1

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

PRECISIONSM RESOLUTION TARGETS

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

**The Automatic Generation of
Software Performance Models From a Prototype**

by

Curtis E. Hrischuk

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

Master of Engineering

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

6 January 1995

© Copyright 1994, Curtis E. Hrischuk

National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services Branch

Direction des acquisitions et
des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

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 0-315-98600-X

Canada

Name

CURTIS HRISCHUK

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	0727
Early Childhood	0518
Elementary	0524
Finance	0277
Guidance and Counseling	0519
Health	0680
Higher	0745
History of	0520
Home Economics	0278
Industrial	0521
Language and Literature	0279
Mathematics	0280
Music	0522
Philosophy of	0998
Physical	0523

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

LANGUAGE, LITERATURE AND LINGUISTICS

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

PHILOSOPHY, RELIGION AND THEOLOGY

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

SOCIAL SCIENCES

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

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

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture	
General	0473
Agronomy	0285
Animal Culture and	
Nutrition	0475
Animal Pathology	0476
Food Science and	
Technology	0359
Forestry and Wildlife	0478
Plant Culture	0479
Plant Pathology	0480
Plant Physiology	0817
Range Management	0777
Wood Technology	0746
Biology	
General	0306
Anatomy	0287
Biostatistics	0308
Botany	0309
Cell	0379
Ecology	0329
Entomology	0353
Genetics	0369
Limnology	0793
Microbiology	0410
Molecular	0307
Neuroscience	0317
Oceanography	0416
Physiology	0433
Radiation	0821
Veterinary Science	0778
Zoology	0472
Biophysics	
General	0786
Medical	0760
EARTH SCIENCES	
Biogeochemistry	0425
Geochemistry	0996

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	
General	0566
Audiology	0300
Chemotherapy	0992
Dentistry	0567
Education	0350
Hospital Management	0769
Human Development	0758
Immunology	0982
Medicine and Surgery	0564
Mental Health	0347
Nursing	0569
Nutrition	0570
Obstetrics and Gynecology	0380
Occupational Health and	
Therapy	0354
Ophthalmology	0381
Pathology	0571
Pharmacology	0419
Pharmacy	0572
Physical Therapy	0362
Public Health	0573
Radiology	0574
Recreation	0575

Speech Pathology	0460
Toxicology	0383
Home Economics	0386

PHYSICAL SCIENCES

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

Applied Sciences

Applied Mechanics	0346
Computer Science	0984

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

PSYCHOLOGY

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

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

**The Automatic Generation of
Software Performance Models From a Prototype**

submitted by Curtis E. Hrischuk, B. Sc., B.Eng.
in partial fulfillment of the requirements for
the degree of Master of Engineering

Chair, Department of Systems and Computer Engineering

Thesis Supervisor

Thesis Supervisor

Carleton University
9 January 1995

Abstract

Early performance estimates for a new software system aid the design process by providing feedback when design decisions can be easily revised. Unfortunately, constructing a performance model of a distributed and concurrent software system can require significant effort. An automated performance model generation technique is described that reduces the model building effort by providing: easy specification of performance experiments, empirical estimates for model parameters, automated model generation, and support for different types of models. A prototype is used to describe a software system, from which causal traces (*angio traces*) are recorded during execution. These traces are then processed into sequences of resource demands (*workthreads*), aggregated into system execution descriptions (*workthread classes*), and combined to generate a performance model. The technique can also be applied at other stages of the development process, including the redesign of existing software.

Acknowledgments

I wish to thank my wife Karen, my daughters Bethany and Amanda, and my son Logan for their support and understanding during this work. You have been a comfort and an inspiration.

I would like to thank my supervisors, Professor Murray Woodside and Professor Jerome Rolia, for their guidance and critique throughout the studies.

In addition, I wish to thank the performance group for their feedback and encouragement. Your discussion has prompted this work.

A special note of thanks to Professor Gerald Karam for responding to my “Yet Another Request” for MLog. He also suggested the angiogram analogy for the angio traces.

Lastly, I will acknowledge my source for finding “out knowledge of witty inventions” (Proverbs 8:12, KJV): thanks to the transcendent God who creates, sustains, and completes.

Table of Contents

Chapter 1.0	Introduction	1
1.1	Motivation	1
1.1.1	Applicability of a Performance Model	1
1.1.2	Constructing a Performance Model	3
1.2	Problem Statement	4
1.3	Contributions to Research	5
1.4	Thesis Organization.....	6
Chapter 2.0	Background.....	8
2.1	Software Performance Engineering	8
2.2	Performance Models	12
2.2.1	Performance Terminology	13
2.2.2	Analytic Queueing Network Solution Techniques	14
2.2.3	Layered Queueing Network Model	16
2.3	Prototypes as Early Executable Designs	19
2.3.1	Several Prototype Environments	21
2.3.2	Prototyping in this Thesis	23
2.3.3	Using Prototypes as Simulation Models.....	24
2.4	Measurement and Tracing.....	25
2.5	Background Summary.....	27
Chapter 3.0	Overview of the Model Building Technique	28
Chapter 4.0	Angio Trace Generation	33
4.1	Angio Traces	33
4.2	Angio Event Types.....	35

4.2.1	Beginning and Ending Traces.....	35
4.2.2	Concurrent Execution	36
4.2.3	Task, Phase, and Service Identification	38
4.2.4	Delay	39
4.2.5	Resource Management.....	40
4.2.6	Documentation.....	40
4.2.7	Event Type Summary	40
4.3	MLog Angio Trace Instrumentation	41
4.3.1	MLog Angio Instrumentation	42
4.3.2	MLog Angio Trace Filtering.....	47
4.3.3	Backtracking	49
4.4	Trace Generation Summary	50

Chapter 5.0 Model Generation.....52

5.1	Angio Trace(s) to Workthread	52
5.2	Workthread to Workthread Class.....	54
5.2.1	Recognizing a Rendezvous.....	57
5.2.2	Recognizing a Task Phase Change	58
5.2.3	Recognizing Forwarding.....	59
5.3	Workthread Classes to Performance Model.....	62
5.4	Workthread to Workthread Class Algorithm	64
5.4.1	Trace Normalization	66
5.4.2	Algorithm for Generating the Workthread Class.....	71
5.4.3	Task Creation Algorithm	72
5.4.4	Resource Demand Aggregation	73
5.4.5	Entry-Entry Call Aggregation Operation.....	74
5.4.6	Workthread Class Parameter Averaging.....	78
5.5	Model Generation Summary	79

Chapter 6.0	Structural Transformations	80
6.1	Workthread Class Clustering	80
6.2	Movement of Entries	82
6.3	Class Cloning and Resource Reassignment	83
6.4	Structure Transformations Summary	85
Chapter 7.0	Workthread Case Study	86
7.1	Prototype of an OLTP System	86
7.2	Angio Traces of Scenarios	90
7.2.1	Validation Command Scenario Trace.....	91
7.2.2	Application Command Scenario Trace.....	92
7.3	OLTP Workthreads	95
7.4	OLTP Workthread Classes.....	96
7.5	The OLTP Layered Queueing Model.....	100
7.6	Experimenting with the OLTP Workthread Classes	103
7.7	Conclusions	104
Chapter 8.0	Summary and Conclusion	107
8.1	Summary of Results	107
8.2	Implementation Considerations.....	110
8.3	Using the Technique.....	112
8.4	Resource Function Characterization	114
8.5	Future Research.....	115
References.....		117

