, especially for real-time critical systems.

As we know, Object-Oriented paradigm provides us with a new way of thinking about problems by using models organized around real-world concepts. The fundamental construct in OO systems is the object, which combines both data structure and behavior in a single entity. OO systems are also characterized by a set of OO specific properties such as encapsulation, inheritance, polymorphism and messaging, etc. which are the major sources of performance overheads. All of these properties and characteristics are quite different from those of traditional systems. This means that various performance methodologies developed for traditional systems such as perfor-

mance modelling techniques and measurement strategies have to be adapted to the particularities of Object-Oriented systems.

Basically, the thesis investigates the following problems related to OO system performance:

1). Measurement and Metrics:

What kind of OO specific metrics and statistics need to be measured and maintained to give us insight about the system's performance under study? How can they be captured?

2). Overheads:

What kind of OO specific performance overheads exist in system? How can we reduce their effects?

3). Modelling:

Can we establish an analytical performance model for an OO system and use this model to compute and predict system performance? Can we have some systematic methods to capture OO system workload parameters for this model?

4). Prevention:

What are the strategies that can improve an OO system's performance? Can we find general guidelines on how OO systems should be designed?

Some research work has been done in OO measurement and metrics by various researchers and companies. Measurement and metrics in OO system serves various purposes. They can help the designer keep track of the technical process used to develop an OO system, control the ultimate system performance behavior and fine-tune the system. The diversified perspectives on measurement combined with the complexity of the product and process being measured make the classification of metrics extremely difficult. As Vaishnavi points out [76], a single metric can be a valid estimation for a number of different attributes that are classified as both product or process factors. Also, some metrics are purely post facto descriptors of product or tracking variables for process, while others are primarily useful as predictors. Finally, a single metric in an OO system is frequently applicable to multiple phases of the development cycle. Vaishnavi and his team in [76] classify OO metrics along four dimensions, they are *product*, *process*, *size/complexity* and *applicable phase*. They also present a set of important OO metrics such as *Average Method Size*, *Average Number of Methods*, *Depth of Inheritance Tree*, *Lack of Cohesion in Method*, *Number of*

Classes, Reuse, and Weighted methods per class etc. Chidamber and Kemerer in [8] also present six design metrics and show how these metrics can be collected. But none of these metrics are performance metrics.

Some other researchers are working on inventing a new OO architecture, for example, Hufnagel and Browne put their focus on answering the fourth category of questions in [34]. They think that much of the execution overheads of Object-Oriented systems have been due to the implementation structure, that is, the objects are built upon a conventional layered software structure provided by the conventional layered operating system and database execution environment, and most of OO system virtues are obtained on this conventional computer architectures at the cost of performance and efficient use of resources. They propose a vertically partitioned structure for design and implementation of Object-Oriented systems to improve system performance, that is, implement objects through *extended type managers*. An *extended type manager* not only implements the *functionality* which defines a class but also implements all *state management functions* such as storage, access control and consistency management (including concurrency control, error detection and fault recovery) for the occurrence of that type so that implementation of type functionality and state management are integrated in a single context. In this case, the performance is improved due to two key design concepts: object semantics can be used in the state management functions of an object type and atomicity is maintained at the type manager boundaries providing efficient recovery points [34].

While some research has been conducted for category 1) and 4) above, those questions in category 2) and 3) about OO performance analysis and modelling remain less studied, which becomes the motivation of this research. The thesis studies the performance characteristics of OO systems and identifies those key OO performance influential factors, then explores the OO specific modelling requirements and finds ways to represent the OO related overheads, and finally proposes an analytical performance model based on SRVN.

The other important aspect that the thesis investigates is the *concurrency issue* in OO systems. Since independent, self-contained objects can be executed simultaneously, *Concurrent Object-Oriented Systems (COOSs)* emerge as a natural extension of Object-Oriented systems. Because of their potential modelling power for comprehensive systems, concurrent OO systems have attracted a lot of attention recently from research community, so it is also a new research area that is currently being explored. The thesis approaches the concurrency issue from the OO

system performance perspective and tries to find an analytical model that covers concurrent OO system modelling requirements and typical system behaviors.

In summary, the thesis has the following objectives:

- 1). Explore OO system performance characteristics, find key OO performance influencing factors and analyze their relative effect to the overall system performance.
- 2). Identify some general guidelines and techniques that can alleviate OO performance degradation and improve its performance.
- 3). Find a way to represent OO system behavior and those OO related performance overheads in a simulation program. Study how these overheads and related statistics can be captured and analyzed.
- 4). Investigate the specific requirements for modelling OO system performance and set up an analytical performance model if possible. Also identify ways for obtaining workload parameters that are needed for the performance model.

1.2 Content Outline

Chapter 2 provides key concepts and background information related to the thesis: Object-Oriented system concepts and development. Concurrent Object Oriented systems and performance modelling of computer systems.

Chapter 3 investigates the specific performance characteristics of the Object-Oriented Systems and key performance factors that have significant impact on system performance; it also proposes a set of guidelines that can improve OO system performance.

Chapter 4 presents a process for Object-Oriented analysis and design of real-time systems. Various important analysis models such as object model and scenario model are built and evaluated. Classes, their responsibilities and collaborators of the system are analyzed to illustrate the process of identifying classes and associations.

Chapter 5 explores how to represent OO system behaviors and functionalities in a simulation model. The chapter investigates how OO specific and non-OO specific performance over-

heads and statistics can be set up and captured properly in the simulation model. The chapter also carries out performance analysis based on statistics results collected from various simulation experiments.

Chapter 6 studies the general modelling requirements and behaviors of typical OO systems and explores the possibilities of modelling an OO system by using available analytical performance models. The chapter describes the new requirements for OO models and shows how OO related performance overheads can be represented in the performance model. The chapter also presents how workload parameters of OO performance model can be obtained and how the performance model could be potentially solved by various methods.

Chapter 7 concludes the study, summarizes the findings and future work.

Chapter 2

Background

This chapter summarizes the key issues and concepts relevant to this thesis. In the first section, key OO system concepts and characteristics are described, followed by a discussion about OO development process. Then, Concurrent OO Systems are discussed as natural extension of OO systems. The text focuses on clarifying various types of concurrency and their distinctive behaviors. Finally, issues related to software performance modelling are examined, several important performance models and their merits are summarized. Various design and performance modelling notations are used in this chapter, detailed descriptions of these notations can be found in Appendix A. Also since the chapter covers more than one field in computer science, for some terms and concepts used in Computer System Performance and Operating Systems, the chapter simply gives references to the bibliography instead of giving detailed definitions.

2.1 Object-Oriented System

Object-Oriented systems have become very popular nowadays because of their reusability and ease of the maintenance. They behave differently from traditional systems and are characterized by terms like objects, classes, encapsulation, inheritance, polymorphism, message passing and patterns. Accordingly, OO system development process have different requirements and procedures. All these key concepts, characteristics, requirements and procedures are discussed in the following subsections.

2.1.1 Key Concepts

An Object-Oriented system is essentially a society of cooperating objects [14]. The concept of *object* emerged simultaneously and almost independently from various domains in the 70's, where various perspectives on objects were formed. For example, persistent data, predefined access, aggregation and classification concepts are from the database domain; abstraction, inheritance, private methods on data aspects are from the programming languages domain; protected resource, restricted and permitted access concepts come from the operating system domain; and the modelling of physical and conceptual entities is from the simulation domain.

Formally, an *object* is defined as an abstraction of a real-world entity. It consists of both private data and a set of methods which performs on these data. These methods define what the object can do or behave, and are often referred as the object's responsibilities or the set of services it can offer.

Objects interact with each other through messaging only. A message is sent for various purposes, such as requesting for services, sending control messages and processing results, etc. An object can be active or passive. An *active object* is the object that is executing. A *passive object*, on the other hand, is in non-executing state. Since objects are only activated by message passing, internally an object can be seen as a set of possible states and set of allowable actions (methods) on the states as shown in Figure 2.1.

Figure 2.1. An Object's Internal State

In Object-Oriented Systems, software is modelled and implemented as collections of cooperating objects. Each object is an instance of a *class*, which is a generic specification for an arbitrary number of similar objects. Classes are organized in hierarchies which model the generalization and the specialization of objects in the problem domain. An class provides a way for software to define attributes and behaviors for a group of objects in one step.

2.1.2 OO System Characteristics

OO systems are distinguished from traditional systems through a set of common characteristics that are well-defined and generally accepted by the industry and the research community. Typical characteristics include encapsulation, classification and instantiation, inheritance, polymorphism, patterns and reusability.

Encapsulation means grouping and information hiding. An object groups and hides its private attribute data and a set of methods from outside access, and only presents a public interface to react to external messages. Other objects may only change this object's states by sending it messages according to the defined interface.

Classification and Instantiation: Objects are grouped into classes and every object is an instance of a class. All instances of a class share a common definition and format, namely common attributes and common methods, but the actual value of attributes might be different. A class is a template for generating instances that represent real world entities.

Polymorphism means the ability to take several forms [44]. Multiple classes can each provide a method with the same name, the client object that invokes the method does not need to know the precise class of object to which it is sending a message, and it is up to the receiver object to supply the correct method. Polymorphism provides a great way to reduce code, and provides a cleaner program interface. It also facilitates the future design process so that designer only has to provide code to communicate with common interface.

Inheritance is a mechanism for generalizing the common parts of multiple classes. It allows for the reuse of software by specializing in existing general solutions. Both attributes and methods can be inherited. The inheritance hierarchy captures all inheritance relationship among classes. There are two kinds of inheritance: *single inheritance* and *multiple inheritance*. *Multiple inherit-*

ance refers to the situation where a class has more than one superclass and thus inherits features from all parents, while for *single inheritance*, a class has only one superclass and the class hierarchy is a tree structure. As Rumbaugh points out in [60], multiple inheritance has greater power in specifying classes and an increased opportunity for reuse at the cost of conceptual and implementation simplicity.

An *message* is the unified communication and coordination protocol between objects. Objects interact with each other by sending, receiving and processing the message requests. When a passive object receives a message from a sender, it will potentially become an active object and process the message request. For a single object, only one message can be processed at a time. Message passing can be synchronous or asynchronous. *Synchronous message passing* refers that message sender blocks and waits until the receiver object accepts then processes the message, and finally returns the answer back to the sender which may then resume its computation [14], while *asynchronous message passing* refers to messaging with no waiting, that is, computation of message sender proceeds immediately after the message is sent to receiver, receiver object will then execute concurrently with the sender object.

The messages are *buffered* and potentially asynchronous [14] because an active object may be busy computing when a message reaches it and only one message can be processed at a time. As shown in Figure 2.2, when object A sends a message to object B, B is busy at that moment and

Figure 2.2. Asynchronous Message Passing

not able to receive the incoming message, so the incoming message gets buffered while waiting for acceptance. After object B finishes the current processing, it will take a message item from the queue and process it. Buffering messages also allows sorting message requests according to priorities, it frees sender from unnecessary waiting for reply by asynchronous transmission, thus desynchronize message sending, receiving and processing.

A *pattern* is a combination of a set of classes and objects as well as their relationships. A pattern can serve as a generic structure or building block in OO system. Once a set of patterns are identified, they can be reused later in similar cases. Some examples of patterns include the client-server pattern, the master-slave pattern, the inter-communication pattern, etc.

Generally speaking, a pattern can be either structural or behavioral. A *structural pattern* is a static pattern that shows structural inter-relationship among objects. It often occurs in layered architectures. For example, in the popular OSI model, seven layers are present, and every layer demonstrates a similar pattern: information is transmitted from source to destination through a channel (whether abstract or physical). On the other hand, a *behavioral pattern* is a dynamic pattern that shows the behavioral change sequences of objects in the system. It also has many applications in real world applications. For example, all telephone calls have a similar execution sequences in a switch: allocating resources, setting-up a call, translation, routing, talking, hang-up, deallocating resources, etc.

Reusability is one of the greatest advantages offered by OO systems. Reusable software reduces design, coding and testing costs by amortizing effort over several designs [60]. Reducing the amount of code also simplifies understanding, which increases the likelihood that the code is correct. There are two kinds of reuse: sharing of newly-written code within a project and reuse of previously-written code on new projects. Reuse is possible in conventional languages too, but OO languages greatly enhance the possibility of code reuse through the set of OO characteristics, such as inheritance, polymorphism, organizing for independence, etc.

Reusability is also closely related to system generality and extensibility. *Generality* is the ability of one system to serve in a large range of circumstances without modification. *Extensibility* refers to the ability of a system to be easily adapted to better meet a particular need, and to respond to changes easily (eg. hardware or software changes, system upgrades etc.).

Reusability can be achieved through two stages: producing reusable software components and reusing them in later development, where a reusable component is a fully tested, relatively independent piece of code with clear interface defined. A carefully designed and implemented OO system, which follows OO system principles, satisfies the necessary OO system requirements and possesses all key OO system characteristics, can potentially become a reusable system.

2.1.3 Object-Oriented Development Process

OO development process is an iterative process in which objects are gradually refined [12]. In a way, it can be viewed as a simulation process where real world objects are modelled with a computer analogy [11] and OO applications can be seen as simulations of their problem domain in the real world. Different methods on how to perform OO requirement and domain analysis as well as OO design within each increment can be found on various OO literature such as [28][36][60]. The following subsections summarize the major steps in OO development process: analysis, design, implementation, verification and testing.

2.1.3.1 OO Analysis

OO Analysis is the first step in an OO development process. The objective of OO analysis is to capture the requirements and to obtain the structural or behavioral sequences of the intended system through analyzing the problem domain. OO analysis models (OOA) are usually refinements of existing techniques, such as entity relationship diagram, state transition diagram etc.

Important analysis modelling strategies used in OO systems include structural modelling, scenario modelling, behavioral modelling and use case modelling. *Structural modelling* focuses on modelling the components of the intended domain, namely, objects, classes and their relationships. The deliverable of *Structural Modelling* is usually called *Domain Object Model*. Frequently used techniques include: ROOM methodology [62][63], Object Modelling Technique(OMT) (by Rumbaugh et al. [60]). A structural model is especially suitable for system domain analysis.

Scenario modelling captures the dynamic interactions among objects for every possible scenario. The frequently used methods for scenario modelling include Timethreads [16][18] and message sequence charts (MSCs)([19][20]). A timethread traces the global dynamic flow of causality through the components of the system of a specific use case, while message sequence chart

records a sequence of messages and events occurred during a particular scenario.

A scenario is usually obtained by conducting a use case analysis. A *use case model* consists of a set of use cases, system and the external objects. A use case represents a related sequence of transactions. Jacobson et al. in [36] described thoroughly how to apply the use cases into OO modelling process. The high level use case model serves both for understanding scenarios and later for system model verification and testing purpose.

Behavioral modelling focuses on the system internal state changes, the events triggers and the processing occurred. The major tools used include state machine chart[19] and the ROOM Finite State Machine (FSM) model from ObjecTime Company (detailed description can be found in [62][63]). ROOM FSM Model is a combination of traditional-state-diagram, OO concepts and telecommunication characteristics, it is more appropriate for modelling behaviors of real-time OO systems, e.g. telecommunication systems.

Among the modelling methods, structural modelling is definitely the driving force since it defines abstraction and captures the essence of the system, but it is not sufficient by itself, it needs to be combined with other modelling techniques such as scenario modelling to provide an integrated dynamic view of the system. Figure 2.3 illustrates some typical steps involved in Object-Oriented Analysis [11].

Figure 2.3. Typical Steps in Object-Oriented Analysis

2.1.3.2 OO Design

Analysis models only records the requirements expressed in the problem statement, and it is up to design models to give a particular solution to these requirements. Object-Oriented Design (OOD) considers the engineering constraints for the underlying system and produces a particular language independent solution for the above analysis models. It is an iterative and refining process, usually the OO system is built as a series of working prototypes that are incrementally refined into the final product. Typical steps involved in OO design are as follows ([11][12]):

- define the system boundary and identify classes that are going to be implemented.
- revisit and refine CRC (class-responsibility-collaborator) card to further clarify class responsibilities. Some previous identified classes might be replaced or consolidated with new implementation classes.
- organize objects into subsystem or processes and consider data storage.
- identify concurrency.
- design object data manager (the subsystem which manages object creation, deletion and changing object state information) and control architecture, allocate and handle global resource.
- determine trade-offs, set priorities and choose a suitable architecture accordingly.

2.1.3.3 OO Implementation, Verification and Testing

During the implementation stage, the design will be mapped into an OO programming language, ideally, this mapping should be automated through automatic code generation, but currently there is often no direct support from a case tool. Correctness, efficiency and reusability are the focus in this stage.

In verification and testing stage, use cases identified in the analysis stage will be evolved as test cases for testing. The goal is to maximize test coverage with minimal test cases. Another aspect with OO program testing is that since the program is organized not by algorithm, but by objects, the execution thread is not sequential any more, rather objects are activated concurrently and dynamically which makes message testing a crucial part.

2.2 Concurrent Object-Oriented System (COOS)

2.2.1 Introduction

Concurrency is one of the important issues that must be properly addressed in many application domains, such as transaction processing, embedded software, process control and telecommunications, since the presence of concurrency may introduce potential problems such as deadlock, starvation, race conditions and critical sections. Two or more processes or execution threads are concurrent when they are active simultaneously disregarding processor allocation. If two or more concurrent processes share the same processor, the user can be given illusion that multiple processes are running at the same time, although at any time, only one process is actually executing. The underlying operating system makes some scheduling mechanisms run them one by one, however, the processes are still *concurrent*, and the operating system treats them as different execution units. Concurrent processes can communicate or synchronize with each other through various mechanisms.

Distributed systems and parallel systems are typical examples of concurrent systems. For example, in a travel claim processing system, there are several different servers used in different stages which are parallel to each other. Multiple claims could be waiting to be processed. When one claim is running on one processing node, the other one can be running concurrently in other processing node. We say these claim processes are executed concurrently.

If the software to be developed is to react to the real world, concurrency and real-time issues are important. Therefore for Object-Oriented systems, in order to enable objects and classes to properly model all relevant characteristics and behavior of the application domain entities, their concurrency and timing properties should be addressed.

In OO system, independence and self-containedness of objects lead naturally to envision their simultaneous activities [14]. Concurrent Object-Oriented Systems (COOS) emerge as a natural extension of object-oriented system. They combine characteristics of both OO and concurrent systems. Concurrent OO systems can help to decompose large programs into a set of small collaborating objects and to exploit new parallel machine architectures, further exploring object concurrency behavior in COOS becomes a very important issue. In the next sections, types of

concurrency are summarized. How to identify concurrency in OO system is also discussed.

2.2.2 Types of Concurrency

In traditional systems, a program executes according to certain built-in algorithms and the process is the basic executing unit. An Object-Oriented system is very different from a traditional system in that the object becomes the basic executing unit. An OO system is basically a collection of objects which interact with each other to render changes to the system state. Furthermore, objects are divided into two categories: active and passive. When a passive object receives message from a sender, it will potentially become an active object in future. The result of these fundamental paradigm changes affect many other system characteristics, for example, the concurrency concept.