List of Figures

Figure 1:	LQM Components	17
Figure 2:	Model Generation Technique.....	29
Figure 3:	wSpawn Example	37
Figure 4:	Naming Convention Example.....	39
Figure 5:	Example MLog Method.....	44
Figure 6:	wBegin Angio Probe.....	45
Figure 7:	wSpawn Angio Probe	45
Figure 8:	Angio Instrumentation of the waitToAccept Method	46
Figure 9:	Angio Instrumentation of the performAsync Method	47
Figure 10:	MLog Angio Trace Example	47
Figure 11:	Example of Loop Using Backtracking.....	49
Figure 12:	Workthread Event Components	53
Figure 13:	Workthread Class Parameters	55
Figure 14:	Workthread Class Components.....	56
Figure 15:	Nested Rendezvous Example.....	57
Figure 16:	Forwarding Example.....	60
Figure 17:	Task Pipeline.....	61
Figure 18:	Individual Workthread Classes	63
Figure 19:	Workthread Class Model and LOM.....	63
Figure 20:	Algorithm Notation.....	65
Figure 21:	Resulting Model Parameters	67
Figure 22:	Normalized Trace of Forwarding.....	68
Figure 23:	Execution of Rendezvous Using Asynchronous Communication.....	69
Figure 24:	Traces of Rendezvous Using Asynchronous Communication.....	70
Figure 30:	Individual Workthread Classes	81
Figure 31:	Clustered Workthread Classes	81

Figure 32:	Vertical Movement of Entries	83
Figure 33:	Horizontal Movement of Entries	83
Figure 34:	Service Fusion of Entries	84
Figure 35:	Resource Assignment of Multiple Workthread Class Instances	84
Figure 36:	OLTP System Configuration.....	88
Figure 37:	Validation Command Scenario Execution and Angio Trace	91
Figure 38:	Application Command Scenario Execution.....	94
Figure 39:	Application Command Scenario Angio Trace	95
Figure 40:	Validation Command Workthread.....	96
Figure 41:	Application Command Workthread	97
Figure 42:	Normalized Application Command Workthread	98
Figure 43:	Validation Command Workthread Class.....	98
Figure 44:	Application Command Workthread Class	99
Figure 45:	OLTP Workthread Class Model and LQM	102
Figure 46:	LQM for Horizontal and Vertical Entry Movement	105
Figure 47:	LQM Resource Demands with Reassignment.....	105
Figure 48:	Model Generation Technique.....	108

List of Tables

Table 1:	Performance Model Type Comparison	12
Table 2:	Argio Trace Event Type Summary.....	35
Table 3:	Angio Event Type Parameters	41
Table 4:	MLog Methods that are Ignored	48
Table 5:	OLTP Workload Intensity	88
Table 6:	OLTP Performance Model Resource Definitions	88
Table 7:	OLTP Task Information	89
Table 8:	Statement IDs and Resource Functions for the OLTP System	89
Table 9:	OLTP Workthread Class Resource Demands	100
Table 10:	OLTP Estimated Resource Utilizations	102
Table 11:	OLTP Application Task Concurrency and Response Time	103

Index of Definitions and Use

A	
active objects	22
angio event.....	30
angio primitive method.....	41
angio probe	41
angio trace.....	5, 30
arity	42
B	
backtracking.....	49
C	
calling	16
causal trace.....	26
closed class	13
component	66
concurrency level.....	64
customer.....	13
customer class.....	13
D	
design prototyping	19
dye	30
E	
empirical performance model parameters	3
entry movement	80
environment information	31
event type.....	34
F	
forwarding	59
frequency parameters.....	3
H	
horizontal entry movement.....	82
I	
idle time	13

	infinite server.....	14
L		
	Layered Queueing Model	16
	LQM	16
M		
	Mean Value Analysis.....	14
	MLog	22
	multi-server.....	14
O		
	occurrence number.....	36
	OLTP	86
	open class.....	13
P		
	performance measures	13
	performance model	9
	performance objectives.....	28
	polymorphism.....	22
	post-rendezvous	17
	predictive performance model parameters.....	3
	prototyping.....	19
	prototyping environment	21
R		
	rendezvous	16
	residence time	14
	resource.....	13
	resource assignment.....	64
	resource function	30, 52
	response time	13
S		
	scenario.....	28
	scenario name	28
	scenario options	28

scenario task	55
server type.....	14
service aggregation property.....	52, 54
service demand	14, 52
service entry.....	16, 30
service entry fusion.....	82
service entry phase.....	17
service time.....	13
single inheritance.....	22
single server	14
software execution model	9
sp.....	9
spawn count	38
specification prototyping	19
statement.....	29
statement instrumentation.....	41
Structure and Performance Specification Method....	9
super-class	44

T

task.....	16
task identifier	33
taskEnter	39
taskPhase	39
taskService	39
tesource arrival rate.....	14
think time	13
throughput.....	14
trace	25, 30
trace context.....	43
trace name.....	29, 36

	trace normalization	66
U	utilization	14
V	vertical entry movement	82
	visit	66
W	wBegin.....	35
	wDelay.....	39
	wEnd.....	35
	workload intensity	13
	workthread	52
	workthread class	5, 30, 54
	workthread class cloning	80
	workthread class clustering.....	80
	workthread.	5
	wReport	40
	wSpawn	36

Chapter 1.0 Introduction

“The average software development project overshoots its schedule by half; larger projects generally do worse. And some three quarters of all large systems are ‘operating failures’ that either do not function as intended or are not used at all.” Scientific American [83]

1.1 Motivation

Ensuring a software system meets its performance objectives is a significant problem. This problem becomes even more difficult to solve for concurrent and/or distributed software systems. *Software performance engineering* prevents and solves performance problems by analyzing a software system and suggesting corrective measures when necessary. Rather than depending entirely on measurement, the approach advocated by this thesis is to use a performance model in the analysis of a software system’s average performance behavior. The topic of this thesis is a technique to automatically generate a performance model.

1.1.1 Applicability of a Performance Model

System level performance is often neglected in the early stages of development a software system. Smith succinctly describes the increasing importance of the performance analysis of software systems:

“Performance refers to system responsiveness: either the time required to respond to specific events, or the number of events processed in a given time interval. For traditional information systems, performance considerations are associated with usability issues such as response time for user commands. For ‘soft’ real-time systems, performance considerations relate

to the ability to adequately handle the external load placed on the system. In 'hard' real-time systems, performance concerns can become correctness issues. In these systems, failure to meet deadlines or throughput requirements is equivalent to producing incorrect results [75]."

Anderson [4] describes an example where the analysis of the average performance resulted in an efficient design for a real-time information system. Smith [74] documents several other instances where such an analysis would have been useful. The analysis may make use of experience, measurement, or performance modeling. Each of these approaches will now be considered.