First, in Concurrent OO System, concurrency can be real concurrency or virtual concurrency depending on the actual number of processors involved. *Real concurrency* means that two or more objects in the system are executing on different processors at the same time. *Virtual concurrency*, on the other hand, refers to the situation where two or more objects are actually running on the same processor. In the second case, only an object is actually active and consumes cpu time, while the other objects are either waiting for I/Os or in the ready queue.

Secondly, concurrency has been used in a broader sense in OO system. It could refer to the situation when two objects are existing at same time, or when two objects have executing threads at the same time or when two objects have internal concurrent activities, or when the objects are distributed on different processors. These types of concurrency can be categorized accordingly as follows:

Object Concurrency (Aggregation Concurrency)

Object-Oriented systems mimic the real world objects, their interactions and relationships. Many objects exist in the memory at the same time. They send messages to each other, process the messages received, coordinate and share the system responsibilities together, and therefore are concurrent. We name this *object concurrency* (referred to as *aggregation concurrency* in [60] by Rumbaugh) or say the two objects are *objectwise-concurrent*.

A dynamic model shown in Figure 2.4 describes a set of concurrent objects, each with its

Figure 2.4. Threads Among Objects

own state information. Object A, B, C are allocated on processor 1, object D, E, F are allocated on processor 2, and B and D are currently executing. All the objects A, B, C, D, E, F are objectwise-concurrent in this case and can change their states independently. The state of the entire system cannot be represented by a single state in a single object, rather it is the cartesian-product of the states of all the objects in it. In many systems, the number of objects can change dynamically as well, which causes system state changes too.

Although all objects are conceptually concurrent, they are, in many situations, inter-dependent. During the implementation stage, many objects can often be folded together into a single thread of control and the actual event of flow among objects is often sequential, which means object concurrency between two objects could be either virtual concurrency or real concurrency. The last happens when the objects are residing on different processor and can be active at the same time.

State Concurrency

Concurrency within the state of a single object arises when the object can be partitioned into

subsets of attributes or links, each of which has its own subdiagram [60]. The aggregate state corresponds to the combined states of all the component diagrams as shown in Figure 2.5. for state 2.

Figure 2.5. Object State Concurrency

This type of concurrency is referred to as *state concurrency*. An object that has state concurrency is actually a multi-threaded object.

State concurrency is a virtual concurrency too since one object can only reside on one processor, and like object concurrency, in any time there is only a single thread of program control existing in the system. Figure 2.6. shows the possible control changes within an object, in Figure 2.6. (a) control is split into two subcontrol threads. For example one object can have multi-threaded controls at the same time, thus is able to send multiple messages to multiple objects, causing concurrent execution of multi-tasks, while in Figure 2.6. (b), two control threads are synchronized into one. This is further illustrated in Figure 2.4 where for object B, the control thread is split into two subthreads and later synchronized.

Figure 2.6. Control Changes Within an Object

Executional Concurrency

Two objects are *executionally concurrent* if they are both executing at the same time. This type of concurrency appears when objects in the system are distributed among more than one processors, where more than one object will be active and stay in execution state. *Executional concurrency* is a real concurrency and each active object is sometimes referred to as a *Parallel Object Machine*. For example in Figure 2.4, object B and D are both active and each reside on a different processor, therefore are *executionally concurrent*.

Despite differences among these various types of concurrency, this thesis doesn't distinguish between them in general except where there is a need to do so.

2.2.3 Identifying Concurrency in OO System

When it comes to identifying concurrency in OO system, researchers have different view points. Most researchers think that even though concurrency is critical, it should still be addressed following identification of objects and classes [13]. On the other hand, Shumate thinks concurrency is critical and must be dealt with early [28], that is, addressing concurrency first and then the object.

Firesmith does not agree with any of these viewpoints [28] and thinks that “addressing objects first without consideration of concurrency and timing issues is also somewhat dangerous. Additional objects (e.g., buffers) and operations (e.g., to protect the attributes) must often be added”. The best approach seems to consider both issues simultaneously, with concurrency and timing issues addressed in terms of both objects (concurrent or sequential) and messages (sequential, synchronous, or asynchronous). Thus the important issues of priority, priority inversion, deadlock, starvation, mutual exclusion, race conditions (detailed definitions can be found on [73]), etc. can be correctly handled.

Another important aspect in identifying concurrency is that once there is any concurrency in the objects, there often needs to be significantly more [28]. For example, if two concurrent objects can send messages to a third object, the third object will probably have to be concurrent. Otherwise, one or more attributes of the third object may be accidentally corrupted due to interleaved access. For example, one operation may only partially update the attribute before its time slice (refer to [73] for definition) is completed and the second operation may then start with a partially updated attribute.

2.3 Software System Performance Modelling

Software system performance analysis should be an important part of the software development process because it can help to ensure that the new system satisfies both the functional and performance requirement. Also it helps to evaluate the various system resource usages, to predict potential system behavior and most importantly, to compare different design choices and hardware arrangements and to identify trade-offs.

Three broad areas exist in the computer system performance area: measurement, simulation and analytic modelling. Depending on the stages of software development, different methods can be applied.

Measurements can be done when the system is built and instrumented. Typical measures include response time, throughput, utilization, cycle time (detailed definitions can be found on [70]), etc. The specific performance measures of interest are dependent on the types of the system. For example, the response time is more crucial than other performance measures in a real time system, whereas throughput is a better description of performance for a batch processing system.

Prior to the construction of the system, in order to predict system performance, either simulation or analytic modelling techniques can be used. *Analytic models* are abstract and simpler representation of the system under study, they can predict system performance with relatively low cost and high speed. On the other hand, *simulation* takes longer and costs more but models the system at a more detailed level and gives more accurate performance results.

For software system performance analysis, some important factors are: the functional design and its correctness, processing load and sequence, hardware capability, resource scheduling and management, execution time and frequency, processor structure and scheduling (see [70] for relevant terminology definitions) etc. Figure 2.7 below shows the data flow during performance analysis.

Figure 2.7. Data Flow in Performance Analysis

Many analytic models exist which can be used for evaluating and comparing performance characteristics of software designs. Some provide the high level framework for generic performance data gathering and computation ([80]); while others focus on the lower system level model construction for studying a specialized area of behavior ([41][42][70]). Several important performance models are summarized in the following sections.

2.3.1 Queueing Network Model

Queueing Network (QN) Model is a powerful analytical model for evaluating system performance ([41][42]). A QN model is a collection of service centers (each with its own queue and servers) and clients. Every service center represents a common resource, shared by clients, and has an established discipline for serving the clients. In a classical QN model, the service centers usually represent hardware resources, clients represent software jobs. Clients are independent of each other except they all compete for service from service center, and they can be divided into different classes according to their statistical identities, such as routing probabilities, service demand etc. ([55]).

Each service center is associated with a waiting queue with multiple service levels. Each client will be competing for specific time of service from a service center. After the service, the client leaves the service center and then proceeds to another service center. The service disciplines could be first come first serve (FCFS) or round robin etc.

A QN model can be either open or closed. In an *open* QN system, clients are allowed to enter or leave freely and the number of clients in the system is not fixed, while in a *closed* system, the number of clients is fixed and no entrance or departure is allowed. Figure 2.8 is an example of a closed queueing system.

The QN model has many advantages. It provides both the modelling power and efficient solutions. The model is easy to build, highly abstract and requires minimum of information as parameters. It has been extensively used for constructing analytical computer performance models, especially for modelling time sharing systems where the jobs compete for shared resources but do not communicate or synchronize directly.

QN model has certain limitations too. It cannot represent software servers with nested service requests, and is unable to represent synchronization directly. It also has limitations in model-

Figure 2.8. Closed QN Model

ling parallel software for distributed systems.

2.3.2 Petri-Net Model

The *Petri-Net (PN) Model* is another analytical modelling method used for performance analysis of concurrent software[2] [48]. Petri-Net contains a set of places, a number of tokens that reside at the places, and a set of transitions that represent events causing tokens to move from one place to another following firing of transitions. PN is very efficient in representing flow models of concurrent software. Figure 2.9 from [55] shows a precedence graph for a system with concurrent tasks and Figure 2.10 is the Petri-Net equivalent representation of the system in Figure 2.9.

The advantages of PN are mainly its modelling power, it is able to model concurrent software at any level of detail.

The disadvantages of a PN include its weakness in modeling contention for resources and its low level of abstraction capability. It has no construct to represent queues and scheduling policies. Also the exact analysis done in Petri-Net proposes a limitation on the type of model: that can be solved (only small models). Finally, PN doesn't offer efficient solution methods: the exact solution lies on constructing Markov Chain Model and hence suffers from the well known problem of state space explosion.

Figure 2.9. Precedence Graph of a Concurrent System

Figure 2.10. Typical Petri-Net System

2.3.3 SRVN Model

Stochastic Rendezvous Network Model (SRVN) is a performance model for multi-tasking parallel software with inter-task communication via rendezvous [56], which is very appropriate to model client-server systems and in special systems with software servers. It was introduced by Woodside in [78], and further developed in [79][57][58].

SRVN model contains a set of concurrent tasks communicating through rendezvous(RV) (see Figure 2.11). The scenario is as follows: the client task sends a RV request to server and blocks waiting for reply. The requests are queued in a single FIFO input queue owned by the server. The server checks its input queue. If the queue is empty, it stays idle; otherwise, it selects the first request from a queue and starts an execution cycle which is the first phase “in RV”. After a first phase of service, the server sends a reply message back to the client which is blocked waiting for response from the server. Upon receiving the reply, the client is released from RV and resume its own execution, while the server enters second phase “post RV” and executes concurrently with the client [56][59].

The communication pattern between tasks is described by an acyclic directed graph illustrated in Figure 2.11, where parallelogram nodes stand for tasks and directed arcs represent RV requests from clients to servers. The tasks at the top are clients (T1 to T6), also called reference tasks, which drive the system by cycling endlessly and generating RV requests to the server tasks. Statistically identical clients may be grouped into different client classes. All the other tasks are servers, also called non-reference tasks. Each server has its own client sets and may offer a range of services with different workload characteristics, modelled by different entries shown as parallelogram slices in the above diagram. Especially, the tasks that are intermediate nodes in the graph (Ts1, Ts2) play a dual role as servers and clients, while the leaf nodes (Tcpu, Tdisk) are pure servers and usually model hardware resources such as cpu and disks.

SRVN provides following modelling capabilities of a system:

- **Modelling Flexibility:** Both system software and hardware components can be modelled.
- **Nested Service:** where a software server uses the services of other servers to provide the requested service to the client.
- **Clients can require any number of services from a server and change randomly the type of request to be sent to the server.**

Figure 2.11. Example of SRVN Model

- Parallel execution between clients and servers.
- Multi-Phase Behavior of Servers: The execution cycle of a non-reference task (server) is characterized by two phases of execution: an “in-rendezvous” phase (phase 1) and a “post-rendezvous” phase (or phase 2). In either phase, a non-reference task may execute on their own and/or request services from other non-reference tasks.

Approximate Mean Value Analysis (MVA) solution is introduced in several papers, as for example [58], [59], [56] and [79].

SRVN model combines some of the advantages of QN and PN models and is effective in modelling concurrent client-server system, for example, transaction processing systems.

2.3.4 Software Performance Engineering Model

Software Performance Engineering (SPE) introduced by Connie Smith ([70]), presents a systematic way to analyze the performance of software systems during their development process. It starts early in software lifecycle and uses quantitative methods to determine satisfactory designs and to eliminate those that are likely to have unacceptable performance [71].

The basic steps involved in software performance engineering are as follows: first, find representative scenarios for the underlying system and collect the important raw data; then build the software execution model. Then for every typical workload scenario, use execution graphs to perform best, average and worst case analysis. An example execution graph about *entering marks scenario* of a university marks system is shown on Figure 2.12. (see Appendix A for the set of notations related to execution graph).

Figure 2.12. Example Execution Graph for *Entering Marks Scenario*

Upon finishing above steps, data obtained from the execution graph are used to characterize

the software and hardware resource requirements and derive parameters for a system execution model. System execution models represent the key hardware devices with queue-servers, and the possible inter-connection paths among these devices by using arcs. The system execution model may be constructed using a information processing graph (IPG) which is a graphical representation of queuing networks. Solution methods for IPG can be found in [37][42][47] [61].

2.3.5 Three View Models

The *Three View Model* (also called *Map-Path-Resource Model* or *MPR model*) is a high level model proposed by Professor Woodside of Carleton University [80] for performance modeling of concurrent software. It integrates existing analytical models such as execution graphs, petri nets, StateCharts, SRVN model etc. into three categories (views) according to the performance data flow. The views are connected by means of a “Core Model” and are used together to extract information related to system integration, such as inter-process overheads, and the delay behavior of separate software components in complex systems. Figure 2.13 shows the inter-relationship among views and core.

Figure 2.13. MPR Framework

For example, Connie Smith's software execution model [70] is really a path model since it is based on different execution path or scenarios. SRVN models are map models, and QN models are resource models since the major activities in QN is resource monitoring.

The integration of views is facilitated by converting results in one view (e.g. scheduling delay for resources) to those in another (such as delays along a path). Figure 2.14 illustrates possible information generating and converting sequences within the model.

Figure 2.14. Different Views in the MPR Model

2.4 Summary

Object-Oriented system is essentially a society of cooperating objects. An *object* is defined as an abstraction of a real-world entity. It consists of both private data and a set of methods that perform on these data. Objects interact with each other through *messaging* only, message passing is buffered and potentially asynchronous. Object can be *active* or *passive*. Each object is an instance of a *class*. Typical OO systems characteristics include *encapsulation, classification and instantiation, inheritance, polymorphism, patterns and reusability*.

OO development process is an iterative process in which objects are gradually refined. Important analysis modelling strategies used in OO systems include *structural modelling, scenario modelling, use case modelling and behavioral modelling*.

In OO system, independence and self containedness of objects lead naturally to envision their simultaneous activities. Concurrent Object-Oriented System (COOS) emerges as a natural extension of object-oriented system. Managing concurrency and synchronization of message handling within an object is an important issue in Concurrent OO systems. In Concurrent OO Systems, concurrency can be real concurrency or virtual concurrency depending on the actual number of processors involved. It can also be categorized as object concurrency, state concurrency and execution concurrency. Concurrency should be addressed at the same time as object identification.

Software system performance analysis should be an important influencing part of software development process. There are three different ways to estimate the performance: *by measurements, by simulation and analytic modelling*.

Concurrency in OO Systems is a new research direction which has become increasingly important for modelling real-time and concurrent applications. At the same time, applying software performance modelling techniques on Concurrent OO System to predict and fine-tune system performance has become even more crucial.

Chapter 3

Performance Study of Object-Oriented System

This chapter concentrates on the performance study of OO systems. Since OO systems have a broader meaning than Concurrent OO Systems, everything presented in this chapter will also be applicable to Concurrent OO Systems. OO system performance characteristics are explored in the first section, followed by identification and analysis of key OO performance influential factors in relation to the underlying OO programming languages. Later, the thesis focuses on finding general guidelines that can improve OO system performance. Within this chapter, there is no distinction between real concurrency and virtual concurrency unless they show different performance behaviors.

3.1 Performance Characteristics of OO Systems

Computer system performance is a critical and important issue of system construction process, especially for real time systems where the response time is a key concern. There is no exception for OO systems either. It must meet the system performance goals set up for the given application. The performance measures of interest are usually response time, throughput, utilization, queueing delays, etc.

Generally, people are aware of the real time performance hit from OO systems despite all the attractive advantages it can offer. Jacobson et. al.[36], Coad [26], Cox [27] etc. all mentioned the potential performance drawback of OO systems in their books. Identifying all the performance

reducing factors and finding the corresponding solutions become critical to the success of OO technology. A rigid methodology needs to be developed to keep track of performance factors during the process of OO system development.

Due to fundamental paradigm change, Object-Oriented systems differ significantly from traditional procedural systems. For example, OO programs consist of a collection of objects rather than an integrated algorithm. This affects the way in which the performance evaluation of OO systems should be done and the techniques we normally use. In order to find a better way to monitor OO system performance, OO system specific performance characteristics should be identified.

To some extent, the performance of OO systems depends on the characteristics of the OO programming language used for implementation. Generally speaking, OO languages are those that have object constructs and follow OO system technical principles, namely encapsulation, polymorphism, inheritance and classification etc. A closer examination of OO programming languages shows that although they all offer Object-Oriented system specific characteristics, the underlying implementation strategies in these languages are usually different, (see Table 1), which greatly affects system performance.

Table 1: Example OO Languages and Their Characteristics

Characteristics	Implementation	OO Languages			
		Smalltalk	C++	Eiffel	CLOS
paradigm	pure OO	yes	no	yes	no
classification & instantiation	classes are objects	yes	no	no	yes
data typing	class name is type	no	yes	yes	no
inheritance	single inheritance	yes	yes	yes	yes
	multiple inheritance	no	yes	yes	yes
	method is shared	yes	yes	yes	yes
	dynamic inheritance	yes	no	no	yes
polymorphism	dynamic binding	yes	no	no	yes

Table 1: Example OO Languages and Their Characteristics

Characteristics	Implementation	OO Languages			
		Smalltalk	C++	Eiffel	CLOS
compilation	interpreted not compiled	yes	no	no	yes
garbage collection	automatic	yes	no	yes	yes

A closer look at Table 1 reveals that no two OO programming languages offer the exact same features. Paradigm wise, only Smalltalk and Eiffel are pure OO languages. In Smalltalk and CLOS, classes are objects and the class name is not a type. The same statement is not true for C++ and Eiffel. Almost all languages support various forms of inheritance and the code is shared for inherited methods. Smalltalk doesn't support multiple inheritance but supports dynamic inheritance (binding of inherited methods during run time) and dynamic polymorphic binding (binding during run time), while C++ and Eiffel only support static inheritance and binding. This is due to the fact that C++ and Eiffel are compiled languages rather than interpreted languages.

Among these OO languages, Smalltalk and C++ are the two widely used ones. In Smalltalk, everything is an object, there is no single data per se, and all codes are organized as methods of objects; while in C++, besides objects, there exist *ints*, pointers, structs, arrays etc., and functions are available in addition to methods. Some other differences include the fact that error messages in Smalltalk are only "message not understood", as opposed to C++ where pointer errors are common; also, the garbage is collected automatically in Smalltalk while in C++ memory management is handled manually. Above all, Smalltalk is a pure OO language and C++ is only a hybrid OO language.

In the following sections, key OO performance factors are identified and examined and techniques for improving OO system performance is developed.

3.2 Key Factors and the Implementation

OO systems provide many advantages. For example, reuse through inheritance and polymorphism provides efficient implementation and reduces the development time significantly. On

the other hand, many things cause OO systems to pay heavily in performance. A close study on the performance factors reveals that while some of them, such as context switching, exist in both OO and non-OO systems, there are other factors that are specific to OO systems. All these OO related factors contribute significantly to the performance behavior of all OO systems, causing its real-time performance degradation. Identifying these factors will enable us to find the causes and hence search for corresponding solutions.

Some key factors are found and studied in the following sections. Different language implementations of those features are referenced here to show the effect of programming language on system performance.

3.2.1 Inheritance

Inheritance is one of the most attractive features offered by the OO paradigm. Yet it may also be an important factor that causes OO system performance degradation, since when a subclass inherits the attributes and methods from its parent class, a search has to be performed either at compile time or run time to locate the inherited methods and bind them. If this search is done at run time, then it produces a degradation in performance.

Binding techniques have a great effect on system performance and is determined by the actual binding time. A binding time is the time when an attribute of some portion of a program, or meaning of a particular construct, is determined [15]. Binding time can happen during the compilation (when the source of the program is encountered), during the linkage phase (when the results of several compilations are combined), or during the execution time (when a program is finally run). Depending on the binding, two types of bindings are possible: static or dynamic. *Static binding* means the binding of the method corresponding to the message is done at compile time or linkage time and is based on the static characteristics of the variable. On the other hand, in *dynamic binding*, the binding of the actual method to the message is based on the class of the instance and is done at run time. Performance wise, static binding is more efficient but not flexible; while dynamic binding enables flexibility but is not efficient.

The effect of inheritance on system performance comes from the fact that methods are shared among superclass and subclasses. Depending on whether the binding of inherited methods is done at run time or compile time, system performance behavior can differ significantly.

If the binding of inherited methods is done at compile time, it is usually less costly. For example, in C++, an object is represented by a contiguous region of memory, a pointer to an object of a class points to the first byte of that region of memory. The compiler turns a call of a member function into an *ordinary* function call with an *extra* argument; that *extra* argument is a pointer to the object for which the member function is called [66] [67] [68]. As illustrated in Figure 3.1, class A's definition is transformed into a memory location with the integer *a* stored. Also

Figure 3.1. Object Implementation in C++

no information relating to member function *f* is placed in the object. A call to *f* is transformed by compiler into an ordinary call with some pointer overhead.

In C++, the binding of inherited methods is done at compiling time, during which a pointer is created to point to the inherited methods [66]. As illustrated in Figure 3.2., class B is a subclass of A, class C is a subclass of B. In memory, objects from class B and C are composed by concatenating the inherited attributes and methods together. All attributes are stored in adjacent memory locations. References to all methods including those defined within the object and inherited ones will be kept in a table, the object simply keeps a pointer to this table. For example, an instance of class C has attributes *a*, *b* and *c* and a pointer to a list of methods *f*, *g*, *h*. A methods invocation like *pc->f(3)* becomes something like *(*(pc->fptr[0]))(pc, 3)* after compilation.

Figure 3.2. Inheritance Implementation in C++

The exact overheads in this scheme is low, including:

- (1). one subtraction of a constant for each use of a member in a base class that is included as the second or subsequent base;
- (2). one word per method in each method table to hold the reference;
- (3). one memory reference and one subtraction for each call of an inherited method;

On the other hand, for OO languages like Smalltalk, the binding of inherited methods is done at run time, which means that in order to find the corresponding method for the message, the

receiver object in execution has to climb all the way up its inheritance hierarchy and asks its superclasses for the ownership of a given responsibility. This definitely creates a heavier performance hit for the system (although upon using caching strategy, the overheads can be reduced).

So dynamic inheritance increases message overhead among the object and its parents because of the necessary searching needed to find the right methods, thus slows down the system response and reduces system throughput.

As Budd points out in [15], it is seldom possible for general-purpose software tools to be as fast as carefully hand-crafted systems. Thus inherited methods, which must be prepared to deal with arbitrary subclasses, are often slower than a specialized code. Besides the penalty in execution speed, the inheritance also creates added program complexity, memory requirements, and message-passing overheads [15]. When inheritance is largely used, the program size increases and the software gets more complicated.

So we see there is actually a trade-off between generality and performance, that is: a general solution might not be optimum for a particular case. Budd also suggests that for inheritance, approximately two to three additional assembly language instructions are used and the average performance penalty is around 10 percent (higher for dynamically bound language like Smalltalk and lower for statically bound language like C++).

System performance behavior also differs depending on whether the type of inheritance is single inheritance or multiple inheritance. Although multiple inheritance is a more natural representation of real world domain and an important structuring tool, it complicates the system significantly and create higher performance costs.

3.2.2 Polymorphic Binding

Polymorphic binding happens when either the receiver object is declared to be an instance of an abstract superclass and should be replaced by the concrete subclass to which this instance belongs, or when the method defined in the receiver object is actually a virtual method that should be replaced by the real methods defined in its subclasses. This replacement again could happen during either the compile time or the run time, obtaining static polymorphic binding or dynamic polymorphic binding, respectively. As stated in the last subsection, the performance effects

caused by these two types are different. In C++ static binding to the real method is implemented using virtual functions as illustrated by Figure 3.3.

Figure 3.3. Polymorphism Implementation in C++

As shown in the figure, implementing polymorphism in C++ involves setting up a reference table of available methods. Class A has two virtual member functions that need to be redefined in the subclasses. Subclass B offers concrete definition of method *g* and subclass C has concrete definition of method *h*. In memory, objects from class B and C are composed by concatenating the attributes and methods together. All attributes for one object are stored in adjacent memory loca-

tions. Similar to the inheritance case discussed in last section, all methods available to a object will be kept in a table, the object simply keeps a pointer to this table. For virtual functions, their abstract definition is overridden by the concrete definition found in subclasses. For example, an instance of class C has attributes *a*, *b* and *c* and a pointer to a list of methods *f*, *g*, *h* where function *f* is not overridden yet and still a virtual function defined in A, function *g* is redefined in class B, hence has new address *B::g*, so does function *h*. A method invocation like *pc->g(3)* becomes something like *(*(pc->fptr[1])) (pc, 3)* in compiler. So actually in C++, overheads caused by static polymorphic binding will be comparable to the overheads in the inheritance case shown in Section 3.2.1.

The exact overheads thus include:

- (1). one subtraction of a constant for access of a base class member of a virtual base class;
- (2). one word per method in each method table to hold the reference;
- (3). one memory reference and one subtraction for each call of a virtual function;

So we see these overheads are comparable to those occurred in inheritance case. This is verified in Chapter 5 where the measured results for the two kind of delays are actually the same in C++.

As for dynamic polymorphic binding, the sender object of a stimulus send a message to a receiver instance of a generic class and the receiver instance can belong to an arbitrary subclass of this generic class. But in this case, the search is performed during run time to find the actual object. After the search the method of the right receiver object is bound to the message. So we can see that although dynamic polymorphic binding simplifies the problems such as the development of general-purpose data structures, the performance cost is high in this case since a run time search has to be performed to discover the real code to execute every time a message is sent

3.2.3 Context Switching

As illustrated in Figure 3.4, internally, all the objects are mapped to different operating system processes, each process is controlled by Task Control Block (TCB) and the ready-to-run queue is organized by underlying operating system at TCB level. Depending on whether the communicating objects are mapped to same process or not, two situations are possible:

Figure 3.4. Objects Mapped to Operating System Processes

1). if the two communicating objects are mapped to the same process, (for example, object A and C in Figure 3.4), then there will be no process context switching at the operating system level, so the only performance cost will be due to the invocation of an object method by another object and stack context switching.

2). If the two communicating objects (such as object A and B in Figure 3.4) are mapped to different processes, then context switching at the operating system level might occur. For example, when object A sends a message to object D residing on a different processor, if D is not executing or not the last executed object, then context switching will occur which involves saving registers, saving process state etc. Similarly when object A sends a message to object B of another process, context switching occurs. In both cases, the delay will be more substantial.

In this thesis, only the second case which is the most interesting one is considered, so whenever the context switching between two objects is mentioned, by default we assume the objects are mapped to different operating system processes. Also the possibility of having light-weight context switches among different executing threads is not considered in the analysis.

If the last executed object on this processor is the same object that will execute, then two possibilities exist:

Possibility 1). when the object finishes executing of a message, its state information gets preserved in the processor's memory and registers until replaced by another executing object which runs on this identical processor. This means if the next executing object is the same object executed last time, then this object doesn't have to reload its context information. In this case, there will be no context switching delays.

Possibility 2). whenever an object finishes executing of a message, the object's state information that were saved in registers will be wiped out. So even when the same object is rerunning immediately, it has to reload all its state information back to the registers. In this case, context switching delays always occur.

Figure 3.5. illustrates the two types of message passing situations where the context switching happens under possibility 1) discussed above. Suppose object A, B, C reside on processor 1, object D, E, F reside on processor 2. When objects on processor 1 send message to objects on processor 2, context switching happens if the receiver from processor 2 is not the object that is executing. On the other hand, if the receiver object is currently executing or is the last executed object on processor 2, no context switching occurs. This type of message is called *cross-node message*, in the above diagram, *msg 1*, *msg 2* and *msg 3* are examples of cross-node messages. Another type of message is *in-node message*, like *msg 4* and *msg 5* shown in the diagram, they are sent between objects from same processor. In this case, context switching occurs every time this type of message is sent.

3.2.4 Messaging

Each object has a set of possible states. Since no direct manipulation is allowed, the object's inner states information can only be obtained and changed through messaging. When an object

Figure 3.5. Context Switch Due to Messaging

wants a service from another object or wants to change another object's inner state, a message request is sent to that object. Upon receiving messages, certain events are triggered which cause either a change of object state or the requested services fulfilled.

Message passing is a more costly operation than the simple procedure invocation by its nature. A detailed trace of the events and delays happening during a direct method invocation reveals that many overheads are embedded inside messaging.

For example, when an object receives a message from another object residing on a different CPU as shown in Figure 3.6, we see that four kinds of delays occur:

Context Switching Delay: When the receiver object shares the CPU with another object, if it is not executing when the message arrives or it is not the last executed object in the system, context switch happens.

Message Passing Delay: This is the communication delay occurring when the sender object

Figure 3.6. Messaging Between Objects Residing on Different CPU

sends the message to the receiver object. In the case where the message sender and the message receiver are running on different processors as Figure 3.6 shows, the delay includes a bus delay. Also when the sender object and the receiver object are on different memory partitions or belong to different processes, longer delay applies.

Message Queueing Delay: If the receiver object is busy executing when a message arrives, the message gets queued in object's private buffer and will be served later. The queueing delay records the time these messages spend in the waiting queue.

Other Overheads: Other overheads include the overhead to build list of message parameters before the message is sent, to setup and follow communication primitives. This overhead happens every time when a message is sent.

Figure 3.7 shows the delay distribution when a message is passed between objects residing on the same processor. In this case, the same types of delay apply. In Figure 3.7, one context switching will occur every time when a message is sent: another one will occur if the message receiver generates reply notification upon receiving and/or processing the message. The only difference is that message is not sent through bus, so the communication delay is smaller in this case (for example, only the cost of copying the message is included).

Figure 3.7. Messaging Between Objects Residing on *Same* CPU

The implementation of the message passing usually involves several additional assembly language instructions, the usual performance penalty is believed to be 10 percent ([15]), although it also depends on the specific language and whether the language is dynamically bound.

One note about delay due to messaging and context switching is that they are not actually OO specific delays and can usually be observed in traditional system as well, the difference is that they appear far more frequently in OO systems since the communication through messaging is much more frequent.

3.2.5 Other Factors

An object encapsulates its states and behavior inside. Because of the encapsulation, direct access of object's own states is not possible, rather it is done by sending requests to the owner object. This kind of access indirection slows down the system response.

Other possible factors include automatic garbage collection and interpretation etc. Automatic garbage collection relieves the designer from tedious memory management duty and reduces the chance to have memory leaking errors (means memory is not referenced by any pointers and is not accessible or usable), but it also implies increased performance cost to the system. The real time results will be affected proportionally to the automation process.

Interpretation is used in some OO languages such as Smalltalk as the compiling method. Interpretation is slower than compilation since code is compiled line by line, thus degrades the system performance in a degree.

3.3 Techniques for Improving OO System Performance

After identifying the key factors which have impact on system performance, the next step is to find a corresponding solution to eliminate or reduce their effects. This becomes even more urgent if the number of possible processors is limited and no extra processing power can be added. OO system performance can actually be improved in several ways, such as increasing the system processing power by adding more CPUs, using a different software architecture, finding ways to fine-tune the system, etc. Before the system is actually built, it is usually difficult to compare the real system performance behavior resulting from different design architectures. Certainly prototyping or simulation could help to give the designers some rough idea about which design is a better design. This is not the focus of this section. The objective here is to find general guidelines which can improve the system performance under any architectures. In the following sections, a set of guidelines are proposed which are inspired by SPE from Connie Smith [70]. While some guidelines are adaption of SPE principles to OO systems, the others are new.

3.3.1 OO Incremental Performance Monitoring Plan

Jacobson et al. state in their book that OO system performance evaluation should be “postponed until system is (partially) built” to avoid all guesses [36]. While this statement is true if we want to obtain accurate performance data, it is not a good idea to wait until the system is fully built and all data becomes available since the system may have severe performance problems such that it is not possible to meet the responsiveness requirements. Simple tuning might not be sufficient or possible especially for large OO systems where the number of objects is huge, object relationship is complex and tracing from implementation back to design and analysis model becomes difficult. For example for a system of moderate size and complexity, the development time is usually 7 to 18 months and needs about 3 to 6 major iterations each of 2 to 4 month long. The resulting system typically has 200 classes and 3000 methods [11]. The performance impact could be huge for such a big system.

Rather, the thesis proposes that OO system performance should be monitored starting from analysis and design stage, not at the testing stage, only in this way, can we detect the problem early and fix it promptly before wasting resources. The Incremental Performance Monitoring plan is proposed here for setting up the concrete steps for monitoring performance in different development stages.

The Incremental Performance Monitoring plan means to monitor OO system performance incrementally against performance requirements and to keep track of the performance issue within each iteration. The basic idea is to integrate the performance monitoring process into the OO system development process, so that within each development iteration, system performance requirements are verified, problems are detected and necessary corrections are performed. As illustrated by Figure 3.8, the OO development process consists of several increments. Every increment represents increasing levels of functionality and more details. Within each increment, the performance behavior of the system is analyzed according to guidelines presented in the next section and adjustments are made accordingly.

3.3.2 Guidelines for Better Performance

When system performance is very critical, which is especially true for real time systems, the guidelines proposed in this section should be used to improve the overall OO system performance. Note that these guidelines are stated in a qualitative way as problems to which to pay attention. Also some of these guidelines are actually inter-related, so improvement in one aspect could affect the result from the other. Therefore some cost and benefit analysis will be necessary to evaluation the overall performance impact on the system from applying a guideline. These guidelines will be demonstrated later for the case studies.

(1). Maintain Clean Object Interface

Object interface design has great impact on the system performance. Cleaner interfaces will reduce the system resources consumption. minimize the relationship between objects and lead to higher system performance, thus the domain should be partitioned and modelled in a way to ensure clean object interface.

Two important factors contribute to clean object interface: cohesion and coupling [12]. *Cohesion* is defined as the measure of the relatedness of elements and attributes within an object,

Figure 3.8. OO Incremental Performance Monitoring Plan

coupling is the measurement of inter-connectness among objects. Gane and Sarson point out the close connection between these two concepts in [29], that is, a highly cohesive module whose parts all contribute to a single function, is not likely to need much coupling to other modules. In OO systems, in order to have clean object interface, it is vital to increase cohesion within an object while at the same time decrease the coupling among objects, which means that during the analysis and design stage, all related operations or behaviors operating on the same set of data

should be organized within a single object. Also the interconnection between objects, such as data sharing should be reduced to increase object independence. Increasing object cohesion and reducing object coupling will both make object interface cleaner, thus eliminate unnecessary messaging.

(2). Group Objects into Collaboration Group

The Collaboration Group concept has been introduced in various places. It is defined as a group of interdependent objects interacting to achieve some objective [17] [32]. Some examples for collaboration groups are shown in [21]. For example, two peer processes in a distributed file transfer system cooperating through the exchange of messages to deliver files in a reliable manner. Collaboration groups represent a reusable architectural abstraction above the level of individual objects that are essential for understanding, designing and reusing OO systems.

The interaction patterns among object components within a collaboration group are defined as the sequence of activities performed by the participants and the temporal order of the message exchanged among them [21]. For example, *client-server pattern* is a behavioral pattern described as a set of server objects and client objects where a server receives request from clients, queues and then processes the requests and returns answer back to the sender [12]. It can be applied and reused on any client-server architecture. The *communication system pattern* includes source, destination and set of transmitters, channels and receivers. The information source sends messages to its destination through a set of transmitters, channels and receiver. Patterns are often repeated throughout a system or a different system with different participating object, thus by organizing objects according to collaboration group and patterns, we could establish a better defined architecture for a given system, also arrange available system resources in a more efficient way. For example, since objects within a collaboration group usually communicate with each other more frequently, we could consider to allocate them in the same processor if possible to achieve some real-time savings. Some other benefits include yielding a cleaner message interface and a better defined responsibilities for each object, reducing inter-object communication time among object and reducing unnecessary message flows.

(3). Minimize Messaging and Message-Passing Overhead

As referred in above two guidelines, inter-object communications both consume a lot of sys-

tem resources including CPU and increase the system response time, especially in a high traffic environment. In order to achieve a better system performance, messaging among objects has to be minimized in several aspects, such as avoid unnecessary message passing, minimize the size of messages, etc.

(4). Add Direct Object Association

When two objects have frequent communication through some other objects, consider to add a direct object association between these two objects, so that the communication path could be shorter. As illustrated by Figure 3.9, object A can only communicate with object F through the

Figure 3.9. Add Direct Object Association

associations or channels among object A and B, B and C, C and D, D and E, E and F. If we add a new association between object A and F, they can communicate to each other directly, rather than through a bunch of other objects, thus reduce the message passing time between them. Rumbaugh [60] also considers adding association as a design optimization technique.

(5). Reduce Code Inheritance Level

This matches localization principle for traditional systems. Reducing code inheritance level means to reduce code access path, thus decrease code accessing time. It can be achieved by dupli-

cating methods used by child levels. This approach has trade-offs since it may lead to higher memory requirement, causes data consistency problem and is also against design of reusability. A compromising solution will be to find and duplicate only the most frequently inherited attributes and methods so that the system performance could be improved at the minimal cost. For example, the relationship among classes in Figure 3.2 could be arranged in a hierarchy as shown in Figure 3.10. According to this guideline, if instances of class C use method *f* frequently and the underly-

Figure 3.10. Reduce Code Inheritance Level

ing OO language supports dynamic binding, then we could consider duplicate the method definition of *f* in class C too to achieve some run time savings. Also try to use single inheritance instead of multiple inheritance since multiple inheritance usually has higher performance penalty [60]. With many added benefits from inheritance to the OO systems, a balance between inheritance level and system performance becomes essential.

(6). Find and Fix Bottleneck Object

A bottleneck is a single contention point that limits the total system performance. An object becomes a software bottleneck when it exhibits a high utilization relative to the utilization of

other objects and its execution speed has become the limitation for the execution speed of the whole system.

After finding the bottleneck object through the performance statistics collected, several methods can be applied to resolve it, for example, allocate more or faster hardware resources to a set of objects to improve execution speed; implement it in a more efficient way, e.g. by using assembly language for part of the codes; change algorithm used; clone object; split the object and rearrange responsibilities of objects to reduce the workload for the original object.