The first approach to performance analysis can be based on experience, intuition or trend extrapolation [23]. This approach can be useful but it is difficult to extend for large systems or non-heterogenous environments. It is only possible if something similar to the proposed system already exists and an "expert" is available.

Another approach to performance analysis is to measure a system's performance during an experiment or under normal use [23]. Measurements are trustworthy since they are taken from the final (or a similar) system. However, even with a small or medium size system it may not be possible to isolate the system variables and behavior to allow generalization of the measurement data [42]. This approach may be impractical due to the system size, expense, or lack of availability.

The last approach uses a performance model of the system during the analysis. The performance model's structure is defined, the behavior is characterized, parameters are assigned to the model structure, and it is evaluated to provide performance estimates for the analysis. Performance modeling is more reliable than intuition because it can produce estimates that can be verified experimentally to validate the model. It is less laborious and more flexible than measurements because the model operates on the key aspects of the system. If it is a new concept being analyzed, modeling is the only techniques from which

to choose [76]. Performance “modelling, provides a basis for gathering, organizing, evaluation, and understanding information about a computer system [23].”

Performance modeling can be used as an aid to the other methods of analysis [23]. Modeling gives credence to an analyst’s intuition because a model allows investigation of the behavior of a system under a wide range of alternatives. Modeling can give guidance for measurements by indicating which experiments are most useful.

1.1.2 Constructing a Performance Model

Gathering information about a system’s structure and execution is an obstacle to constructing a performance model. The two main approaches to developing the structure and parameters of a performance model are *empirical* and *predictive*.

The fully empirical approach is analogous to measurement based performance analysis, in which: the system is implemented in some form and its performance is measured. If warranted, adjustments are made to the system to enhance its performance. This is possible only in the latter stages of design with a nearly complete system definition and some form of implementation. This restricts the empirical approach to a stage in the development process when important decisions have been made and are costly to alter.

The predictive approach characterizes software execution patterns, constructs a performance model to generate performance estimates, and then uses the predictions to guide modifications. This gives earlier warnings and supports more effective adjustments, which can be followed by an empirical study of the completed software. A weakness of the predictive approach arises in determining the model structure and parameters without an implementation as a guide. Model parameters may depend on system data values that interact in complex ways, which are difficult to estimate and are resolved only during execution. This is particularly true of *frequency parameters*, such as data-dependent choices and loop conditions, which are difficult to estimate with a fully predictive approach.

Both the empirical and predictive approaches expend considerable effort and cost to construct the model. Usually a model is constructed from a non-existent or small information base, by a performance modeling expert. The model construction is laborious, involving the examination of design documents and the source code to extract the model structure and parameters.

A software development methodology has been proposed [5] which uses an early executable design for functional verification and a separate performance model for performance analysis. The performance model is constructed using the predictive approach and suffers from its weaknesses.

It should be possible to exploit the executable design for model construction purposes, more directly. This thesis will show that an executable design can be used to empirically provide parameter values (e.g., frequency parameters) which can then be used in a predictive model.

In summary, there is a need for a general and systematic method of constructing a performance model that can appraise data-dependant parameters and the necessary structural content.

1.2 Problem Statement

The problem this research proposes to solve is:

“Given an executable software design description, construct a Layered Queueing Network performance model which includes structural information and data-dependent frequency parameters.”

The approach presented in this thesis uses an executable design, in the form of a prototype, to obtain parameter estimates by combining the empirical and predictive approaches. The software system is prototyped at an early stage so that the main execution paths are defined, but the full functional detail of the paths may be incomplete. Sufficient detail is included to estimate the model parameters of the execution paths expected to affect

performance most. Instrumentation is added to the prototype to indicate where a scenario begins. Scenarios of typical end-to-end operations are executed and are recorded in a tracing format referred to as an *angio trace* which maintains the causal relationships of events. The *angio trace* provides enough information to construct the sequence of resource demands, called a *workthread*. The *workthread* is then aggregated into a *workthread class* which characterizes the average resource demands and task interactions of a scenario. A performance model can then be generated by combining several *workthread classes*. Its service time parameters are still found as in the predictive approach but its frequency parameters are empirically determined. Fully asynchronous task interactions are not currently supported by the technique.

The technique integrates work from four different research areas: software performance engineering approaches, computer system performance modeling, prototyping, and tracing techniques.

1.3 Contributions to Research

The main contribution of this work is *the development of a technique that automatically constructs an analytic performance model for Software Performance Engineering use*. Other contributions that enable the main contribution or allow it to be more easily applied are:

- *Demonstration of the technique.* An analytic performance model of a client-server system is constructed from an MLog prototype [29].
- *Tools to support the technique.* This included the development of *angio tracing* for MLog and of a program that reduced *angio traces* into *workthread classes*.
- *Definition of a form of tracing suitable for constructing performance models.* *Angio tracing* can record characteristics of different software architectures, software development approaches, and implementation technologies.

- *Definition of a software system description for model experimentation.* The workthread class characterizes behavioral and structural information of a software system sufficiently for performance modeling purposes. It allows the analyst to manipulate the model structure in an intuitive manner for experimental purposes.
- *Automated transformations of traces to models.* Description of automated transformations that begin with an angio trace and construct an analytic performance model.

The significance of these contributions is:

- *They can be used with several software development methodologies.* Supports an iterative software development cycle, as well as the traditional waterfall life-cycle.
- *They provide representative, timely performance models.* The model transparently tracks the forefront of the development, providing more precise performance estimates as the prototype is refined. The advantages of the technique are most fully realized if it is part of an evolutionary prototyping approach based on stepwise refinement from abstract models to detailed realistic products [56], or an incremental prototyping approach where the system is based on one overall design but built incrementally [34]. This permits the analyst to focus on the principles of Software Performance Engineering rather than model building.
- *They provide a framework to allow for the reuse of performance information.*

1.4 Thesis Organization

Chapter 2 provides background material for the topics which are involved in the transformation steps. Chapter 3 is an overview of the model generation technique. Chapter 4 describes the angio event structure, angio trace format, and an implementation of angio tracing. Chapter 5 describes the transformation steps from an angio trace to a performance

model. Chapter 6 discusses possible performance experiments that can be considered once an initial model is developed. Chapter 7 introduces the case study of an On-Line Transaction Processing software performance engineering problem. Chapter 8 gives conclusions.

Chapter 2.0 Background

"Progress, far from consisting in change, depends on retentiveness. Those who cannot remember the past are condemned to fulfill it." George Santayana

There are four research categories that were synthesized to develop the model generation technique. They are introduced in the following order: software performance engineering approaches, computer system performance models, prototyping, and tracing techniques.

2.1 Software Performance Engineering

Performance analysis is usually applied after a system has been designed or installed, in an attempt to solve a performance issue. This "fix-it later" attitude has proven to be costly and unacceptable in practice, and Smith [74] provides examples. A better approach is to use Software Performance Engineering (SPE) to ensure the performance expectations of a computer system are met. A guiding principle for Software Performance Engineering is that it should begin during the initial conception of a system and continue through its entire lifecycle.

SPE uses a performance model to understand the performance characteristics of a software system. SPE approaches can be characterized by the manner in which a performance model is constructed, either in an empirical or predictive fashion. Four SPE approaches are discussed to compare their performance model construction techniques.

SYSREM\SREM [1] [67] is a software specification methodology which uses an empirical approach to constructing a performance model. The Requirements Statement Language [2] is used to specify stimulus-response combinations in a fashion similar to an

extended finite state machine. The approach provides module decomposition, concurrency, and resource allocation. The analysis assumes that a single extended finite state machine is active at any time. SYSREMSREM uses simulation for functional and performance verification. Once algorithms for system operations are supplied and performance budgets are allocated to components of the extended finite state machine, a simulation is generated to verify response times for a stimulus-response relationship. This performance information is available after most design decisions have been made because a complete system definition is needed for the simulation. The late availability of performance estimates is a characteristic of the empirical approach.

Three performance engineering approaches which take a predictive approach to model construction are Smith's SPE approach [74] [75], the *Structure and Performance Specification Method (sp)* [81], and the software development methodology of Baker, Chester, and Yeh [5]. With the predictive approach, two models are constructed: a *software execution model* and a *performance model*. The software execution model provides parameters for the performance model, which are then used to produce performance estimates.

Smith's SPE approach uses an *execution graph* to represent the software execution. The constructed performance model is an abstract queueing model that is called an *Information Processing Graph* [70]. An execution graph describes the software execution in a fashion similar to a flow-chart. To derive an execution graph the system documentation or implementation is analyzed to provide estimates for the number of times each software component executes, and the resource usage of each component. The execution graph provides the workload parameters for the Information Processing Graph. The Information Processing Graph is then applied to a queueing network modeling tool to produce performance estimates.

The *Structure and Performance Specification Method (sp)* [85] uses a model which characterizes software execution as abstract data types that call subcomponents a certain

number of times (i.e. a call-graph). The number of calls on the primitive, bottom-level components which result from higher level operations are estimated and then used as parameters in a corresponding performance model. For example, an *sp* model can provide visit counts in a queueing network model. Each component is assigned hardware resource demands which are of three types: communications, processing, and memory use. *Complexity functions* are used to specify hardware demands in a manner which is independent of the environment and workload. The additional information that is needed to generate performance estimates are: 1) a workload model to map component resource demands to actual resources and 2) the hardware device characteristics (e.g., CPU speed).

Baker, Chester, and Yeh [5] have proposed a software development methodology that adopts the predictive approach of software performance engineering. The analyst describes resource usage in a fashion similar to complexity functions and constructs a hierarchical performance evaluation model. A queueing model is then constructed with this information. Prototyping is used for functional verification of a design but was not used in developing the performance model.

There are several concerns which are common to the various SPE approaches:

- *Semantic gaps between the system description, software execution model, performance model, and the implementation.* Both the empirical approach and the predictive approach must bridge the gap between the system description and a performance model. The predictive approach also constructs a software execution model which introduces another layer of abstraction. The relations between the system description, software execution model, performance model, and implementation are loose and informal. Changes to the system need to be reflected in the corresponding software execution and performance model, which is not (necessarily) straight-forward.
- *Model parameters can be difficult to estimate or measure.* Software execution may depend on system data values which are difficult to predict and are resolved

only during execution. This is particularly true of data and resource dependencies which affect choices and loops during execution.

- *Model structure can be difficult to abstract and describe.* Extracting the structural information from a system description, describing it in the software execution model, and including it in to the performance model is not well addressed by the SPE approaches. Identifying the software components which interact can be difficult since they may not be resolved until execution.
- *A standard input format is missing.* Each SPE approach began with a system description. A general, common input data format that allows performance information to be used or interchanged amongst the approaches should be attainable.
- *Model construction is hampered by the lack of a general framework for model parameterization.* Such a framework should provide a single interface that allows various factors to be incorporated into the model, such as software architecture [27], software design approach (e.g., object-oriented programming [66]), and implementation technologies (e.g., client-server [73], multi-threaded tasks [39]). The framework should also be flexible enough to study a new software system's performance behavior under alternative designs and situations, especially at an early stage when design decisions can be easily revised. As a design evolves, increasing detail should be easily incorporated into the framework to allow more complex performance concerns to be examined. The framework should be general enough to construct different types of performance models, as suggested by Ammar [3]. This would allow advances in performance model solution techniques to be incorporated into the framework.

This research addresses these concerns. A special type of trace which contains structural information provides a standard input format. Execution which depends on data dependencies is recorded by the trace and reflected in the corresponding model parameters.

A series of transformations is performed on the trace to bridge the semantic gaps of the models. The transformations produce a characterization that is suitable for storage and for conducting performance experiments.

2.2 Performance Models

There are many types of models which produce performance estimates, such as those shown in Table 1. Different models are used at different stages in the development of a software system. Selecting a model type depends upon the information available, the goal of the investigation, and the stage in the system's development. The workload estimate [23] can provide an estimate of resource consumption during system analysis when high-level operations are being defined. Once resource use has been assigned a Queueing Network Model (QNM) can be used to estimate the effect of resource competition [23]. More sophisticated models incorporate contention for passive resources and the software structure (e.g., Layered Queueing Network Model [19] [41]).

Model Type	Model Detail	Example Questions
Workload Estimate	Resource demands of scenarios	Will the resources support the load, or will they saturate?
Queueing Network	Resource demands of scenarios assigned to resources	Will contention for resources result in the user response times to become too long?
Layered Queueing Network	Resource demands of scenarios assigned to tasks and resources	Does contention within the software task architecture result in the user response times becoming too long?

Table 1: Performance Model Type Comparison

The next subsection provides a review of performance terminology. This is followed by an introduction that discusses exact and approximate solution techniques for analytic queueing models. The Layered Queueing Network Model is used in the case study, so it is discussed in the last subsection. For further information about queueing systems with applications to computer performance estimation, the reader is referred to [76].

2.2.1 Performance Terminology

Terminology will now be reviewed, which is taken from [42] and [74]. A resource has a queue of jobs waiting for it, and provides service to those jobs. For this research, a *resource* is a device or software system component that is used to perform work. A resource can be either active or passive. An *active resource* is a component which actively executes [70], such as a CPU. A *passive resource* does not provide a service but is acquired before other resources can be used; it is held during these services [76]. An example of a passive resource is a semaphore.

A *customer* is a client which requests service from one or more resources. A *customer class* characterizes a number of customers that can be viewed as behaving identically. In an *open class*, the customer population is assumed infinite with a specified rate of arrival (e.g., a transaction workload). A *closed class* is a fixed population of customers which circulate through the system, with a delay that each customer observes before submitting its next request (*think time*). An example of a closed class is a terminal workload with a limited number of active terminals. The *workload intensity* [23] specifies the workload of a performance model using the parameters of the open and closed customer classes.