(7). Rearrange Execution Order

Rearrange execution order means to activate the objects in a different order to increase parallel execution among objects. This can be illustrated by the following example. Here assume synchronous message passing is used. Suppose there are five objects A, B, C, D, E, each of them provides certain services and the processing time required are 1, 5, 2, 4, 3 accordingly. Object A sends a requests to B and C, object C sends request to D, when object E receives request from both object B and D, it starts processing the messages. If the execution order shown Figure 3.11 (a) is used, namely, A sends message to B first, then sends to C after receiving reply from B. B

Figure 3.11. Rearrange Execution Order

sends message to E, E then waits for message from D before starting processing. When C finishes, it sends message to D, D sends to E when it finishes, then E starts executing. As we can see from Figure 3.11 (c), the total time needed in this case is 15 and E spends 6 units of time waiting for second message. If we change the execution order to the one shown in Figure 3.11 (c), let A sends message to C first, then to B. When C finishes, D and B start processing in parallel, and finally E gets executed. This time the total time needed is 11, only 1 unit of time is spent on waiting by object E as shown in Figure 3.11 (d).

This guideline can usually be applied when there are more than one parallel ways to achieve the same functionality required. In this case an efficient way can be found by comparing the total processing times of different possibilities.

(8). Localization of Information

If an object uses some information stored in another object frequently through association to this object, consider moving the information into this object itself. In this way, accessing time can be significantly reduced as illustrated by Figure 3.12. Also if an attribute is calculated dynami-

Figure 3.12. Localization of Information

cally, rather than calculate it every time needed, a local copy is maintained and updated periodically to achieve run time saving. One drawback is that localization of information tends to make bigger object, so a balance should be kept.

(9). Avoid Unnecessary Polymorphic Binding

This guideline is the application of the *fixing principle* from SPE [70]. As stated in Section 3.2.2, direct binding of inherited methods is efficient but loses flexibility; polymorphic binding offers flexibility but is not efficient. There is no single right answer for which binding techniques should be used, rather, it depends on the priorities. For example, in system where responsiveness is primary concern (e.g. real time system), try to identify most frequent polymorphic binding spots and replace them with direct binding to improve system performance. In many cases, the flexibility on this matter probably is very limited by the programming language used, still when performance requirement is most critical, direct binding of the methods to be used will certainly be more preferred.

(10). Cloning Objects

In case one object in system becomes a bottleneck object, cloning is a good way to increase the system throughput. Cloning an object means to create a new identical object to process messages and destroy it when the required processing is finished. As illustrated by Figure 3.13, object A sends request message *msg 1* to object B. After it receives the message, B creates a cloned object B', and forwards this request *msg 1* as well as *sender's object id* to B'. B' then starts processing the message. when it finishes, it sends back the answer to the original sender according to the *sender's object id*. As soon as B' finishes processing and replies back, B' gets destroyed.

This approach also has its trade-offs. First of all, cloning objects creates new overheads on object clone's creation, deletion and state management. Secondly, cloning objects proposes new requirements on system resources which could be very limited, such as memory storage and number of available processors. Finally, cloning objects could potentially cause data consistency problem for those commonly shared data among the worker copies, e.g. the routing tables. Hence this guideline should only be used if the potential benefits can overweight the cost and the new requirements on both system resources and data consistency management, etc. can be satisfied. Some overall analysis and decision have to be made to figure out the combined impact. For exam-

Figure 3.13. Cloning an Object

ple. in analyzing the requirement for managing the cloned objects, we have to consider whether the required processing time is long enough to offset the newly created overheads; we should also consider the fact that this overheads could be reduced if there are enough new requests coming in so that the cloned object copies could keep processing them without being destroyed.

(11). Increase Concurrency Level

An increased level of concurrency enables a better performance. In order to maximize concurrency, synchronous message passing has to be reduced to avoid unnecessary waiting. One way to reduce synchronous message is to replace it using two asynchronous messages if the sender doesn't need the reply from receiver to proceed to next processing. If the sender does need the reply from receiver before going to next processing, check if it will need the result immediately, if not then it could perform other sub-computation while waiting for answer back.

Adding more processors is a general way to increase the concurrency level since the number of active objects in the system will increase if more processors are available for doing processing.

Finally use two phase synchronous message passing to replace one phase synchronous message passing can increase the concurrency level too. In two phase synchronous message passing, the results are returned as soon as they are obtained, and all the unimportant processing such as garbage collection and clean-ups, is left to the second phase of the receiver's processing, so that sender can resume its computation while the receiver keeps working on the unfinished processing. This way, by increasing the concurrency level within the system, the overall performance can be improved.

3.4 Summary

To some extent, the performance of OO systems depends on the characteristics of the OO programming language used for implementation. A closer examination of OO programming languages shows that although they all offer Object-Oriented system specific characteristics, the underlying implementation strategies in these languages are usually different.

Also in this chapter, key OO factors such as inheritance, polymorphic binding, message passing and context switching are identified and their impact on OO system performance is studied.

OO system performance should be monitored starting from analysis and design stage, not at the testing stage using the guidelines provided. This can be done by following Incremental Performance Monitoring Plan and a set of guidelines. The Incremental Performance Monitoring Plan means to monitor OO system performance incrementally against requirements and to keep track of the performance issues within each iteration. The guidelines proposed to improve OO system performance include: maintaining clean object interface, grouping objects into collaboration group, minimizing messaging and message-passing overhead, adding direct class association, reducing code inheritance level, finding and fixing bottleneck object, rearranging execution order, localization of information, avoiding unnecessary polymorphic binding, cloning object and increasing concurrency level.

Chapter 4

Case Study on a Real-Time Concurrent OO System

In this chapter, a case study on a real-time concurrent OO system called Automatic Conference Call System (ACS) is presented. OO analysis and design of ACS application are carried out and various important analysis models, such as the object model and scenario model are built and evaluated as part of the research. Classes, their responsibilities and collaborations are studied to identify important classes and associations. Those techniques presented in Chapter 3 are also applied here in the case study during analysis and design stages to make sure that the architecture proposed is both functionally correct and performance wise efficient. Chapter 5 then builds a simulation model of this ACS system focusing on representing its concurrent Object-Oriented properties and behavior and on capturing OO specific and non-OO specific performance statistics.

4.1 Automatic Conference Call System (ACS) Description

The case study is related to call processing in telephone switching system taken from [11]. The Automatic Conference Call(ACS) is a service available to the business subscribers of telephone company. In a nutshell, the ACS service will allow any group of users to establish and moderate a voice conference call which simulates the real face to face meeting. Each conference call has a single conference controller whose role is similar to that of a meeting chairperson or moderator. This person will be able to perform the following actions:

Initiate a conference call: The conference calling service must be invoked by the served

user, employing an initiating conference call request. When the service provider receives the request to allocate resources for the conference call, it checks to see that the requested conference can be established. This procedure includes conference resource availability check. The required resources are then allocated.

Add new party: The conference controller can request that a new party be added to a new or existing conference call. The conference call server will setup a call with the new party indicated by the party ID provided by the controller. The conference call server will make use of a subscriber directory to derive the actual phone number from the party ID. The subscriber directory also maintains the common name and picture of the party. Upon call establishment, the conference will be connected to this new active party.

Split party: The conference controller can request that a specified party be removed from the conference but remain connected to the conference controller. Execution of this request requires that the network establish a new call between the conference controller and the specified party, since that party is no longer associated with the conference call.

Isolate party: The conference controller can request that a specified party be prevented from communicating with the conference but not removed from it.

Reattach party: The conference controller can request that the specified party be reattached to the conference. Successful execution of this command permits a previously isolated party to again converse with all other parties that are connected to the conference.

Terminate the conference: The conference controller can request that the conference be terminated, which means every party associated with a particular conference be disconnected, and all conference resources be de-allocated.

The ACS is implemented by using object-oriented technology. All requests and replies are done by using messages. Some objects are implemented by reusing the superclasses already existing in the basic call software.

When conference calling is invoked, conference resources are allocated to the served user; and similarly when it is terminated, resources will be de-allocated, the resources include: the conference circuits, the physical line occupied and processor needed, etc.

During the initiation phase, conference controller invokes the conference call service. Once the conference is in the active state, the service is under the control of the conference controller. Any party other than the conference controller is called a conferee. All participants in the conference call are considered parties. Party ID is the controller's reference to a participated conferee.

The ACS application will have a graphic user interface(GUI) implemented and run on subscriber's microcomputers or workstations. Both conference controller and the conferees will be able to see the conference attendee's images.

4.2 OO Analysis and Design of ACS Application

4.2.1 Call Processing Activities

The Automatic Conference Call (ACS) application is a service running inside a telephone switch system on top of Basic Call layer. It uses all the network services related to call processing, billing etc. A telephone switching system is a large distributed system with a central processor and a set of peripheral processors. The central processor is the heart of a telephone switch where all the call processing related activities are carried out. The peripheral processors will off-load non-call-processing related computing from central processor and perform some administrative duties or pre-processing.

Figure 4.1 illustrates the call processing activity sequences in the telephone switches. (an arrow denotes the execution sequence). First when a call is originated, the line and channel resources are allocated and the dial tone is applied. Then the user-dialled digits are collected and loaded. When the switch detects that the digits means a feature invocation, it finds out the actual service that is invoked, if the service exists, then more service related resources are allocated, such as bridge or a conference circuit. In case the service is not valid or not subscribed or there are no such special resources available, the call is routed to exception treatment, such as tones and announcements. After special resources are allocated, a new dial tone is applied to request for destination directory number. When the digits of a directory number are collected and parsed, the translation process maps them into various translation paths, such as preliminary translation, translating area codes, translating secondary digits and screening. After every step is successfully finished, the call is routed to the appropriate trunk or lines, the connection is set up, related logs.

5X

billing and operational measurement records are recorded, and the two parties start talking. If in any step problem occurs, the call is routed to exception treatment, special tone and announcements are applied. When finally the conversation finishes, one of the parties hangs up, then logs, billings and operational measurement are notified to stop recording, resources allocated to this call get de-allocated, and the call is disconnected.

4.2.2 The ACS Object Model

From the ACS description presented in Section 4.1, a set of objects can be identified, each of them with distinct responsibilities in the application. These objects also have some similarities, for example, similar role, behavior or responsibilities. They can be further categorized into the inheritance hierarchy as shown on Figure 4.2. Note that besides using the analysis and design methods presented in Section 2.1.3, the performance guidelines identified in Section 3.3 are also applied during the process of deriving the objects and their responsibilities to ensure both design correctness and efficiency. For example, each object is designed with a clearly defined set of responsibilities; the interconnection among objects such as data sharing is minimized in order to ensure a clean object interface. All the collaborators of a class are identified and analyzed in order to capture the message exchange pattern among them; unnecessary messages and message indirection are also avoided to minimize messaging cost. Finally information used frequently by an object is localized to reduce messages.

Among the classes, *conference_call* and *regular_call* are two *call* classes in the telephone system. In this application, *conference_call* keeps track of state information pertaining to a conference call, such as *conf_controller* id, number of conferees, each *conferee*'s id, the current state of the conference call and the processor the conference call group members are running on. It also owns *conf_call_circuit* resources obtained through *conf_call_server* and *line* resources obtained through *call_processing_server*. The objects that *conference_call* collaborates with include *conf_controller*, *conferees*, *conf_call_server*, *line* and *conf_call_circuit*.

Regular_call has similar responsibilities as *conference_call*. It keeps track of state information related to a regular call, such as *reg_caller* id, *reg_callee* id, current state of regular call and the processor the regular call group members are running on. It owns *line* resource obtained through *call_processing_server* and collaborates with *reg_caller*, *reg_callee*, *line* and *call_pro-*

Figure 4.2. Inheritance Hierarchy of ACS

cessing_server.

Conf_controller, *conferee*, *reg_caller* and *reg_callee* are all subclasses of phone user class. In ACS application, *conf_controller* controls the conference call, sends various processing requests to *conf_call_server*, such as to initialize a conference call, to add new party, to split party, to isolate party, to reattach party and to terminate a conference call. It also keeps track of conference call stages. Its collaborators include *conf_call_server*, *conf_controller_terminal*, *conference_call*, *call_processing_server* and *conferees*.

Reg_caller object controls the regular call. It sends connecting and terminating processing requests to *call_processing_server* and keeps track of regular call stages. Its collaborators include

call_processing_server, *regular_terminal*, *regular_call* and *reg_callee*.

Conferee and *reg_callee* objects behave very similarly. They receive connecting or terminating requests from *call_processing_server*, and react accordingly to its current state. *Conferee* collaborates with *call_processing_server*, *conference_call*, *conf_controller* and *conferee_terminal*, while *reg_callee* collaborates with *call_processing_server*, *regular_call*, *reg_caller* and *regular_terminal*.

Conf_call_circuit and *line* are subclasses of *resource* class, *conf_call_circuit* keeps track of state information for each conference circuit in system while *line* keeps track of state information for each line in system. *Conf_call_circuit* collaborates with *conf_call_server* and *conference_call*. *Line* collaborates with *call_processing_server*, *conference_call* and *regular_call*. *Line* class is further divided into three subclasses depending on types of lines used, they are *fiber*, *copper* and *other*.

The *server* object is complex object with small object parts, built inside and hidden internal behaviors. The responsibilities the *server* object owns are internally shared and executed by component objects. Since the internal objects will never be stand-alone, but rather collaborate to provide server functionalities, they are considered to be an integrated server object and are not modelled individually.

In ACS application, the *server* class has four subclasses: *conf_call_server*, *call_processing_server*, *database_server* and *disk_server*. *Conf_call_server* receives and processes requests from *conf_controller*, sends call processing related requests to *call_processing_server* and sends conference queries to *database_server*. It also acquires and releases *conf_call_circuit* resource on behalf of conference call. Its collaborators include *conference_call*, *conf_controller*, *call_processing_server*, *database_server*, *conf_call_circuit*, etc.

Call_processing_server receives and processes requests from *conf_call_server* and *reg_caller*, sends translation and routing queries to *database_server*, and sends connecting and terminating requests to *conferees* or *reg_callees*. It also acquires and releases line resources on behalf of *conference_call* and *regular_call*. Its collaborators include *conf_call_server*, *reg_caller*, *database_server*, *line*, *conferee* and *reg_callee*.

Database_server receives and processes requests from *conf_call_server* and *call_process-*

ing_server. It also sends *conf_query* (for querying conferee information) and *xlar_query* (for query related to translation and routing) requests to *disk_server*. Its collaborators include *conf_call_server*, *call_processing_server*, *disk_server*.

Disk_server is used to simulate delays associated with physical disk I/O. It receives and processes *read*, *write*, *xlar_query* and *conf_query* request from *database_server*. It collaborates with *database_server* object.

Conf_controller_terminal, *conferee_terminal* and *regular_terminal* are three subclasses of *terminal* class. *Conf_controller_terminal* collaborates with *conf_controller*, it is the physical device to send out requests issued by *conf_controller*. Similarly, *conferee_terminal* is the physical device to send out requests issued by *conferee*, *regular_terminal* is the physical device to send out requests issued by *reg_caller* and *reg_callee*.

The object model in Figure 4.3 depicts all the relationships among these objects, such as uses, subclassing, aggregating, etc. and the cardinalities for each relationship (a detailed description of the object model notations can be found in Appendix A). The object model also reveals that a server could be a *single server* or a *multi-server*. A *multi-server* is made up of one *dispatcher* and a set of *worker* copies. All the worker copies share one buffer where all the newly received messages are queued up. The dispatcher will then forward the messages to the non-busy worker copies.

All these objects are *objectwise-concurrent*. Since each conference call group resides on a single processor, *conf_controller* and *conferees* are actually virtually concurrent. Same argument is true for *reg_caller* and *reg_callee*. For *server* objects, such as *conf_call_server*, *call_processing_server*, *database_server* and *disk_server*, since they are all located on different processor, they are *executional-concurrent*. The concurrency is a real concurrency. Other concurrencies such as the concurrency between *conference_call* and *regular_call*, the concurrency between *line* and *circuit* etc. are all execution concurrencies.

4.2.3 Mainline Use Cases and Their Execution Scenarios

Mainline use cases represent the key scenarios occurring in a system and are important for performance analysis. In ACS application the mainline use cases include: initiate a conference call, add conferees, talking and terminate conference call.

Figure 4.3. Object Model of ACS Application

They are the scenarios that the study focuses on, since they represent the basic behavior a conference call has and will be the determining factors for whole system's performance. Other less important use cases include: split party, isolate party, reattach party.

For the mainline use cases, all the dynamic interactions among conference call members and the sequence of messages and events occurred during a conference call are captured in the message sequence chart (MSC) shown in Figure 4.4. The execution scenarios are as follows:

First, a `conf_controller` picks up the phone and dials the feature activation code to *activate the conference call*, which generates a `init` message to `conf_call_server`. The `conf_call_server` then tries to acquire a conference circuit. If the circuit resource is successfully obtained, the `conf_call_server` notifies the `conf_controller` to proceed by sending `reply yes`, otherwise, the `conf_controller` has to wait and try again.

Upon receiving the `yes` notification, the `conf_controller` sends `add` request to `conf_call_server` to add all the conferees. In order to retrieve all information about the conferees, such as image, phone number, name, etc., `conf_call_server` sends a `conf_query` message to a database server. Once `database_server` finishes the query, it sends the query result back.

If the query is successful, `conf_call_server` sends `connect` message to `call_processing_server` passing the conferee's directory number and requests to set up a connection between `conf_controller` and conferee. Call processing server first checks if there is a free line. If yes, the switch will translate and route the call to the proper destination (conferee) through the line. The destination conferee's phone rings.

Once the conferee's phone goes off hook, the connection is established, logs, operational measurement and billing system are notified, the call goes into *talking* state (shown as a heavily shaded bar in Figure 4.4). The conference call server continues adding all the conferees in this way. If during the setup process, failure occurs, such as unable to acquire resources, the call is routed to treatment and proper tone and announcements are applied.

When the conference call is about to finish, the `conf_controller` sends a *terminating* message to the `conf_call_server`. `Conf_call_server` forwards the message to the `call_processing_server` which disconnects the call and deallocates the line resources assigned to the call. When all conferees are disconnected, `call_processing_server` then notifies `conf_call_server` by sending message

Figure 4.4. Message Sequence Chart for *Conference Cull*

finish. Upon receiving *finish* notification, *conf_call_server* frees the conference circuit used and the conference call is finished.

Similarly, the message sequence chart for regular call is captured and represented in Figure 4.5.

Figure 4.5. Message Sequence Chart for *Regular Call*

From the mainline scenarios presented earlier, we can obtain the state transition diagram for ACS application as shown in Figure 4.6. Detailed descriptions about state transition diagram notations are included in Appendix A.

Figure 4.6. Conference Call State Transition Diagram

4.3 Summary

This chapter presents a case study on a real-time concurrent OO system called Automatic Conference Call (ACS). OO analysis and design of ACS application are carried out, a set of domain objects and their attributes are identified. Their responsibilities and collaborators are examined, also the concurrencies among the objects are considered.