In this work, the *performance measures* quantitatively describe the average behavior of the system. The performance measures which are most frequently cited are:

- *Response time*: It is the interval between: a) the end of a request submission and the beginning of the corresponding response from the system, or b) as the interval between the end of a request submission and the end of the corresponding response from the system.
- *Idle time*: The period during which a resource is not being used.
- *Service time*: The average time, per visit, that a customer requests from a resource.

- **Residence time:** The average total time a customer spends at a resource, including waiting and service time.
- **Throughput:** The rate at which the requests are completed by the system.
- **Utilization:** The fraction of time a resource is busy servicing requests.
- **Service demand:** The average total amount of time a customer requires service at an active resource, per customer request. The product of the number of visits to a resource and the mean service time at that resource.
- **Resource arrival rate:** The rate at which customers arrive for service at a resource.
- **Server type:** This work considers three possibilities: *single server*, *multi-server*, and an *infinite server*. Single server resources provide service to a single customer. Multi-server resources can service a fixed number of customers concurrently. An infinite server is a resource for which there is no queue but a specified service time.

Having presented the terminology, analytic queueing models are now discussed.

2.2.2 Analytic Queueing Network Solution Techniques

Analytic queueing models [50] provide steady-state performance estimates. The solution techniques for analytic queueing models are classified according to the mathematical accuracy of their estimates, either a mathematically exact or an approximate solution. “Exact” refers to how the solution relates to the performance model, not how the performance model predictions relate to the system being modelled. The Layered Queueing Model, which is discussed in the next section, uses an approximate solution technique. This section discusses the motivation for approximate solutions, as compared to exact ones.

Exact analytic queueing models use *Mean Value Analysis* (MVA) to economically produce performance estimates. The type of queueing network model solved using MVA

is called a *separable queuing network* model. "In an intuitive sense, a separable network has the property that each resource acts (largely) independently of the others" [23]. There are five assumptions which guarantee that a system is separable and can be characterized as an MVA model. The reader is referred to [23] for a description of MVA and these assumptions.

Exact analytic models allow several different resource scheduling policies, such as: processor sharing (PS), first-come-first-served (FCFS), and last-come-first-serve preemptive resume (LCFSPR). FCFS at a resource requires exponentially distributed service times with the same mean service time for all customer classes at that resource. With the PS queueing discipline all jobs receive simultaneous service from the processor at a rate which is inversely proportional to the number of jobs at the processor [76]. PS is an approximation to round-robin scheduling.

The MVA solution technique is a basis for developing more efficient approximate solution techniques, such as Bard-Schweitzer [68] and Linearizer [45]. Although approximate analytic techniques do not produce mathematically exact predictions, "the loss in accuracy from moving to techniques that only approximate the exact solution of a model is usually much lower than the error inherent in representing a system using an analytical model" [76]. The approximate analytic techniques have several advantages over exact MVA, such as: lower storage requirements, lower computational cost [69], and additional modeling capabilities (e.g., memory contention, priority queues, lock contention, parallel execution and dynamic load sharing [76]). Other approximate analytic techniques which are discussed in the literature are: Flow Equivalent Service Center (FESC) [23], and the Method of Surrogate Delays [60]. This thesis uses a Layered Queueing Network Model because it generates performance estimates which include contention due to software servers.

2.2.3 Layered Queueing Network Model

A Layered Queueing Model (*LQM*) characterizes queueing at the hardware resources and at the concurrent computer programs (*tasks*) that provide services. The LQM is appropriate for modeling client-server architectures [73], where client tasks requests services from server tasks, because the LQM uses the architectural information when solving for the performance estimates. In an LQM, a client task requests service using a communication mechanism like a remote procedure call [11], where the client task is blocked until its server task replies. This task interaction is called a *rendezvous*.

Client tasks experience contention due to competition for service from server tasks. A server task becomes a bottleneck [43] when its utilization is greater than any of the hardware resources it uses [65]. Two LQM analytic solution techniques, based on approximate mean value analysis, are the Stochastic Rendezvous Networks [84] and the Method of Layers [39].

The structure of an LQM is hierarchical, and has the following components:

- *Task*: A concurrent program (e.g., a Unix process or ADA task) which exists in the system to be modelled. A task will make demands at resources and act as a client by *calling* other tasks for services; using an RPC or sending a request message. A task can act as: 1) a pure client by only requesting services from other tasks, 2) a pure server by only processing requests for service, or 3) as both a client and server, where it receives requests from clients and processes them by generating its own requests to other server tasks. A *reference task* is a concurrent program that acts as a pure client because it never gives service to other tasks. Any task can make use of resources.
- *Service entry*: A task's *service entry* characterizes the resource demands for satisfying a client's request for a particular service. A reference task has one and only one entry, which is never called by other tasks. A serving task can have multiple entries.

- **Phase:** The resource demands of a serving task's entry are grouped in at most two phases. Each phase characterizes the calls from its entry to the entries of other server tasks. A server's first phase begins when the request is accepted. A server's second phase characterizes its execution after a reply has been sent to the client (*post-rendezvous*). The server task must complete its second phase before another client's request is accepted.

Both client and server tasks make visits to devices from an entry's phase. An average service time is specified for a client's visit(s) to a resource. In general, the mean value for service time (per visit) at a device is assumed to be known.

Figure 1 shows the structural components of a LQM. The two tasks of Figure 1 are shown as parallelograms and their entries are shown as the framed labels within each task. Devices are shown as circles. Each arc represents a call to a task or visit to a device, where the framed number is the average number of calls/visits. The service time for each visit to a device is not shown in the figure. In Figure 1, the SERVER task has the two entries "S1" and "S2". The CLIENT's entry "C1" makes two calls to the SERVER's entry "S1" and three visits to the DEVICE. The SERVER's entries would also make calls to other entries and visits to devices (which are not shown).

Figure 1: LQM Components

In a LQM, a serving tasks can be one of several different types, including [41]:

- *Single server with multiple entries.*

- **Multi-server:** Multiple serving tasks share a common queue for retrieving customers that are served concurrently. The number of concurrent requests which can be serviced is a fixed size.
- **Producer-consumer server:** It is used to predict the waiting times of producers and consumers using a fixed size buffer.
- **Synchronization server:** Provides a synchronization point for two tasks. The synchronization server includes a service phase during which the synchronizing tasks receive service in unison. At the end of the service phase, the client tasks are released.

In this research only single server and the multi-server types are considered.

An LQM produces performance estimates that are not available from a queuing network. These LQM performance estimates involve the client task and serving task. These measures include: the residence time of a client task at a server task, the response time of a client task at a server task, and the total throughput at a server task, and its utilization.

The information needed to construct a LQM includes both the behavior and software structure of the system:

- **Device information:** The scheduling policy of each device. The supported scheduling policies are described in the literature [39].
- **Task information:** The task type (reference or non-reference), the list of entries belonging to this task, the resource on which the task is allocated, task priority (if it applies), and a task identifier.
- **Server type:** The type of server that represents the task, such as a multi-server, delay server, single entry server, producer-consumer, or synchronization server.
- **Entry and phase information:** This includes the total number of entries and the workload parameters associated with the phases of each entry. The phase information for each entry is: the mean phase service demand and the mean

number of calls from this entry to other entries per invocation. Each phase is assumed to have an exponentially distributed service times and a geometrically distributed number of calls.