Several important analysis models are constructed for ACS application. The relationship and cardinalities among objects are represented through ACS object model, and detailed messaging

behaviors among objects in both conference call and regular call are captured through message sequence charts. Finally, the state transition diagram of a conference call is constructed to show various system state transitions in the duration of a conference call.

Chapter 5

ACS Simulation Model

This chapter presents the ACS simulation model constructed for the Automatic Conference Call System introduced in Chapter 4. In first section, a brief introduction is made to reveal the basics behind computer simulation and rationale for constructing the ACS simulator. Then PARASOL ([50][51]) modelling capabilities for OO system are explored and steps for constructing an OO simulation program using PARASOL are presented. Later a PARASOL based simulation model for Object-Oriented ACS application is built and performance related statistics are obtained. The emphasis has been put on correctly representing the ACS system behavior, accurately modelling and capturing OO related performance overheads. Finally analysis is carried out to show object response time and utilization behavior, software bottleneck in system is identified and studied, various OO related delay effects are examined.

5.1 About the Simulation

5.1.1 Introduction

Computer modelling and simulation has been considered by research and engineering communities as a cost-effective methodology for solving problems, especially in the field of computer system performance evaluation. Simulation is defined by Hoover and Perry in [33] as the process of designing a mathematical or logical model of a real system and then conducting computer-based experiments with the model to describe, explain, and predict the behavior of the real system. It has become very attractive because of its flexibility and representativeness. Simulation

focuses study on sources of variation and is able to rerun, augment, and replicate experimental runs. It is used widely in computer performance analysis to examine the accuracy of an analytic approximate algorithm.

Key requirements for a simulation model include representativeness and appropriate abstraction so that unnecessary details are avoided and assumption are reasonable. Only when there two requirements are both met, can we ensure that the simulation model being built is both cost effective, and yet representative to the real system, so that decisions based on model behavior reflect those based on actual experiments.

5.1.2 Motivation for ACS Simulator

The ACS simulation model is built by using PARASOL, a distributed system simulator kernel ([50][51]). The ACS simulation focuses on representing the system's concurrent OO properties and behaviors, hardware configuration is represented only at a very abstract level. Also key factors that have strong impact on OO system performance will be modelled and their impact on OO performance will be studied. The main objectives of this chapter is to provide answers to the following questions:

- How to model Concurrent OO System characteristics and behaviors through simulation?
- How to represent OO characteristics in a simulation program and how to reflect various OO specific and non-OO specific performance overheads?
- How to capture all possible OO related delay factors into a simulation program and to obtain various performance related statistics? How will these overheads affect the overall system performance?

5.2 Using PARASOL to Model OO System

PARASOL is a "medium-to-fine-grained" simulator for distributed and parallel computer systems ([50] [51]). It can be used either as a performance measuring tool, or as system prototyping tool for testing parallel or distributed algorithms and emulating system functionalities.

PARASOL offers a flexible, user configurable, simulated network on which user defined tasks execute. Its kernel is entirely written in ANSI C and is therefore highly portable and easily extensible. In addition to the typical elements usually found in a discrete simulation library, e.g.

random number generators, statistics gathering mechanisms, event manipulation routines, etc., PARASOL contains a number of component constructors to facilitate the modelling of the system hardware, software and its network topology. Also, PARASOL provides a concurrent task paradigm and a set of system calls to facilitate task management, inter-task communication and task synchronization, etc. PARASOL is also an object-based simulation tool: its task, port and message building blocks closely resemble those concepts defined OO system. The basic functionalities PARASOL provides can be categorized in the following areas:

Hardware Configuration: PARASOL provides functions to simulate a set of hardware resources, such as CPUs, buses and links between different hardware components. Each component has its private queue with definable queueing sequence as illustrated by Table 1. In PARASOL, hardware components are set up in the *genesis* task by calling three types of functions: `ps_build_node`, `ps_build_bus` and `ps_build_link`.

Table 2: PARASOL Hardware Components

hardware component	queueing policy	queue item
node	HOL, FCFS, PR, RR	task
link	FCFS	message
bus	FCFS, Random	

Task Management: tasks are used to model everything except processing nodes, buses and links. PARASOL offers functions for a user to create, resume, kill and suspend a task. Tasks can be in different executing or non-executing states and can have different task priority assigned to them. They communicate with each other through messaging.

Inter-task Communication: PARASOL uses a port-based inter-task communication mechanism, and ports have a virtually unlimited capacity for queueing incoming messages on a *First Come First Serve* basis. Seven system calls are available for sending and receiving messages.

Task Synchronization: An array of spin-locks and counting semaphores are provided for implementing mutual exclusion synchronization or as building blocks to construct more elaborate synchronization mechanisms.

Statistic Capturing: PARASOL provides two ways to facilitate statistics capturing. One is

called *automatic capturing* to obtain hardware device utilization statistics. Another way is *user defined statistics capturing* to capture mean device utilization statistics, e.g. queue length, cpu utilizations, etc. or to collect mean population statistics, such as mean response time, mean waiting time, etc.

A typical PARASOL model has three major types of components: *a driver* which manages tasks, queues, system calls and collects statistics; *an execution environment* which is the hardware platform for the program, including nodes, links and buses; and *a set of user-defined tasks*.

The ACS simulator is built on top of PARASOL kernel and utilizes the system calls, library functions and other constructs provided by PARASOL. It is constructed according to following steps:

- from object model identify objects and functionalities to be implemented. Also identify those objects which have dynamic run time behavior. These objects will be simulated as PARASOL tasks in the simulator. For simplicity, they will be called *task objects*. The rest of objects will be simulated as static data structures;
- transform the responsibilities of objects into services and behaviors of task objects and decide on the number of instances for each task class;
- decide the hardware platform for each task object;
- decide the type and number of performance statistics that need to be captured;
- implement identified environment, tasks and statistics. First define a special control task called "genesis" within which users define the hardware topology, the initial configuration of the system; then write set of user defined tasks;
- execute the program and obtain performance results;

A small clarification should be made here about *objects* and *processes*. From Section 3.2.3, recall that we choose to have all objects internally mapped to different operating system processes. One process usually has many objects within it. In this ACS simulation program, some implementation choices are made. Since those objects which have dynamic run time behavior can be simulated very nicely using PARASOL *tasks*, they are implemented as individual operating system *processes* or *tasks*. However it should be noted that potentially it is possible to map several objects to a single *process*, in which case the *process* becomes a complex object made up of several *component objects*, and the messaging among the *process object* and its *component objects* will become the internal method invocation. As we will see from Section 5.3.4, performance

overheads due to internal method invocation are modelled in the ACS simulation, related statistics such as number of internal method invocation are also captured. Therefore the analysis done here can be applied to the case where more than one objects are mapped to a single operating system process.

In the following sections, objects and functionalities implemented in ACS simulator are identified. Components for this simulator are described and explained, detailed object interaction behaviors are illustrated.

5.3 ACS Simulation Model

The ACS simulation model mimics the operating behaviors of an Automatic Conference Call service for a telephone switching system. The description of ACS feature can be found in Chapter 4. The ACS simulator has about 5500 lines of C code.

5.3.1 ACS Model Component Description and Behavior

The implementation of ACS simulation model uses the OOA and OOD models obtained in Chapter 4. It covers the mainline use cases presented on Section Section 4.2.3. Not all the objects and inheritance hierarchies are implemented in the simulator. In Figure 5.1, all the objects that are simulated in the object model are shaded. Among them, *conf_controller*, *conferee*, *reg_caller*, *reg_callee*, *conf_call_server*, *call_processing_server*, *database_server* and *disk_server* objects are implemented as PARASOL tasks since they possess key run time behaviors. *Bus* object is provided by PARASOL kernel, other shaded objects are implemented as data structures or functions. For example *callee* is implemented as a data structure which keeps track of phone receiver's user id, current state, line occupied if any, think time, and non-busy probability. In *Fiber*, *Copper* and *Other* classes, only one method *setup_line()* is implemented, their state information are kept track of by their superclass *Line*. Objects are modelled by using PARASOL tasks in the simulator because of their close resemblance to objects.

Also, not all call processing functionalities shown on Figure 4.1 are simulated. Only those functionalities which are key for representing the conference call system behaviors are simulated. The simulated functionalities are shown as shaded on Figure 5.2. Among them *feature invocation* and *allocating special resources* are specific to activating a conference call, all other shaded parts

Figure 5.1. Object Model of ACS Application

Figure 5.2. Implemented Call Processing Components

are applicable to both conference call and regular call.

The hardware platform of the ACS simulation model is made up of a set of processor nodes and a central bus. Basically seven processing nodes are simulated, which have either uniprocessor or multiprocessor built on them. Processor nodes are divided into three categories: *server processor node*, *conference_call processor node* and *regular_call processor node*. There are four of server processing nodes in the simulator, *conf_call_server* copies execute on a multi-processor node. *Call_processing_server* copies run on a multi-processor node too. A *dispatcher* object is used to distribute all the incoming processing requests to different *conf_call_server* copies. During actual implementation, *dispatcher* object is implemented using PARASOL provided *shared-port* construct. Same mechanism is used to manage *call_processing_server* copies. *Database_server* runs on a uniprocessor node, so is *disk_server*.

The *generator* object is an artificial object used specifically in the simulator to generate the conference call activating requests. Its role is similar to the physical user in the sense of load generating. Since *conf_controller* object has too many responsibilities, an implementation choice is made to make up a separate *generator* object to be responsible for load-generation for conference call groups. This *generator* object will be sharing the processor node with *genesis* task. For the regular call groups, the load-generating function is embedded into the *regular_caller* object.

Every conference call group and regular call group will be executing on one uniprocessor node (see Section 5.3.2 for the assumptions made), which means *conf_controller* and conferees in a *conference_call* group share one cpu, same situation for *reg_caller* and *reg_callee*. Finally all these processing nodes are connected by using a central ethernet-like bus and all communications among objects in different processor nodes are carried out through the bus.

Figure 5.3 ~ Figure 5.5 show object state information for all the implemented classes. Since PARASOL doesn't support encapsulation for object, the encapsulation and other OO characteristics are implemented in the program through controlling object access point for data or methods (single point access). The attribute data and the methods are implemented simply by using data structures and procedures.

In ACS simulation model, one object is allowed to offer multiple types of services, thus able to process multiple types of .message requests. For example, *cp_server* can process request *CON-*

Figure 5.3. Attributes for *User* and *Call* classes

Figure 5.4. Attributes for *Server* class

NECT or *TERM* sent by *cc_server* and request *reg_connect* or *reg_term* sent by *reg_caller*. Similarly *db_server* can process *conf_query* requests to obtain conferee information, and *xlar_query* requests for translation and routing.

Object cloning is also supported for server class, *conf_call_server*, *call_processing_server* and *database_server* can have any number of cloned copies in the simulator.

Semaphores are used to ensure mutual exclusion during obtaining and releasing line and circuit resources.

Figure 5.5. Attributes for *Resource* Class

All the inheritance relationships among shaded objects shown on Figure 5.1 are modelled in ACS simulation model. When inherited methods are invoked, method look-up delay will occur. Also all task objects will be subclasses of *Object* superclass and thus inherit some common attributes and methods from it.

Polymorphism is also represented in the ACS simulation model. It allows a class to define a set of simulated virtual functions which will be replaced by the concrete functions defined in the subclasses. For example Figure 5.5 shows that *Line* class has an abstract method called *setup_line* which is overridden by the one of the three *setup_line* methods defined in subclass *Fiber*, *Copper* and *Other* since different line type requests different line setup procedures. Another example is the *Call* class shown in Figure 5.3. *Call* class has a set of abstract methods defined, such as *get_node*, *set_node*, *get_status* and *set_status* etc. These abstract methods will be overridden by concrete methods defined in *Call* subclass *Conference_call* and *Regular_call*.

Messaging are simulated by using PARASOL message mechanisms used for tasks. For

other non-task objects, messaging behavior is simply recorded and some delays are applied without explicit send and receive messages.

5.3.2 Assumptions

An actual phone user can assume different roles in different calls, for example, to be a regular residential caller or callee on one time and to be a conferee the next time. For the switching program, although the physical users are different for every call, their logical representation in program's memory and roles are the same. Therefore for *User* class in ACS model, its subclasses *Conf_controller*, *conferee*, *reg_caller* and *reg_callee* represent the logical entities in the switching program, rather than real phone user.

It is also assumed that *conf_controller* and the set of conferees within the same conference call group will be sharing the same processor, which is a little different from real-life applications. In real switching networks, one processor at the end user level controls a group of lines/terminals, therefore there do exist context switching among those lines/terminals. Here in ACS simulator, although the *conf_controller* and conferees within a conference call group do not necessarily have to be located on the same processor, in order to model the context switching among the group of lines/terminals existing in real life applications, we assume here that all the conference call group members share a same processor. Same assumption is also made to regular call groups so that both *regular_caller* and *regular_callee* within the same regular call group reside on the same processor.

In this study, we also assume that when an object finishes executing a message, its state information will get preserved in the processor's memory and register until replaced by another incoming executing object which runs on this identical processor. So for objects executing on same processors, if the next executing object is the same object executed last time, it doesn't have to reload its context information. It is also assumed that when an object finishes executing, its state information gets preserved in the processor's memory and registers until later replaced by another executing object which runs on this identical processor. Finally, all the *task objects* (objects that have dynamic run time behaviors) are assumed to be big complex objects and are mapped to an operating system process (a PARASOL task).

2 of/de 2

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

PRECISIONSM RESOLUTION TARGETS

5.3.3 Delay and Overhead Representation

This section shows how the OO related performance overheads are obtained and represented in ACS simulator. Note that the accuracy of these delay figures is not of the highest importance here: the intended objective is to find the delay overhead figures with reasonable accuracy so that we can plug them into ACS simulator framework to test their effects on the overall system performance. Therefore the delay figures found here do not necessarily represent the real delay metrics for the given system platform.

Inheritance

In ACS simulation model, every class is associated with a distinct value to represent its relative location or level in the inheritance hierarchy. If we use *inheritance delay per step* to denote the delay caused by climbing up *one* higher level of inheritance hierarchy, and *number of look-up steps for a method* to denote the difference between the starting hierarchy level and the ending level, then

$$\begin{aligned} & \text{the total delay time caused by searching for a method into superclasses} \\ &= \text{inheritance delay per step} * \text{number of look-up steps for this method} \end{aligned}$$

The value used for *inheritance delay per step* in this simulation was estimated from a single C++ program. During the measurement, it is assumed that the total inheritance delay during a method invocation grows proportionally with the number of look-up steps in the inheritance hierarchy. A lump sum delay is obtained for a fixed number of climbing-up steps and is then averaged to obtain the inheritance delay per step. For example, for a 33MHz 386DX PC with 8Mb of memory, the approximate delay is 2 μ S.

Messaging

All the requirements and overheads related to messaging can be well represented by using PARASOL provided communication related functions. Message : passing among objects residing on same processor and different processors are differentiated in PARASOL by using different library functions. *Ps_send* is used for intra-processor messaging and *ps_bus_send* is used for inter-processor messaging. Also delay due to *message passing* between objects has been set to be proportional to the size of the message sent.

Message queues with FCFS scheduling policy is modelled in ACS model by using PARA-

SOL built-in message queue owned by each task. If an object is executing when a message arrives, the message gets queued in the receiver object's message queues in a FCFS order. Two kinds of ports are used in ACS simulator: *regular port* that is owned by a single object and *shared port* that is owned by *object clones* (identical object instances). For example, all *conf_call_server* copies share one single port called *cps_shared_port*. These two types of port are both supported by PARASOL functions.

Since both message communication delay and message queueing delay are embedded in all related PARASOL functions, they are not simulated separately in the simulator. Delay due to context switching during messaging is managed separately under context switching delay category. The other delays happened during messaging, such as setting up communication primitives and building message parameter lists, are represented in ACS program by applying a small delay associated with each messaging.

Also since the performance overheads from *intra_process* messaging (for example, messaging from object A to C in Figure 3.4) and *inter_process* messaging (e.g. messaging from object A to B in Figure 3.4) are different, as previously stated in Section 3.2.3 and Section 3.2.4, different delay figures and statistics data (also defined in Section 5.3.4) are used in ACS program so that the performance impact from these two types of messaging can be observed and tested separately.

Context Switching

There are many situations that cause context switching, such as the use of different operating system scheduling algorithms, which happens in non-OO system as well. Here we only consider the case where the context switching is caused by messaging among *task objects* that are residing on same the processor or on different processors.

A variable is used in ACS model for each processor to keep track of last executed object. Whenever the next executing object is the same object executed last time, this object doesn't have to reload its context information, thus the delay caused by context switching is zero. On the other hand, if the next executed object is a different object, it has to load all its state information into the processor registers, in this case, the number of context switches gets incremented and the delay due to context switching will be applied. Since *Conf_controller* and *Conferees* will be served by the same processor, context switching delay is applied either when they

receive messages or when they send messages to each other. The same is true for *Reg_caller* and *Reg_callee* objects.

The context switching time used in ACS simulator was obtained through a small UNIX based C program by letting two processes signalling to each other. By executing this small program on a SUN 33 MHz Sparc Station on which the ACS simulator will be running, we can obtain the approximate delay figure which is 266 μ S per context switch on this type of workstation.

Polymorphic Binding

In ACS simulator, polymorphic binding might happen on *Line* class and *Call* class when the abstract methods defined in the two classes are overridden by concrete classes from their subclasses. When this happens, a *polymorphic binding delay* is applied. The value used for this delay in this simulation is obtained through a C++ program. During the experiment, similar assumptions like those assumed in inheritance case on page 81 are made to obtain an averaged delay figure for the given platform. Therefore the delay figure does not necessarily represent the real metrics. For a 33MHz 386DX PC with 8Mb of memory, the approximate delay is 2 μ S.

5.3.4 Performance Statistics Capturing and Description

Three categories of performance statistics are collected, one category includes a set of *conventional* performance statistics related to each object, the second category includes utilization data for each cpu in the processor nodes, the third one includes statistics for evaluating OO related performance overheads.

The *conventional* performance statistics include *cycle_time* which is collected for *User* objects and *Server* objects, and *utilization*. They are captured by using PARASOL provided statistics capturing functions. From them throughput and other performance data can be calculated.

Besides the conventional performance statistics, the following OO performance specific metrics are collected: number of direct method invocations, number of context switches, number of polymorphic binding occurrences, number of look-up steps for inherited methods and number of internal method invocations within an object. Following is a detailed description of these OO related statistics and when they are to be collected.

Number of external method invocation records the number of external messages received which causes a direct method invocation. The messages can be sent by objects residing on a different CPU or the same CPU, and the cost for processing these two kind of messages are different as shown in Section 3.2.4. In ACS model, only the sum of external messages is recorded, the delay caused by context switching during method invocation is addressed by another statistics, and delay due to different messaging media (via bus or direct) is reflected by using PARASOL provided *ps_send* and *ps_bus_send* calls. *Number of external method invocation* statistics is implemented by adding an attribute called *num_exter_mthd* to each object (in *object* superclass), and this attribute will get incremented by the object whenever a message is received.