This information includes more detail than is needed for other analytic performance models (e.g., QNM). A benefit of the additional detail is that the performance impact of queueing for software servers can be assessed. It also allows for the performance estimates to be related to the software system structure without reinterpretation. However, the additional details tax an analysts' information gathering and management ability. The automation of information management and model construction for a LQM would be a substantial gain.

2.3 Prototypes as Early Executable Designs

Prototyping is an approach used to increase the probability that the first production version of a software system will be satisfactory. It is a tool for decreasing the risk of design failures [6]. A brief categorization of prototyping methods is discussed which focuses on the use of the prototype. Several prototype environments are then introduced, with a comparison of the qualities which are important to this thesis. The reader is referred to [20] for a review of other classifications for prototyping.

The purpose of a prototype can be to validate a specification or design. *Specification prototyping* is used to determine whether user requirements are satisfied. *Design prototyping* involves a detailed exploration of potential implementations to determine its adequacy at a technical level.

Specification prototypes [6] can be used to refine and validate specifications through trial use and feedback. The end product is a prototype that serves as a verified specification. Specification prototype languages are suitable for specification and validation simultaneously [62]. Specification prototype languages have two forms:

- **Declarative:** Specify what the program must do by establishing some properties of the computations which must be respected [26]. Prototype execution is specified by equations which define the operational semantics for abstract data types.
- **Role based:** Specify what the program must do by defining and assigning operational roles to concurrent entities which interact. The operations that the system performs is the sum of those which the entities provide. The system specification can be modified by refining, redefining, or reassigning the operations of entities.

The specification semantics can be textual or graphically based (e.g., StateCharts [32], SDL [9]).

Design prototyping examines the adequacy of a proposed solution at a technical level [6]. The purpose of design prototyping is to: perform software-hardware trade-offs; model the environment to be designed; permit separation of structure and execution in models; support top-down design through multiple levels of refinement; support building block composition, physical realization, and the testing of its designs [82].

Both specification and design prototyping can provide information that is useful for constructing a performance model. A declarative specification prototype can identify the steps to satisfy an operation, provided the prototype environment can isolate each operations. The role based specification prototype includes software structure details that can be used when constructing a performance model. The design prototype has the necessary detail for a performance model. This thesis provides a way of making this information available for building a performance model. The next subsection examines prototype environments and discusses the elements which can be utilized for this purpose.

2.3.1 Several Prototype Environments

A *prototyping environment* is similar to a programming environment, incorporating tools for: design, coding, testing, and debugging. Five representative prototyping environments are:

- **PAISLey:** PAISLey is a specification prototyping tool which supports an “operational” approach [86] [87] to software specification. It “formulates a system to solve the problem and specify a system in terms of implementation-independent structures that generate the execution of the specified system.” The executable specification is composed of a set of asynchronous tasks, where each task has a state and goes through a never-ending sequence of discrete state changes. Tasks communicate using point-to-point, two-way, mutually synchronized inter-task communication. PAISLey is a functional language which is described in [89] and [88].
- **Statemate:** Statemate is a specification prototyping tool which is used for executable specification analysis, design, debugging, and documentation of large and complex computer systems. It is based on a visual formalism [31] that is an extension of finite-state machines [32] [33]. A specification is described from three views: hardware and software modules define the structure, operations are described in work units, and the execution formalism is an extended state machine [37] [38].
- **CAPA:** Computer Aided Prototyping System (CAPA) [54] [10] is used for the generation of rapid prototypes by a top-down decomposition model that combines dataflow (with real-time constraints) and control flow [55] [56]. Computer systems are described as networks of independent tasks that communicate through buffered data streams. The data streams carry exception conditions or values of arbitrary abstract data types. Tasks are modelled as data transformations or state machines. The Prototyping System Description Language [53] [52] is used to describe the

prototypes. An example of CAPA's use is provided in [51]. CAPA is a design prototyping tool which can also be used for specification prototyping.

- **MLog:** Mlog [29] combines: the packaging features of an object-oriented programming systems (OOPS), the computational model of Prolog, concurrency based on communicating sequential tasks, and the rendezvous for inter-task communications [28]. MLog is used for specification prototyping of real-time systems. Mlog prototypes are composed of *active objects* and *passive objects*. Passive objects are provided for abstract data-type definition and use. An active object executes in a "run-until-blocked" fashion. In general, active objects communicate using a message-passing rendezvous: the sending object waits for a reply, the receiving object will invoke a method when it accepts the message, and the result of the method is sent as a reply to the sending object. A rendezvous occurs if and only if both sender and target agree to communicate. Asynchronous messages can be sent between active objects. Nested acceptance is allowed, but requires that the most recent rendezvous be terminated first. MLog provides the object-oriented features of *single inheritance* and *polymorphism*. MLog is discussed further in section 4.3.

- **ObjecTime:** ObjecTime is a commercial prototyping tool which supports the Real-Time Object-Oriented Modeling (*ROOM*) software development approach [72]. It is tailored for real-time systems, and provides an iterative and incremental process which refines a prototype into a final implementation. It has the ability to capture, validate, and sustain the architecture of software systems using executable models. "The primary component of a ROOM model is an actor class, which represents a set of objects, using the broadest definition of an object as a logical machine of arbitrary scope. An actor interface is defined in terms of protocols, which represent sets of related messages. Actors have a behavior, which is described by a state machine called a ROOMchart and by extended state variables defined in terms of data classes" [72].

2.3.2 Prototyping in this Thesis

For the purposes of this research, a prototyping environment is a very-high level language which models the logical aspects of a system, including such things as division of the work among concurrent entities (tasks) and communication between tasks at a high level (e.g., [6], [14]). This definition does not preclude the use of any of the prototyping environments previously discussed.

Two characteristics of prototyping environments which are useful for this research are the type of execution clock and the tracing facilities. The clock is used to temporally order the execution of each concurrent entity. This clock may have no relation to the execution time in the final (implementation) system. Each prototyping environment provides the ability to record traces of events that occur during prototype execution to a file. A prototype clock's time base can be:

- *Logical Time:* Each executed activity is considered to consume one logical clock tick, which has no correlation with the final system. This allows simulation of concurrent sub-operations, but does not order events as they would occur in the final system. The prototype trace is simply the list of statements executed by the prototype. Statemate, ObjecTime and MLog have logical clocks.
- *Quantized Time:* Concurrent statement execution is temporally ordered by a clock which approximates real-time, but with a finite resolution. A prototype trace is a partially ordered set of events, where the uncertainty in event ordering depends on the resolution of the clock. PAISLey and CAPA have quantized clocks.
- *Simulated Time:* Real execution time is emulated between sub-operations, and concurrent events are ordered as they would occur in the implementation. The prototype trace is the ordered set of events that occur. Statemate and ObjecTime have a mode that simulates prototype execution in this fashion.