Number of context switches is a counter associated with changes in any computer execution domain during the processing of a transaction caused by messaging. It represents the flow frequency of data between units of computation. The counter will be incremented whenever an object which receives a message is sharing a processor with some other objects and is not the last executed object in that processor. In ACS model, since *conf_controller* and *conferees* from a conference call group will be sharing one processor, context switching will occur upon receiving a message, same thing is true for *reg_caller* and *reg_callee* in regular call group. An attribute is used for each *Call* class object to keep track of the last executed object identifier. So whenever an object in the *Call* class receives a message, it checks whether it is last executed object: if so then no context switch occurs for processing this message, otherwise, the number of context switches gets incremented and context switching delay is imposed.

Number of polymorphic bindings records the number of polymorphic binding happenings in the program. In the ACS simulation, polymorphic binding happens for *Call* and *Line* classes. *Call* class provides abstract methods called *get_status*, *set_status*, *get_node*, *set_node*. During run time whenever these abstract methods are bound to the concrete methods defined in *Conference_call* or *Regular_call* subclasses, the sum gets incremented. Same situation applies to *Line* class too. This statistic is implemented as attribute kept by *Line*, *Conference_call* and *Regular_call* classes.

Number of lookup steps for inherited methods records the total number of steps the object has climbed up during the execution time in order to find the inherited methods from its superclass. This statistics is collected on a per object base and implemented as an attribute for *Object* superclass and inherited by all its subclass objects. The assumption made here is that the code for all inherited methods is available on the CPU while the object executes.

Number of internal method invocations is a way to record the number of nested method invocations for an object upon receiving an external message request. Again it is collected for each object individually. It is implemented as an attribute for *Object* superclass and inherited by all its subclasses. This statistics combined with number of direct method invocation collected gives an average number of nested method invocations per external message request.

All delays and overheads are controlled by using boolean switches. By turning on a switch corresponding to one particular performance factor (such as inheritance), the performance overheads related to that factor will be applied to the program and eventually affect the final performance statistics collected from the simulation. Also performance statistics specifically related to that factor such as number of look-up steps for one inherited method will be collected. Through this way, effects of these OO overheads caused by each individual performance factor can be observed and compared.

5.4 Detailed Simulation Model Behavior

In this section, a detailed architecture of the simulation model is presented, including the hardware platform, execution environment, task objects and functionalities. Also OO related delay and overhead representation are described, statistics collected from simulation are analyzed. This section focuses on explaining detailed ACS model execution behavior such as object creation, run time object interactions and the execution sequences.

5.4.1 Object Creation

Figure 5.6 illustrates the object creation process. As shown in the diagram, all task objects run on processor nodes which are denoted by round rectangles. At first all processor nodes are created by a task called *ps_genesis* and the bus communication among the nodes is set up. Once the execution environment is established, global statistics are created and all task objects are launched. All the standard ports owned by objects are created. Shared port and acknowledgment ports get created one the task objects start executing. The arrows shown on the diagram means *create*, among them light-colored arrows show the creation of processor node, and dark ones show the creation of task objects.

Note that in the simulator, the *dispatcher* object is implemented using the shared-port con-

Figure 5.6. Object Creation Process

struct provided by PARASOL system. Also, the number of regular call groups and number of conference call groups are decided randomly during simulation. Each regular call group will be located on a uni-processor node. Same thing is true for each conference call groups. Finally, number of conferees in a conference call group is chosen randomly and there will be only one `conf_call_controller` per conference call group.

Note also that the naming for some classes in ACS simulator has been changed for simplicity reasons, `conf_call_server`, `call_processing_server`, `database_server` and `conf_controller` are referenced in the above diagram simply as `cc_server`, `cp_server`, `db_server` and `cc_controller`. The same changes are made in the diagrams shown in the following two sections.

5.4.2 Object Interaction Diagram

Figure 5.7 ~ Figure 5.9 show the objects' behaviors when they interact with each other through messaging during the period of a conference call. Four basic scenarios including initiating a conference call, adding conferees, talking and terminating conference call are covered. The number in the bracket following the message denotes the ordering of message passing. For example, message flow for initiating a conference call and adding conferees are shown in Figure 5.7. First, generator sends `start` message to `cc_server` dispatcher requesting to start a session. One of the `cc_server` copies receives the request, acquires conference circuit resource and sends a `yes` reply back to `cc_controller`. Having been permitted to start a session, `cc_controller` sends an `add` message to `cc_server` dispatcher to add a conferee to a conference call. Again one of the `cc_server` copies receives the request, sends a `conf_query` message to `db_server` dispatcher to obtain information related to each conferee. If the information is *not found*, the conferee is marked *non_exist*. On the other hand, if the information is *found*, then `cc_server` sends a `connect` conferee message to `cp_server` dispatcher to pass the conferee information. One of the `cp_server` copies receives the message, acquires proper line resources, does translation and routing, and sends a `connect` command to that conferee. If the conferee is not involved in another call when the command arrives, it gets *connected*, otherwise a *busy* message is sent back. In either cases, `cp_server` notifies `cc_server` the results obtained, `cc_server` in turn replies to `cc_controller` that the conferee is *added* or *busy*. `Cc_controller` then updates conferee's state according to reply received, proceeds to next step and tries to add the *busy* conferee later.

When all the conferees are added, the conference call is in a *talking* stage as illustrated by

Figure 5.7. Message Passing During INIT & ADD in Conference Call

Figure 5.8. Message Passing During TALK in Conference Call

Figure 5.9. Message Passing During TERM in Conference Call

Figure 5.8, during which *cc_controller* and conferees talk to each other. Other conference groups might be in different call stages. The same scenario applies to regular call groups.

Finally when conference call is about to finish as shown in Figure 5.9, *cc_controller* sends a *term* message to *cc_server* dispatcher, one of the *cc_server* copies receives message, then sends *term* message to *cp_server* dispatcher for every connected conferee. One of the *cp_server* copies receives the message and requests the corresponding conferee to terminate. If the conferee is successfully terminated by indication of *finish* message, *cp_server* deallocates line resource assigned to that specific conferee. Once all the conferees are *disconnected*, *cc_server* deallocates circuit resource and reports *successful* termination to *cc_controller* which in turn notifies generator that is finished. Then the conference call enters the *inactive* stage.

Similarly, message interactions among objects during a regular call are illustrated in Figure 5.10, Figure 5.11 and Figure 5.12 with three basic scenarios covered. It is simpler than for the conference call service. During the initiating stage, *reg_caller* sends *reg_connect* message with *reg_callee* information to *cp_server* dispatcher. One of the *cp_server* copies receives the message, acquires line resource on behalf of *reg_caller*, does proper translation and routing, and sends the *connect* command to *reg_callee*. If *reg_callee* is not involved in another call, it gets connected, the call enters talking stage as illustrated by Figure 5.11. Otherwise a *busy* message is replied back, *cp_server* de-allocates line and reports to *reg_caller* that *reg_caller* is *busy*. When the call is about to be disconnected in Figure 5.12, *reg_caller* sends a *reg_term* message to *cp_server* dispatcher. Similar to what happened in the conference call case, one of the *cp_server* copies receives the message, disconnects *reg_callee* by sending a *term* command, de-allocates line resource assigned, and reports *finish*. The regular call then enters the *inactive* stage.

5.4.3 Execution Sequences

The execution sequences explained in last sections can be well represented through Timethread diagrams (detailed description about timethread can be found in [17] and [18]). Figure 5.13 and Figure 5.14 show the conference call related objects and their activating and serving sequence with time for *initiating conference call*, *adding conferee* and *terminating conference call* scenarios during a conference call. Similarly Figure 5.15 and Figure 5.16 show the regular call related objects and their activating sequences with time for *starting up* and *terminating* scenarios during a regular call.

Figure 5.10. Message Passing During *INIT* in Regular Call

Figure 5.11. Message Passing During TALK in Regular Call

Figure 5.12. Message Passing During TERM of Regular Call

Figure 5.13. Timethread for INIT & ADD in Conference Call

Figure 5.14. Timethread for TERM Scenario in Conference Call

Figure 5.15. Timethread for *INIT* & *ADD* Scenarios in *Regular Call*

Figure 5.16. Timethread for TERM Scenario in Regular Call

5.5 Performance Analysis

The ACS simulation model is used for a number of test cases to obtain a set of performance results under different load and server copy configuration. In the following subsections, the system behavior changes with respect to response time, software bottleneck and OO related delays are observed and analyzed. All the simulation experiments were run sufficiently long to yield performance results with 99% confidence interval of less than $\pm 1\%$.

5.5.1 Response Time Behaviors

The response time of the client is usually related to the load of the system. In the ACS simulation, the factors that determine the load of the system include number of conference calls, number of regular calls, think time of generator, conference controller, conferees, *reg_calls* and *reg_callees*. For brevity, load change caused by the think time of the client is not studied, rather we use a set of think time values producing a system behavior such that the utilization of the servers will be low when the number of conference calls and regular calls are small, and vice versa.

First the *cc_controller*'s response time in function of the number of conference calls is examined. The number of regular calls is fixed at value of 5 in this case. *cc_server*, *cp_server* and *db_server* are configured to have 2 copies, 4 copies and 1 copy respectively. Since we study three different functions of a conference call (initiate a conference call; add conferees and terminate a conference calls), the response time for these functions are examined separately and are named as *init_time*, *add_time* and *term_time* to mean conference call initiation time, conferee adding time and conference call termination time. *Init_time* change in relation to the number of conference calls is shown in Figure 5.17 (a). All the *init_time* shown are the average initiation time of all *cc_controllers* in system in a particular experiment. The same arguments apply to *add_time* shown in Figure 5.17 (b) and *term_time* shown in Figure 5.17 (c). All the response times increase significantly with the increasing number of conference calls. Here the curve is not very smooth, because in each experiment, the number of conferees in each conference call group is randomly chosen (to match up with the real application), so the total number of conferees in system is not necessarily proportionally increased with the number of conference calls.

Similar behavior can be observed for a regular call. This time number of conference calls is

Figure 5.17. Response Time Behavior for Conference Call Client

fixed to be 5, *cc_server*, *cp_server* and *db_server* configurations remain to be 2 copies, 4 copies and 1 copy. Different response time are measured for the two functions in the regular call and are named as *init_time* and *term_time*. *Init_time* change in relation to number of regular calls is shown in Figure 5.18 (a) and *term_time* change is shown in Figure 5.18 (b). Again both response time are the average time of the set of statistics collected during a regular call. We can see from the graphs that both *init_time* and *term_time* increase with the increasing number of regular calls.

Figure 5.18. Response Time Behavior for Regular Call Client

5.5.2 Software Bottleneck

The software bottleneck is a performance problem that appears in client-server systems with intermediate servers that require services from other servers. In Concurrent OO Systems, an object assumes the role of both a client and server, processes message requests from other client objects as well as sends nested requests to its server objects. In either case, this object is shown "active" or "busy". When this object finally becomes saturated (i.e. shows high utilization), it becomes the bottleneck object and slows down all its client object in the system.

As Figure 5.19 illustrates, in case of software bottleneck, the hardware servers still have processing capacity (utilization equals 0.7), but the software is highly utilized (utilization equals $1.9/2 = 0.95$) and limits the throughput of the whole system.

The bottleneck behavior is verified using ACS simulation model. Again the number of regular calls is fixed at value of 5 in this case. The cc_server, cp_server and db_server are configured to have 2 copies, 4 copies and 1 copy respectively. As we saw from the response time diagrams shown last section, with the increasing number of conference calls, conference_call client response time increased significantly. From that same experiment, we can obtain utilization figures for all server objects too. Figure 5.20 shows that the utilization for cc_server increased in

Figure 5.19. Software Bottleneck

Figure 5.20. Software Bottleneck Behavior in Conference Call Scenario

function of the number of conference calls. Eventually `cc_server` becomes highly utilized and starts slowing down `cc_controller`. For example, when number of conference calls is 7, cpu utilization for `cc_server` is only 50 percent, but `cc_server` utilization is 94.53 percent, which means

cc_server object has become the software bottleneck for the system.

Similarly, in the regular call case, where number of conference calls is fixed to be 5, cc_server, cp_server and db_server configurations remain to be 2 copies, 4 copies and 1 copy, we can see that when number of regular calls is increased, cp_server becomes saturated causing the reg_call client response time increased dramatically as shown previously in Figure 5.18 (a) and (b). Also cc_server becomes highly utilized too as shown in Figure 5.21 (a), in turn causing

Figure 5.21. Software Bottleneck Behavior in Regular Call Scenario

increased response time for cc_controller (Figure 5.21 (b)). For example, when the number of regular calls is 40, cpu for cp_server has utilization about 76 percent, while cp_server has utilization at 99 percent, which confirms that cp_server has become the bottleneck of the system.

The solutions for alleviating software bottleneck effect are discussed in [52]. Among the solutions are multi-threading and cloning. The cloning approach is the cheapest in many situations because it is a purely software solution and is supported by many operating environments, provided that there is no critical section in the cloned task. In ACS simulator, cloning is supported through the PARASOL kernel: all the clones will be running on same set of processors as the original task. Figure 5.22 illustrates the resulting system behavior after cloning is applied to cc_server

Figure 5.22. Utilization & Response Time Change with `Cc_server` Cloning Level

object. In the experiment, the system is configured to have 5 conference call groups, 3 regular call groups, `cp_server` has 4 copies and `db_server` has one copy.

As shown in the diagram, cloning level of `cc_server` is increased in steps, from Figure 5.22 (a) we see decreased utilization for `cc_server` and increased utilization for `cpu` of `cc_server`. In Figure 5.22 (b) we see `add_time` for `cc_controller` has been improved with the increasing number of `cc_server` copies. Bottleneck behavior has been reduced. The graph also shows that with the reduced `cc_server` utilization, `cp_server` utilization has increased. When number of `cc_server` copies equals 6, most of the potential gains are achieved, the limit to further increase is the capacity of the processor that `cc_server` copies are running on, which is 99 percent utilized, and the capacity of `cp_server`, which is at nearly 90 percent utilization. We see the bottleneck has been pushed down to its server object in this case, which matches the study results shown in [52]. Similar results can be obtained from cloning `cp_server` as shown in Figure 5.23. under the following configuration: number of conference call groups equals 5, number of regular call groups equals 45, `cc_server` has 2 copies and `db_server` has 1 copy. Only in this case, the bottleneck is not pushed down, but pushed up to `cc_server`.

Figure 5.23. Utilization Change with *Cp_server* Cloning Level

So we see that by finding the bottleneck object and fixing it using cloning methods according to the guidelines presented in Section 3.3.2, the system's performance can be improved and the bottleneck can be alleviated. On the other hand, since the bottleneck tends to shift to another task, either the one below it in server hierarchy or the one above it, it may be necessary to clone several tasks to achieve the maximum performance, a strategy which is ultimately limited by the hardware resources.

5.5.3 OO Related Delay Behaviors

As mentioned in section Section 5.3.4, each type of performance statistics and overheads are controlled through a boolean switch. Then during experiment time, by turning on a switch corresponding to one particular performance factor (such as inheritance), the performance overheads related to that factor will be applied to the program and eventually affect the final performance statistics collected from the simulation. Also performance statistics specifically related to that factor such as number of look-up steps for one inherited method will be collected.

Through this way, performance results with the delay included can be obtained and later compared against the previous results obtained. Figure 5.24 shows the conference call `add_time` changes when each one type of overhead is considered. We can see from the graph that in all three

Figure 5.24. Response Time Change under OO Performance Overheads

cases, add_time values are higher than the one obtained without overheads.

Also the overheads are observed to be 10 to 25 percent from above graph. In a large real life application, these overheads will be higher since the class hierarchy is deeper and more software layers are present.

Also from the simulation experiments, overhead statistics specifically related to OO system are obtained. The average values for different number of conference calls are shown on Table 3.

Table 3: Performance Overhead Statistics Collected from ACS Simulation

Average number of look-ups per object per call	Average number of polymorphic binding per call	Average number of context switches per call	Average number of messages per object per call	Average number of internal method invocations per object per call
176.07	9.73	12.44	195.56	101.02

From the table, we can see that generally speaking, during each conference call, the average total number of climb-up steps in inheritance hierarchy for an object is about 176, the average

number of polymorphic binding happenings per conference call duration is about 9.7, the average number of context switching per conference call duration is 12.44, and the average number of messages per object per conference call duration is 196. Also we can see that about one internal method invocation occurs inside an object for every two messages received.

5.6 Summary

A simulation model is constructed to capture the behavior of the Automatic Conference Call system and its concurrent Object-Oriented properties. OO specific and non-OO specific performance overheads are represented on the top of the base architecture. Further performance analysis on the statistics obtained reveals the object response time behavior, bottleneck behavior and OO related delay behavior. The effectiveness of object cloning on alleviating bottlenecking behavior is also demonstrated.

Since we did not have a dedicated object-oriented system simulation tool available, PARASOL was chosen as the base system upon which the Automatic Conference Call System was built. PARASOL is an object-based simulation/prototyping tool offering a whole array of system modelling functionalities and is currently the best we could find for this simulation. On the other hand, since it is not written specifically for modelling object-oriented system purpose, many concurrent OO system characteristics such as encapsulation, inheritance, polymorphism, etc. have to be implemented separately through the user written program, which leads to a higher programming effort.

Chapter 6

Abstract Performance Modelling of Concurrent OO System

Among the possible ways for system performance evaluation, performance modelling is becoming increasingly important, not only because of its flexibility and low cost, but also its wide applicability. It can be used in various stages of system lifetime, i.e. before or after the system is constructed. An abstract and elegant performance model enables us to get the key performance results quicker and can be very informative during the system development process. This chapter aims to evaluate current performance modelling techniques with respect to their ability to model concurrent OO systems by using the case study presented in Chapter 4. The focus is especially on finding answers to the following questions:

- What are the specific requirements for modelling Concurrent OO system performance?
- Is SRVN sufficient for performance modelling of COOS system?
- If new model features are needed for a given concurrent OO system, what will these be?
- Can we find a systematic way to obtain workload parameters for this extended model? Can various performance overheads (include OO specific overheads) be represented in the performance model?

Note that the thesis focuses on the modelling power of analytical performance models applied to OO systems, not on their solution methods. Therefore the new extended model is neither solved analytically in this chapter nor by simulation.

6.1 Performance Modelling Requirements

6.1.1 Object Behaviors

As mentioned before, OO system differs significantly from traditional system in many aspects. One important aspect is that OO system is essentially a collection of cooperating objects rather than an algorithm or collection of processes, hence modelling the behavior of object properly into OO performance model becomes a crucial part of OO system performance modelling. This implies that in order to capture the essence of OO system, *object should become the constructing entity* or unit in the analytic performance model. Some other behaviors that should be also supported in the intended performance model are identified next.

Object interaction behavior: objects interact with each other only through messaging. All processing requests are sent to the appropriate object by embedding them into a message. Upon receiving the message, the receiver object processes message or sends further request to another object. Different messages can have different processing requirement, frequency and complexity. For any object at any time, at most one message can be processed at a time. From the example object interaction diagram shown on Figure 6.1, we see object A interacts with object B and D by

Figure 6.1. Example Object Interaction Diagram

sending message $m1$ and $m3$, similarly object B and D communicate with object C by sending message $m2$ and $m4$.

OO system specific characteristics: These characteristics are the essential part of OO system, and their influences over OO system performance should be represented in the analytic model. Example characteristics include encapsulation, inheritance, polymorphic binding, etc. (introduced and studied in Chapter 2 and 3).

Class of service: usually every object owns more than one responsibilities and can offer various classes of services corresponding to different message requests.

An object can be either a server or a client or both: every object has its own state and responsibilities, which means that it is able to process certain number of requests and provide certain classes of services. In this sense every object is a potential server. On the other hand, no object is an island. All object stand in relation to others, on whom they rely for services and control [5]. When an object needs a service which cannot be fulfilled by its own capabilities, it sends the service request to other objects which offer this service. In this sense, every object is a potential client. The role as a client or a server is usually referenced within a certain boundary or context (often referred to as a cluster). For example, the same object can be a server in one context (only receives and processes message requests, never sends request to any other objects); and a client in another context, only send's message request and never does any processing. Therefore an object in OO systems typically has dual identities and embeds both server and client characteristics together. This can be shown on Figure 6.1, where object A is a client, object C is a server, object B and D are both a server and client in the diagram. The fact that object A is a client in this particular diagram doesn't mean that it cannot be a server though, since that part of interaction might exist but simply beyond the context of this diagram.

Message buffering: message buffering should be supported in an analytic performance model so that when a message arrives, if an object is busy executing, the message does not get lost, rather it will be stored in the private buffer owned by the receiver object and taken out later for processing when the receiver object becomes free. The scheduling algorithm for the buffer could be according to different scheduling policies, for example, FIFO or priority.

Flexible message pattern: an object can send any number of messages and can send differ-

ent types of message requests to another object for different services.

Nested requests: when the message request sent by sender object cannot be fulfilled by the server object, or when the services the sender object requests are the responsibilities shared by the receiver object and some other objects, the receiver object may choose to send a nested request to another object and asks for its service.

Object cloning: object cloning is used to alleviate system bottleneck behavior and to increase system concurrency level. The behavior of the cloned object is discussed in Chapter 3 and illustrated by Figure 3.13. The proposed performance model should be able to model this cloning behavior.

6.1.2 Concurrency

An OO system is essentially a concurrent system by its nature. Since OO system is made up of collection of objects, when the system executes, multi-objects will co-exist in the system's memory and several execution threads can co-exist in the system. As illustrated earlier in Figure 2.4, all the objects in that graph are *objectwise-concurrent* since they coexist at the same time, object B is *state concurrent* since control threads are split inside B and later synchronized, and object B and D are *executionally concurrent*. The performance model for OO system should be able to model all these types of concurrencies.

6.1.3 Resource Contention Among Objects

Among all the objects, some of them are purely resource objects, such as cpu, disk, etc. which provide processing resources, and some of them are server objects which provide services. When an object needs certain resources, it sends a message request to this resource object, so do other objects which have same demands. This creates hardware resource contention for resource objects. Similarly, contention happens when many objects requests service from a server object, in this case, it can be referred as software resource contention. For example in Figure 6.2, both *object 1* and *object 2* need services from *object 3*, which creates a possible software contention situation. Also all objects in the diagram will have requests for cpu object, thus creates a hardware contention. Clearly this resource contention behavior among objects should be captured by performance model.

Figure 6.2. Object Contention

6.1.4 Message Request Types

By analyzing the typical message behaviors present in ACS application, we see that the message request types in Concurrent OO system can be categorized as follows:

Synchronous Rendezvous: this request type could happen if sender object and receiver object are executing on different processors. As shown from the timing diagram in Figure 6.3, the sender object makes a rendezvous request to the receiver object, then it is blocked. The sender resumes execution only after receiving a reply from the receiver. All rendezvous requests are queued in a single FIFO queue at the receiver. The receiver accepts the first request from its queue and executes an *in-RV* phase. During this phase, the receiver may make nested requests to other servers. After ending the first phase and sending a reply to the sender, the receiver may enter its *post-RV* phase which is parallel with the execution of the sender. In this second phase, the receiver executes on its CPU and can make requests to other objects. At the end of the second phase the current cycle is completed. The receiver starts a new service if there are queue requests or becomes receive-blocked waiting for another request to arrive. In this case, the behaviors and rules played

Figure 6.3. Request Type I: Synchronous Rendezvous

by these two objects can be characterized as *non-refusable* and *blocking*.

Synchronous Rendezvous phase 1: sender object needs the results obtained from the receiver object's processing after it sends messages to callee, so it block-waits until receiving reply from receiver. For receiver, after it processes the request and sends back the reply, it processes next request if there is any available in queue or idle. This is a special case of Rendezvous request with only in RV phase.

Most message request present in ACS show this type of behavior. Figure 6.4 shows the time diagram for the message sent between *cc_controller* and *cc_server*. After *cc_controller* sends *init* or *add* or *term* message, it blocks and waits for reply to come back. When *cc_server* receives *cc_controller*'s request, it does some processing, then sends nested message to some other servers such as *cp_server* for further services. When finally it finishes all nested requests and processing, it returns the result back to *cc_controller*. As soon as *cc_controller* receives a reply, it ends self-blocking and resumes next execution.

Figure 6.4. Request Type II: Synchronous Rendezvous Phase I

Non-blocking Message Sending: when the sender object sends a message, but does not wait for the reply to come back, rather it proceeds immediately with its next procedures, and later when the receiver object finishes processing, it will send a notification message back to the original sender. In this case, the receiver object always executes concurrently with sender object. This message type usually occurs when the message passing does not result in any new computing results or the sender object's immediate computing steps doesn't need this computing results right away. The behavior of this message type can be characterized as non-blocking.

Figure 6.5 shows the timing behavior of this type of message by using objects from ACS example. The messages between *generator* and *cc_controller* in ACS application are of *Asynchronous No-waiting* type. *Generator* just sends a *start* message to each *cc_controller* after every conference call inter-arrival time. It does not care the result of the conference call, rather it keeps its own processing such as tracking and changing conference call state, recording time passed for a conference call, and starting up another conference call after the inter-arrival time by sending

Figure 6.5. Request Type III: Non-blocking Message Sending Request

start message to the *cc_controller* in another *conference_call* group. When a reply message from *cc_controller* comes back some time later, *generator* then handles the message and updates conference call states.

Message Forwarding: when an object receives a message which is not within its own processing responsibilities, it forwards the message to another object which owns the responsibility without actually doing any processing itself. This can be illustrated by Figure 6.6, where *object A* sends request *msg1* to *object B*. Since *B* doesn't have processing knowledge for *msg1*, it forwards *msg1* to *object C* which offers such service. For *object A*, from the time it sends the request *msg1* until it receives the reply back, during this period of time, it could be blocked-wait for reply or executing some other job. Thus message forwarding is a special type of nested message request.

Figure 6.6. Request Type IV: Message Forwarding

Tentative Command Message: Tentative command request type is a new request type identified from ACS application. In ACS, the communication pattern between *cp_server* and *conferee* is very different from others and is illustrated in Figure 6.7. When *cp_server* sends a connect command to *conferee*, if *conferee* is currently involved in another call, it notifies busy and refuses to set up the connection as shown in Figure 6.7 (a) (through time-out). On the other hand, in Figure 6.7 (b) if *conferee* is idle at that time then it replies yes, the connection is set up. *cp_server* then can go on serving some other requests. The connection is terminated as soon as *cp_server* sends a term command (Figure 6.7 (b)) or when *conferee* hangs up which triggers a disconnect message sent to *cp_server* (Figure 6.7 (c)). In this case, message sender *cp_server* is not a pure client and message receiver *conferee* is not a pure server. The behavior and roles played by two objects can be characterized as refusable and non-blocking. We can see from the timing diagram that *Tentative Command Message* is neither a forwarding message type nor a rendezvous type request. The same type of message requests exist between *cp_server* and *reg_callee* in ACS application.

Figure 6.7. Request Type V: Tentative Command Message

6.2 SRVN Modelling Capabilities

The Stochastic Rendezvous Networks (SRVN) model introduced in Chapter 2 is a performance modelling tool for concurrent software using a rendezvous (RV) for communications between parallel processes or tasks. This section examines the possibility of using SRVN model for Concurrent OO system. The main capabilities being examined include whether SRVN model can support various modelling requirements and the message request types among objects presented in last section.

SRVN model is considered because as we observed from last section, Concurrent OO System fits nicely into the concurrent Client-Server system category and SRVN has been a proven accurate method to model concurrent client-server application

The basic modelling block in SRVN model is a SRVN task, represented by a parallelogram in a diagram. If we use a task to model object, we see that many object behaviors shown in last section can be supported by SRVN tasks, including nested service, object's dual identities as a server and a client, multiple classes of services, message buffering, multi-phase behavior of servers in case the request type is Synchronous Rendezvous. SRVN task also supports flexible message pattern so that number of requests and type of requests are not constrained in any way. Also various OO specific characteristics an object has, such as encapsulation, inheritance, polymorphic binding, etc., could be potentially supported in SRVN model by first abstracting them into the corresponding overheads and then combining them into the service time for that object. Also object cloning is supported by the multi-server feature of SRVN model.

The second aspect to be examined is the concurrency behavior and resource contention behavior presented in OO system as summarized by last section. In an SRVN model, all reference tasks are concurrent, a server and its clients are also concurrent, hence object concurrency is well supported. Since in SRVN model, client task and server task can actually reside on different processors, execution concurrency is supported too. For state concurrency in Concurrent OO System, since it usually appears in multi-threaded objects, in which case, one control thread can be split into multiple threads and multiple control threads are synchronized into one thread. Multi-threaded object can be modelled by multi-threaded task in a way.

The resource contention behavior among objects can be modelled in SRVN model well.

SRVN can model resource contention for both software resources and hardware resources. If several tasks are collocated on the same CPU, the impact of the processor contention can be modelled by transferring to the processor the execution time of the software tasks collocated on it. Moreover, execution time differences of each task phase and of each task can also be modelled by simply using the multi-entry server concept. For example, the object contention behavior captured in Figure 6.2 in last section can be easily mapped into the SRVN model shown in Figure 6.8. where

Figure 6.8. A Mapped SRVN Model

Task1, Task2 are collocated on same server Task3 and Task1, Task2, Task3, Task4 are collocated on same CPU. CPU object is modelled as a server in SRVN model that has only “in RV” phase and is a pure server due to the fact that it never makes requests to other servers.

Not all message request types identified in Section 6.1.4 can be supported by SRVN model. SRVN model only covers situations where synchronization and communication are unified, meaning the caller sends a message, then it waits (in RV stage) until it receives reply from the callee and usually the results are given along with reply; then it executes in parallel with callee

which is in POST RV stage usually. SRVN model supports *Synchronous Rendezvous* and *Synchronous Rendezvous Phase I* (as a special case of Synchronous Rendezvous type), but it does not support *Non-blocking Message Sending*, *Message Forwarding* and *Tentative Command Message* request types. Although these message types can potentially be modelled by SRVN through approximation, for example we could approximate the Tentative Command Message happened in ACS simulator by embedding this behavior into `cp_server` object. Of course, this kind of approximation faces the danger of violating object encapsulation, that is: only the object itself should know its internal state information. Therefore although SRVN supports resource contention well, the mapped SRVN model shown in Figure 6.8 for the diagram in Figure 6.2 might not be a valid one if not all of the messages involved in the diagram belong to Synchronous Rendezvous type or Synchronous Rendezvous Phase I type.

As a conclusion, although SRVN model supports many Concurrent OO System performance modelling requirements, it fails to support some message request types presented in the last sections. Also, the support for multi-threading inside an object has to be extended. The most difficult part, developing new analytical solution methods for these extensions is beyond the scope of the present thesis.

6.3 OO Performance Model

As we concluded from last section, SRVN model cannot represent all Concurrent OO System performance modelling requirements, an extended performance model needs to be constructed to allow increased modelling capability for additional request types, and to calculate their performance results accordingly. We simply call this extended model *OO Performance Model*. In the following section, a complete model description is given (some features discussed in the previous sections are included here for completions).

6.3.1 Model Description

The *OO Performance Model* consists of a set of active or passive objects. They communicated with each other through message request which may include arguments and control information. The sender object acts as a client, sending a processing request while the receiver object behaves like a server, processing request and replies back if there is any.

In the OO performance model, the following features are supported:

- an object has dual identities and can act like a server or a client, or both;
- a message is buffered and message pattern can be flexible;
- message can belong to different message types;
- server object can send nested request to other objects;
- server object can offer multiple classes of services;
- server object can have multi-phase behavior of servers in case the request type is Synchronous Rendezvous.

An object assumes dual identities. It can be a client, sending message request to other object for processing, or can be a server, receiving and processing the request made by some other objects and later returning the results back to the originating object. More frequently, objects perform the role of both clients and servers, they offer a range of services and also send request to other objects for other services.

Nested message request sent by receiver object is supported, messages sent by an object can have different message patterns as well as different frequency and complexity. Also there is no limit on total number of messages that can be sent.

Not all the objects are active at the same time, but they all have the possibility to execute concurrently. When a message arrives at a server object which is busy doing processing, message gets queued. Queueing discipline could be FCFS (First Come First Serve) or according to some predefined priorities.

Messages send among objects can belong to different request type. Currently identified request types include *Synchronous Rendezvous*, *Synchronous Rendezvous Phase I*, *Non-blocking Message Sending*, *Message Forwarding* and *Tentative Command Message*. All the timing behaviors of these request types have been shown in Section 6.1.4. These request types can be identified according to following descriptions and represented in OO performance model by using the notations shown below.

Synchronous Rendezvous Phase I: For this type, sender object block-waits for reply from receiver object and computation of sender is suspended until reply is returned back. If receiver object is not busy at that time, it processes the request right away, otherwise the request gets

queued in the receiver's private waiting queue and will be accepted and processed later. For receiver object, once it finishes processing the request, then the job is done. After it returns the answer, the sender object gets unblocked immediately, the receiver itself will go on processing next message either from queue or received from another object. This type of message request happens when the sender needs the processing result back before proceeding to next computation. *Synchronous Rendezvous Phase I* request is represented in graph as shown in Figure 6.9.

Figure 6.9. Notation for *Synchronous Rendezvous Phase I* Message Type

Synchronous Rendezvous: This type has two phases, the first phase is the same as last type. After the receiver object returns the result, sender object gets unblocked and starts executing while the receiver object continues second phase processing. This type is used to allow receiver object to perform some administrative processing such as re-initialization without delaying the computation of sender object. *Synchronous Rendezvous* request is represented in graph as shown in Figure 6.10.

Non-blocking Message Sending: For this type, sender object proceeds with its own computation immediately after sending the message to the receiver object. The receiver object receives the message (if non-busy) and processes it concurrently with sender object's processing. *Non-blocking Message Sending* increases concurrency by triggering new activity and avoiding waiting. *Non-blocking Message Sending* request is represented in graph as shown in Figure 6.11.

Message Forwarding: When an object receives a message which is not within its own processing responsibility, it forwards the message to another object which owns the responsibility

Figure 6.10. Notation for *Synchronous Rendezvous* Message Type

Figure 6.11. Notation for *Non-blocking Message Sending* Request Type

without actually doing any processing itself. *Message Forwarding* request is represented in graph as shown in Figure 6.12.

Tentative Command Message: This is a new type of request identified. For this type, the behavior and roles played by two communicating objects are characterized as refusable and non-blocking, it is neither a forwarding message type nor a rendezvous type request. *Tentative Command Message* request is represented in graph as shown in Figure 6.13.

Also various OO system specific characteristics can be supported in the model through abstraction. Basically, first consider the performance overheads and delays that are caused by each of these characteristics, then combine these delays into the service time for the object where the

Figure 6.12. Notation for *Message Forwarding* Request Type Figure 6.13. Notation for *Tentative Command Message* Request Type

delay occurred. The various ways for obtaining these delays have been presented in Section 5.3.3 and Section 5.3.4, and the procedures for combining them into the workload parameters will be discussed in Section 6.3.3.

OO Performance Model can be constructed through the following main procedures and these procedures are used in next section to derive the OO Performance Model for ACS application. These main procedures are:

- construct object interaction diagram;
- construct message sequence chart;

- categorize all messages into different message request types defined in *OO Performance Model*;
- identify and decide hardware platform, allocate objects into different processor;
- map object model into *OO Performance Model*.

6.3.2 OO Performance Model for ACS Application

By applying the rules and using the building blocks presented in last section, we can obtain the OO Performance Model for ACS application as shown in Figure 6.14.

In the diagram, all the objects interact with each other through messaging. Messages and their directions are represented by using *directed arcs*. Different message request types are represented by using different *arc* notations introduced in Section 6.3.1. There are in total three types of requests appeared in the application, they are *Non-blocking Message Sending* for *start* and *finish* messages between *generator* and *cc_controller*, *read* and *write* messages between *db_server* and *disk_server*, *Tentative Command Message* for all messages between *cp_server* and *conferee* or *reg_callee*. Finally *Synchronous Rendezvous Phase I* is the request type for most other messages in the ACS application.

Also *generator* and *reg_caller* objects are client objects in this context, only sending requests to server objects. Most other objects show dual identities, behave like both client and server. All *cc_server*, *cp_server* and *db_server* send nested requests to lower level server for nested services. Messages are buffered if the receiver object is busy, messages sent to the receiver object can request for different services and same requests can be sent to a receiver object many times. Concurrency of the objects are well represented in the model. Hardware servers are not modelled here, but contention for software server is represented.

The OO Performance Model for ACS captures multiple classes of services too. For example, *db_server* has two entries, each one represents a distinctive class of services: to handle query requests from *cc_server* to conduct conferee information search, and to handle translation and routing requests from *cp_server* to search translation and routing tables.

In summary, an OO Performance Model for ACS application can model many system behaviors as good as the detailed simulation model in Chapter 5 and is a promising tool for study-

Figure 6.14. OO Performance Model of ACS Application

ing and analyzing OO system performance quickly. On the other hand, abstraction brings some weaknesses to the model too, that is it cannot be as detailed as a simulation model, for example, resource availability are not considered in the model, line and circuit resources are assumed to be always available, and therefore the performance results obtained from analytic model, also informative, will not be exact.

6.3.3 OO Overhead Representation

The principal performance measures for the model are the throughput for each of its clients. All other measures such as response times, server utilizations, etc. can be derived from client cycle times and other model parameters.

In previous sections, basic building blocks and features of OO Performance Model were described. One question remains: how to combine OO related performance overheads and delays into *OO Performance Model* parameters so that the response time calculated can truly represent the real OO system performance result. The workload parameters used will be the parameters like service time, visit ratio, etc.