A prototyping environment's tracing facilities are usually used for testing and debugging. It is shown in this thesis how the traces can be adapted to provide the structural and

behavioral information for constructing analytic performance models. Any of these time bases can be used for this research.

2.3.3 Using Prototypes as Simulation Models

A prototype can be used to construct a performance model in two fashions: 1) the prototype can itself be used as a simulation model, or 2) an analytic model can be developed based on the prototype. Leung [50] compared analytic and simulation solution models and found:

“The principal advantage of a simulation model over an analytic model is that the former allows the incorporation of many more details, which are often difficult to explicitly include in an analytic model. It’s generally true that any numerical results which can be obtained by analytic means can always be produced by simulation but not vice versa. Simulation models, however, are both expensive in development cost - programming, debugging, and validation - and in running cost. Nor does a simulation model permit much sensitivity analysis and extrapolation from a given set of experimental conditions, since a change in a single parameter of the model usually means re-running the entire simulation. It is fair to say that simulation models are primarily empirical in character, and their results admit only limited interpretation. Analytic models, on the other hand, though they cannot perhaps accommodate as many details as simulation ones, offer a relatively inexpensive alternative for evaluating system performance as well as permitting conclusions to be drawn respecting the causal relationships among system parameters.”

The important point for this research is that analytic models provide faster results than simulation models, so they offer a greater ability to explore relationships between system parameters. This capability is important during the initial stages of development when

many performance experiments are conducted for evaluating design alternatives. For these reasons, the research focuses on analytic performance models.

2.4 Measurement and Tracing

A *trace* is an ordered record of events which have occurred within a system. The event ordering is usually time based and may be fully defined (e.g., global time-stamped), or partially-ordered (e.g., logical clock). As discussed in section 2.3, the model building technique presented in this thesis can use either of these.

Traces are used for many purposes, including:

- *Formal methods:* Trace theory is a mathematical way of representing and reasoning about concurrent tasks [57]. It is a mechanism which may be used to advantage when designing programs for concurrent computations by systematically deriving a concurrent set of programs from a specification [63].
- *Specification and verification:* Traces can be used early in software development for verification of a specification. Hoare [17] has suggested this and has developed an algebra for this purpose. Kourie [47] produced an example of this approach, using a Prolog prototype to produce CSP traces for verification of a specification.
- *Debugging:* Debugging techniques can make use of tracing. For example, Ada task traces have been advocated as a debugging tool [16] for dynamic and static analysis. Bates has described using abstract executional analysis (via high-level, filtered traces) to support debugging of distributed systems [8].
- *Performance prediction and analysis:* To analyze the performance characteristics of memory cache architectures, traces of data-access are used to drive a simulation [79]. Logs in actual systems are a form of traces which record execution and can be used in performance analysis [42], where the events record the use of resources.

Trace generation has several characteristics, of which the most important are: the definition of the events and the information to record, instrumentation for recording events, and causal ordering of events. By examining recent research, some guiding principles for performance oriented traces can be obtained.

The events to be recorded in a trace should be a complete set. Larus [48] [49] has developed a technique to reduce the amount of recorded events from tracing a single program, for use in a trace driven simulation. His technique examines and instruments a program to identify and record *significant events* (e.g., basic block boundaries) during the program's execution. The significant events are then used as markers to regenerate a full trace by post-execution analysis of the trace and the original program. This research defines a set of events for constructing a performance model.

Programmer effort for tracing should be minimized. Ogle, Schwan, and Snodgrass [59] built a monitoring system to collect, analyze, and report information for distributed applications. The trace instrumentation probes were generated automatically for each application and inserted by hand into the application code. Data collection from each probe could be enabled and disabled. For performance oriented traces, effort should be minimized for program instrumentation and data collection control.

The causal dependence (or independence) of events for a concurrent, distributed system must be recorded to describe its behavior [57]. For example, Helmbold, McDowell, and Wang [35] used a set of traces for detecting critical races for programs with semaphore style synchronization. The data recorded for each synchronization event was: the effected semaphore, the name of the task that performed the operation, and a vector of timestamps. The time vector is used to impose a partial order on recorded events. The set of event traces was used to construct a *causal trace*, where a "causal trace is a trace that indicates, for each blocking event (e.g., semaphore Wait) the particular enabling event (e.g., semaphore Signal) that caused the event to become unblocked. A causal trace contains arcs from each enabling event to the blocking event it unblocks, in addition to the arcs from each event to

the next event.” This research extends the notion of a causal trace for the purpose of performance model construction.

2.5 Background Summary

This chapter has reviewed the categories which form the background of this research, namely: software performance engineering approaches, computer system performance models, prototyping, and tracing techniques. The next chapter will provide an overview of the technique.

Chapter 3.0 Overview of the Model Building Technique

“If we are to be rational designers, we must begin knowing what we must do to succeed.” D.L. Parnas [21]

This chapter provides an overview of the model construction technique which is described in detail in the following chapters. Figure 2 is an outline of the flow of information for the technique which can begin once the prototype is operational. It is assumed that performance experiments have been identified and performance objectives are formulated for each performance experiment. The analyst selects only those performance experiments judged to be most significant, such as heavily used requests or requests of special importance. A requirements document will usually provide this information. Unlike functional testing, not all behaviors need to be covered for performance. A performance experiment shall be referred to as a scenario.

The technique has several steps, of which the first six can be automated:

- Record angio traces for selected performance scenarios (steps #1 and #2).
- Reduce the traces to workthreads and workthread classes (steps #3 and #4).
- Construct and evaluate a performance model to produce performance estimates (steps #5 and #6).
- Compare the performance estimates with performance objectives and improve the design if necessary (step #7).

A scenario is defined by its *scenario name* and *scenario options*. The scenario name is provided by the analyst to differentiate major types of scenarios. Scenario options reflect

Figure 2: Model Generation Technique

variations in the request data or in the prototype data state. Each scenario name and set of options are associated with a *trace name* that identifies the recorded execution. The analyst uses the trace name to associate the scenario with the performance model predictions. Instrumentation is added to the prototype so that the trace name is recorded in the angio trace.

The *statements* of the prototype language are associated with estimates of resource demands. A prototype does not need to emulate resources (e.g. disks or CPUs) but only the sequence of statements executed by tasks. A task can be an object, a cyclical operating system process, a data flow transform, or a state machine. Partially implemented prototypes can be used in some environments. Examples of such prototyping environments are: PAISley [88], PSDL [10], ObjecTime [58], SDL [9], Statemate [33], and MLog [25]. For this research the MLog prototyping environment was enhanced to produce angio traces.

When the prototype is executed it records *angio events* that are gathered and filtered into *angio traces*. The phrase “angio trace” is derived, by analogy, from an angiogram which is a visualization of an individual’s blood flow produced by injecting a radio-opaque dye into the blood stream and taking an X ray of the dye dispersion. Similarly, an angio trace is a description of a prototype’s execution, where a unique identifier (*dye*) is injected in the prototype for each trace name. The dye follows the control flow in the servicing of a scenario and each statement the prototype executes is logged. Each item that is logged is an angio event that provides causal and resource demand information. For the rest of the document, the term *trace* will refer to an angio trace.