The OO related performance overhead metrics such as inheritance delay per step, polymorphic binding delay, delay per context switching, delay per message passing and delay per internal method invocation etc. can usually be obtained through the metrics supplied with the system or obtained by measurements. The delays that occurred when the object is processing a message (namely providing one service) can be obtained by combining these delay metrics figures and the occurrence frequency data. These delays can then be added to this object's service time for this particular service, and thus reflect the calculated performance results. This can be illustrated by Figure 6.15, which is a OO Performance Model for the OO interaction diagram shown on Figure 6.2. In the diagram, different types of message requests are represented by using different arc notation introduced in Section 6.3.1. Different delays due to inheritance method look up, polymorphic binding, messaging and internal method invocation for an object can be added into the service time of the corresponding service offered by this object. For example:

$$\begin{aligned}
 & \text{service time for processing } \underline{msg_1} \text{ in object 4} \\
 &= \text{the actual service time} + \text{messaging delay} + \text{inheritance delay} \\
 &\quad + \text{polymorphic binding delay} + \text{internal method invocation delay}
 \end{aligned}$$

Figure 6.15. Represent OO related overheads in OO Performance Model

And similarly, delay due to context switching can be added into the service time of the corresponding service offered by processor, for example:

$$\begin{aligned}
 &\text{service time for processing } \underline{\text{msg2}} \text{ in cpu} \\
 &= \text{the actual service time} + \text{context switching delay}
 \end{aligned}$$

Potentially, the solutions for the OO Performance Model can be found through the following

methods:

- by a simplified simulation. For example, the OO Performance Model for ACS can be simulated without considering line and circuit resources to obtain performance results for the model.
- by PetriNet solver (detailed description about PetriNet can be found in [48]).
- by an analytical solver so that given the required workload parameters of a Concurrent OO System, the solver will be able to figure out the underlying system performance information automatically. Developing such a solver is a topic for future research.

6.4 Summary

Performance modelling for Concurrent OO system is different from traditional system in some aspects. The performance modelling for OO System requires that objects become the constructing entities or building blocks of the model. Accordingly, model behavior is determined by the OO system characteristics such as object interaction, object dual role as senders and receivers of messages, nested requests and flexible message patterns etc. Concurrency and contention for resources are other key aspects that have to be captured by the performance model. An important requirement is to support a diversity of message request types; some such types identified in the thesis are: *Synchronous Rendezvous*, *Non-blocking Message Sending*, *Message Forwarding* and *Tentative Command Message*.

Among the known analytical performance models, SRVN matches better than others the OO modelling requirements, however, despite many attractive modelling capabilities, SRVN is still missing some features for modelling OO System performance.

Features of a *OO Performance Model* are proposed according to the requirements identified in the thesis. It uses the notion *object* as model building blocks and supports all the object behaviors identified, concurrency among objects, contention requirements as well as multiple type of messages. Notations for representing different message types identified are developed and possible ways to include OO related performance overheads into the model parameters are discussed. OO Performance Model for Automatic Conference Call application is constructed as an example to show what the OO Performance Model looks like and how it can be constructed.

Chapter 7

Conclusions

7.1 Research Summary and Contributions

With the increasing popularity of Object-Oriented system, more and more designers have become aware of the performance drawback of OO system that are caused by various execution overheads due to OO system characteristics. This research is motivated by the tremendous demands from industry that a OO system specific performance modelling and analyzing framework is needed for capturing the OO related overheads and predicting the OO system performance behaviors.

The thesis focuses on analyzing and modelling performance behavior of Concurrent OO system. First the thesis investigated the specific performance characteristics of Object-Oriented system, analyzed those key OO factors that have a strong impact on OO system performance. These factors are inheritance, polymorphic binding, message passing and context switching. The thesis then proposes a set of techniques that can improve OO system performance in general. Among them, the Incremental Performance Monitoring Plan focuses on monitoring system performance early and incrementally, while the set of guidelines are intended to offer designers shortcuts for better performance. Those identified guidelines include: maintaining clean object interface, grouping objects into collaboration group, minimizing messaging and message-passing overhead, adding direct class association, reducing code inheritance level, finding and fixing bottleneck object, rearranging execution sequence, localization of information, avoiding unnecessary polymorphic binding, cloning object and increasing concurrency level.

Based on a system description taken from [11], analysis and design are carried out by applying the techniques identified. The proper system object model is constructed, system messaging behavior and state transition behavior are also captured. Then a simulation model of this real-time OO system is built. The simulation model includes the competition between various objects for hardware and software resources, and different overheads (usually hidden to the programmer). The model represents the workload imposed on a computer system by an OO program, which is the first step toward the construction of a performance model capturing the timing properties of both the software and hardware components of the system, including various OO related overheads. Also statistics are captured and analyzed to reveal potential performance problems.

In the last part of the thesis, requirements for an OO specific analytical performance model are established. These requirements include object interaction behavior, object dual role, nested requests, flexible message pattern, concurrency, resource contention, etc. Also four types of message request types are identified: Synchronous Rendezvous, Asynchronous No-waiting, Message Forwarding and Tentative Command Message. Based on these requirements, extensions to the SRVN model are proposed. The extended modelling features could become potential research directions for the future.

7.2 Future Work

The following future work can be done in the area of performance analysis of Concurrent OO system:

- 1). Find analytical solution methods for the extended OO Performance Model.
- 2). Identify more request types and extend the OO Performance Model to support other classes of OO software.
- 3). Design a simulation tool that could be used specifically for modelling concurrent Object Oriented system.
- 4). Measurements of OO software already built and implemented could be used to validate the modelling approach.
- 5). A method has to be developed to balance system reusability and its performance because of the trade-offs between the two. On one side, reuse relies on heavy usage of inheritance,

dynamic binding, etc. and hence would potentially degrade system performance; on the other side, reusable software saves potential work and increases development speed. For example, we need to find answers for the following questions:

- i). how to measure reusability and the potential savings from it?
- ii). how to compare the trade off between the OO performance and reusability?

References

- [1] R. Agrawal, M.J. Carey, M. Livny, *Concurrency Control Performance Modelling: Alternatives and Implications*, ACM Transactions on Database Systems, Vol 12, No.4, December 1987
- [2] M. Ajmone Marson, G. Balbo and G. Conte, *A class of generalized stochastic Petri Nets for the performance evaluation of multiprocessor systems*, ACM Trans. on Computer Systems, 2(2):93-122, May 1984
- [3] C. Atkinson, *Object-Oriented Reuse, Concurrency and Distribution*, ACM press, Addison-Wesley, Reading, MA, 1991.
- [4] C. Atkinson, S. Goldsack, A. Di Maio, and R. Bayan, *Object-Oriented Concurrency and Distribution in DRAGOON*, *Journal Object Oriented Programming*, Vol. 2, No. 6, March/April 1990, pp. 11-19.
- [5] K. Beck and W. Cunningham, *A Laboratory for Teaching Object-Oriented Thinking*, Proceedings of OOPSLA'89, pp. 1-6, ACM/SIGPLAN, New Orleans, Louisiana, October 1-6, 1989.
- [6] L. Berlin, *When Objects Collide: Experience with Reusing Multiple Class Hierarchies*, Proceedings of OOPSLA/ECOOP'90, Vol. 25, No. 10, pp. 181-193.
- [7] Blum, B. I., *Software Engineering, a Holistic View*, Oxford University Press, Inc. New York, NY, 1992
- [8] S.R. Chidamber, C.F. Kemerer, *A Metrics Suite for Object-Oriented Design*, IEEE Transactions on Software Engineering, Volume 20, June 1994.
- [9] T. Biggerstaff (ed.), *Software Reusability*, Vol.1, ACM Press, Addison-Wesley, Reading, MA, 1989.
- [10] BNR Training Manual, *Object-Oriented Analysis and Design Concepts*, Software Architect, 1993
- [11] BNR Training Manual, PSC Inc. *Introduction to Object-Oriented Technology*, 1993
- [12] BNR Training Manual, PSC Inc. *Object-Oriented Analysis and Design*, 1993
- [13] G. Booch, *Object Oriented Design with Applications*, Benjamin/Cummings, Redwood City, CA, 1991.
- [14] Jean-Pierre Briot, *Object-Oriented Concurrent Programming*, Litp-Universite Paris VI.
- [15] Timothy Budd, *An Introduction to Object-Oriented Programming*, Addison-Wesley, 1991

- [16] Buhr, R.J.A. and R.S. Casselman, *Case Studies in Threads and Contracts*, TR-SCE-92-06, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, 1992
- [17] Buhr, R.J.A. and R.S. Casselman, *Architecture with Pictures*, Proceedings of OOP-SLA'92, pp. 466-483, ACM/SIGPLAN, Vancouver, Canada, October 1992
- [18] R.J.A. Buhr, R.S. Casselman, and B. Foster, *Designing with Timethreads*, TR-SCE-93-05, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, CAN., (*In progress*).
- [19] CCITT, *Functional Specification and Description Language (SDL)*, 1988, 1992.
- [20] CCITT, *SDL User Guidelines*, 1988, Geneva
- [21] Casselman, R.S., *A Role-Based ARchitectural Model Applied to Object-Oriented Systems*, M.Eng. thesis, OCIEE-93-11, Department of systems and computer engineering, Carleton University, 1993
- [22] Casselman, R. S., R.J.A. Buhr and F. Pomerleau, *Notation for Threads*, TR-SCE-92-06, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, 1992
- [23] S. Chandra, J.R. Larus, A. Rogers, *Where Is Time Spent in Message-Passing and Shared Memory Programs*, ASPLOS VI, October 1994
- [24] S.R. Chidamber, C. F. Kemerer, *A Metrics Suite for Object-Oriented Design*, IEEE Transaction on Software Engineering, Volume 20, June 1994.
- [25] P. Coad, "Object-Oriented Patterns," *Communications of the ACM*, Vol. 35, No. 9, September 1992.
- [26] Coad, P. and E. Yourdon, *Object-Oriented Analysis*, Prentice-Hall, Englewood Cliffs, NJ, 1990.
- [27] Cox, B., *Object-Oriented Programming, An evolutionary Approach*, Addison-Wesley, Reading, MA, 1987.
- [28] D.G. Firesmith, *Object-Oriented Requirements Analysis and Logical Design: A Software Engineering Approach*, John Wiley & Sons, Inc., New York, NJ, 1993
- [29] Gane, C. and T. Sarson, *Structured Systems Analysis: Tools and Techniques*, Englewood Cliffs: Prentice Hall, 1978
- [30] D. Halbert and P. O'Brien, "Using Types and Inheritance in Object-Oriented Programming," *IEEE Software*, Vol. 4, September 1987, pp. 71-79.
- [31] P. Heidelberger and K. S. Trivedi, Analytic queueing models for programs with internal concurrence. *IEEE Transactions on Computers*, vol. C-32, pp. 73-82, January 1983.

- [32] R. Helm, I. Holland, and D. Gangopadhyay, "Contracts: Specifying Behavioural Compositions in Object-Oriented Systems," *Proceedings OOPSLA/ECOOP'90*, Vol. 25, No. 10, pp. 169-180. ACM/SIGPLAN, Ottawa, Canada, October 1990
- [33] Hoover, S. V., and Perry, R.F., *Simulation: A Problem Solving Approach*, Addison-Wesley Publishing Co., New York, 1989
- [34] Hufnagel, S.P. and J.C. Browne, *Performance Properties of Vertically Partitioned Object-Oriented Systems*, Software Engineering, Volume 15, August 1989, pp. 935-946.
- [35] D. Ingalls, "A Simple Technique for Handling Multiple Polymorphism," *Proceedings of OOPSLA/ECOOP'86*, Vol. 21, No. 11, pp. 347-349.
- [36] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, *Object-Oriented Software Engineering (A Use Case Driven Approach)*, Addison-Wesley, Reading, MA, 1992.
- [37] R. Jain, *Art of Computer Systems Performance Analysis*, New York: Wiley, 1990
- [38] R. E. Johnson and B. Foote, *Designing Reusable Classes*, Journal of Object-Oriented Programming, Vol. 1, No. 2, June/July 1988, pp. 22-25.
- [39] A. Kapelnikov, R. R. Muntz, and M. D. Ercegovac, *A methodology for performance analysis of parallel computations with looping constructs*, Journal of Parallel and Distributed Computing, no. 14, pp. 105-120, 1992.
- [40] A. Kapelnikov, R. R. Muntz, and M. D. Ercegovac, *A modeling methodology for the analysis of concurrent systems and computations*, Journal of Parallel and Distributed Computing, no. 6, pp. 568-597, 1989.
- [41] S. S. Lavenberg, *A Perspective on Queueing Models of Computer Performance*, Performance Evaluation, vol. 10, pp. 53-76, 1989.
- [42] E. D. Lazowska, J. Zahorjan, G. S. Graham and K. C. Sevcik, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*, Prentice-Hall Inc., Englewood Cliffs, N.J. 07632, 1984
- [43] B. Meyer, *Eiffel: The language*, Prentice-Hall, Englewood Cliffs, NJ, 1992.P.
- [44] Meyer, B., *Object-Oriented Software Construction*, Prentice-Hall, Englewood Cliffs, NJ, 1988.
- [45] J. W. Miernik, C. M. Woodside, J. E. Neilson and D. C. Petriu, *Throughput of stochastic rendezvous networks with caller-specific service and processor contention*, in *Proceeding IEEE Infocom '88*, pp. 1040-1049, March 1988.
- [46] J. W. Miernik, C. M. Woodside, J. E. Neilson and D. C. Petriu, *Throughput of stochastic rendezvous networks with priority tasks*, in *Performance of Distributed and Parallel Systems* (T. Hasegawa, H. Takagi and Y. Takahashi, eds.), pp. 511-525, Elsevier Science Pub-

lisher, 1989.

- [47] M.K. Molloy, *Fundamentals of Performance Modelling*, New York: Macmillan, 1989
- [48] M. K. Molloy, *Performance analysis using stochastic Petri Nets*, IEEE Trans. on Computers, C-31:913-917, September 1982
- [49] Naylor, T.H., and Finger, J.M., *Verification of Computer Simulation Models*, Management science, Vol. 14, 1967, pp.92-101.
- [50] J. E. Neilson, *PARASOL Users' Manual*, School of Computer Science, Carleton University, Ottawa, Canada.
- [51] J. E. Neilson, *PARASOL Tutorial*, School of Computer Science, Carleton University, Ottawa, Canada.
- [52] J.E. Neilson, C.M. Woodside, D.C. Petriu, S. Majumdar, *Software Bottlenecking in Client-Server Systems and Rendezvous Networks*, SCE-92-17, Department of Systems and Computer Engineering, Carleton University, 1992.
- [53] H. Nassbaumer, *Computer Communication Systems, Volume 1*, Wiley and Sons, New York, NY, 1990.
- [54] Dartzen Peng and Kang G. Shin, *Modeling of concurrent task execution in a distributed system for real-time control*, IEEE transactions on computer, vol. C-36, N). 4, April 1987.
- [55] D. C. Petriu, *Approximate solution for stochastic rendezvous networks by Markov chain task-directed aggregation*, Ph. D thesis, SCE-91-21, Department of systems and computer engineering, Carleton University, 1991
- [56] D. C. Petriu, *Approximate Mean Value Analysis of Client-Server Systems with Multi-class requests*, Proceedings of SIGMETRICS'94 Conference on Measurement & Modelling of Computer Systems (Nashville, Tennessee), pp. 77-86, May 1994.
- [57] D. C. Petriu and C. M. Woodside, *Approximate MVA for software client/server model by Markov chain task-directed aggregation*, In Proc. of The Third IEEE Symposium on Parallel and Distributed Processing, (Dallas, Texas), December 1991.
- [58] D. C. Petriu and C. M. Woodside, *A New Mean Value Analysis of Client-Server Software by Task-Directed Aggregation*, SCE-93-29, Department of systems and computer engineering, Carleton University, 1993
- [59] D. C. Petriu, S. Majumdar, Jingping Lin and Curtis Hrischuk, *Analytic Performance Estimation of Client-Server Systems with Multi-Threaded Clients*, SCE-93-15, Department of systems and computer engineering, Carleton University.
- [60] Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.

- [61] C.H. Sauer and K. M. Chandy, *Computer Systems Performance Modelling*, Englewood Cliffs, NJ: Prentice-Hall, 1981
- [62] Selic, B. and McGee, J. , *Object-Oriented Design Concepts for Real-Time Distributed systems*, Real-Time and Embedded System Workshop, ACM-OOPSLA'91
- [63] Selic, B., G. Gullekson, J. McGee and I. Engelberg, *ROOM: An Object-Oriented Methodology for Developing Real-Time Systems*, 1992
- [64] Shlaer, S. and S. Mellor, *Object Lifecycles, Modelling the World in States*, Prentice-Hall, Englewood Cliffs, NJ, 1992.
- [65] Shumate, K., Keller, M., *Software Specification and Design, A Disciplined Approach for Real-time Systems*, John Wiley & Sons Inc., 1992
- [66] SiliconGraphics Computer Systems, *C++ Language System Overview*, November 1992
- [67] SiliconGraphics Computer Systems, *C++ Language System Product Reference Manual*, November 1992
- [68] SiliconGraphics Computer Systems, *C++ Language System Library*, November 1992
- [69] Sinha, Alok, *Client-server computing*, CACM, vol. 35, July 1992.
- [70] Smith, Connie U. , *Performance Engineering of Software Systems*, Addison-Wesley Publishing Company, Inc., 1990
- [71] Smith, Connie U. 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
- [72] W. Stallings, *Data and Computer Communications, Second Edition*, Macmillan Publishing, New York, NY, 1988.
- [73] Tanenbaum, A. S., *Modern Operating Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1992.
- [74] Thomasian, A. and P. Bay, *Analytic queueing network models for parallel processing of task systems*, *IEEE Transactions on Computers*, vol. C-35, pp. 1045-1054, December 1986.
- [75] Thomasian, A., "Performance analysis of task systems using a queueing network model," *IEEE*, pp. 234-242, 1985.
- [76] V. Vaishnavi, *Object-Oriented Metrics, Developer's Manual*, Georgia State University, January 1994.
- [77] Wegner, Peter, *Dimensions of Object-Oriented Modelling*, ACM Computer,

- [78] Woodside, C. M. , *Throughput Calculation for Basic Stochastic Rendezvous Networks*, 1987, Performance Evaluation, vol. 9, pp. 143-160, North-Holland, 1989
- [79] Woodside, C. M., J. E. Neilson, D. C. Petriu and S. Majumdar, *The rendezvous network model for performance synchronous multi-tasking distributed software*, SCE-89-8, Dept. of Systems and Computer Engineering and school of Computer Science, Carleton University, March 1989.
- [80] Woodside, C. M., *A Three-View Model for Performance Engineering of Concurrent Software*, SCE-93-19, Department of systems and computer engineering, Carleton University, 1993
- [81] Wirfs-Brock, R., B. Wilkenson, and L. Wiener, *Designing Object-Oriented Software*, Prentice-Hall, Englewood Cliffs, NJ, 1990.

Appendix A

Notations

A.1 TimeThreads (Buhr)

Timethreads have proven useful for specifying and designing concurrent systems [18]. Here a subset of the timethread notations are included.

Figure A.1. Timethread Notations

A.2 OMT Notation (Rumbaugh)

Figure A.2. Object Model

A.3 Context Diagram

Figure A.3. Context Diagram

A.4 State Transition Diagram

Figure A.4. Dynamic Model

A.5 Message Sequence Charts (MSCs)

Figure A.5. Message Sequence Chart Notations

A.6 Execution Graph Notation (Connie Smith)

Figure A.6. Execution Graph

END

3 1 - 0 5 - 9 5

FIN