Angio tracing minimizes instrumentation errors and programmer effort by providing generic instrumentation probes. The prototype environment is modified so that once a dye is injected, the executed statements are automatically recorded and associated with the dye. Provided the prototype environment’s system functions have been instrumented, the analyst only has to inject a dye to produce a trace. The analyst is freed from explicitly recording the task identifier or event type information because the prototype environment provides this information.

A workthread gives the sequence of resource demands to service the scenario. It is built from the trace and a set of *resource functions* (which give the costs of different statements in the prototype). The workthread also includes information needed to deduce interactions between tasks.

Analyzing the workthread produces a *workthread class* that characterizes the average resource selection and demand parameters for a scenario. It also has structural information about the software that is useful for building performance models. In this research, the software components of a workthread class are based on the LQM structure components (section 2.3.3): *task*, *service entry*, and *service entry phase*. It should be emphasized that a service entry is different from a prototype task’s entry. A service entry characterizes the resource demands for a service that a task provides and is a modelling

abstraction. By contrast, the task entries in a prototype are structural; they relate to the type of inter-task message being received.

After a performance evaluation technique is selected, the workthread classes are combined with *environment information* to generate a performance model. The environment information contains system configuration details that can be considered separately from the software execution behavior. The environment information includes resource characteristics (e.g. scheduling discipline), task information (e.g. task population), the workload intensity of scenarios, and resource allocation. The environment information is provided by the analyst, which allows different models to be constructed. This can be useful for studying the system's performance over a wide range of circumstances.

In principle, any suitable type of model that uses average parameter values can be generated, such as a workload estimate [23], queueing network model [23], and a Layered Queueing Network model [84] [41]. Table 1 on page 12, considered three types of models listed in order of: increasing complexity, information requirements, and solution cost. A simple model is appropriate when the prototype itself is very simplified. As the prototype's level of detail increases, the design decisions being explored will be more intricate, so more complex models are justified. This research focuses on constructing a Layered Queueing Model.

The technique is based on several assumptions, which are:

- A scenario has a typical execution behavior that can be recorded in an angio trace.
- A scenario's execution behavior, resource selection, and resource demand are independent of the execution or load imposed by other scenarios.
- Executed statements correspond closely enough to execution in the target implementation to allow the implementation resource demands to be assigned to prototype statements (via the resource function).

In summary, the prototype's software components and behavior are recorded in the trace and used to find the structure and parameters of the model. The statements in the trace are associated with model resource demand parameters. The software component information recorded in the trace is used to develop the model's structure and to map the resource demands onto this structure.

It should be pointed out that the terms "trace name", "dye", "angio trace", "workthread", and "workthread class" correspond to the same scenario at different stages in the analysis. For example, the dye is used during prototype execution to record the angio trace.

Chapter 4.0 Angio Trace Generation

“The king is familiar with these things, and I can speak freely to him. I am convinced that none of this has escaped his notice, because it was not done in a corner.” Paul, Acts 26:2

This chapter describes steps #1 and #2 of Figure 2 on page 29, in which an angio trace is recorded. The first subsection describes angio traces and the event information. The subsequent subsection (section 4.2) discusses the “event type” field which provides the information for model building. An implementation of angio tracing is presented in section 4.3. A visual notation for angio traces and workthreads is presented in section 4.2.2.

4.1 Angio Traces

Angio tracing is tailored for recording information that is used to generate a performance model. It identifies the concurrent execution of tasks associated with a scenario for the performance model. The three characteristics of tracing that will be used to present angio traces are: the trace event definition, trace event instrumentation, and the preservation of causal dependency between events.

Each angio event records structural and behavioral information. Each event has the following fields:

- ***An identifier for the dye that was active when the event was recorded.***
- ***An identifier for the prototyping language statement that was executed.*** This information is used as a search key for selecting the resource function for the event.
- ***An identifier for the task that executed the statement.*** The task identifier is used to categorize tasks with the same behavior. Those tasks with the same task

identifier are modelled identically in the performance model. So, it is not the same as an operating system's task identifier (e.g. Unix task ID). In this thesis we assume that tasks with the same task identifier do not communicate with one another.

- *Event type information for constructing the performance model.* This is the topic of the next subsection.
- *Event parameters for resource demand calculations or trace management.*

Experience has shown that object-oriented prototyping environments should also record the executed statement's class, and the class of the executing task.

To build a performance model, it is necessary that statements be associated with the scenario that caused them. This causal relationship between the events in the angio trace and the scenario are maintained as follows:

- *Run-time dye propagation.* The dye crosses task boundaries during task interactions and persists across task context switches during execution. For example, inter-task messaging is modified to pass a sending task's dye to the receiving task. The receiving task then associates any statements it executes with the dye that was received with the message.
- *Post-execution analysis.* A naming convention is used for inter-related dyes to support post-execution causal analysis. Concurrent threads of control which cooperate on the same scenario are related by trace name. The naming convention is discussed further in section 4.2.

If it is necessary for messages to be queued internally by a task, then the queueing operations need to retain the dye information with the message. Otherwise it is possible that the event recording and trace naming become uncoupled, so further analysis is not possible.

The event type, and event parameter fields, provide information for model construction, which is the topic of the next section. Unless otherwise indicated, an angio event shall be referred to as an “event”.

4.2 Angio Event Types

An *angio event type* records the relationships between recorded events, the software system’s structure, and the performance model. The event types used in this research are summarized in Table 2. In section 4.3, an instrumentation probe is provided for each event type. To reduce instrumentation effort the default event type is one in which resources are consumed, as specified by a resource function. Event types for recording looping or choice selection are not needed by this approach. The event types in Table 2 are discussed in the following subsections.

Event Types	Description
wBegin	Begin recording of a (sub-)trace with a specified dye.
wEnd	Ends a (sub-)trace.
wSpawn	Records the point at which a parent (sub-)trace has spawned a child subtrace.
wDelay	The currently executing task is delayed for a specified time period.
resUse	Records a resource, and a resource demand, in the trace.
taskEnter	Records a task identifier.
taskPhase	Indicates the phase of the currently executing task should be increased.
taskService	Provides a name for the service which the task is providing, which is used to name the workthread class service entry.
wReport	Outputs text in the trace for documentation purposes.

Table 2: Angio Trace Event Type Summary

4.2.1 Beginning and Ending Traces

The smallest trace is composed of the event types for starting (**wBegin**) and stopping (**wEnd**) a trace. These event types do not correspond to resource demands or actual statement execution, but are used to demarcate the beginning and end of a valid trace. The **wBegin** event type initiates the recording of a (sub-)trace by injecting the dye during prototype execution.

The dye is formed from two parts: the *trace name* and an *occurrence number*. Initially, the trace name identifies the scenario directly. However, as discussed in the next subsection, when a subtrace is spawned its trace name is a derivative of the scenario name. The occurrence number is automatically provided by the run-time to distinguish different instances of the same dye (scenario).

The **wEnd** event type ends a (sub-)trace. It is operationally equivalent to stopping the recording of events *for the current dye in the executing task*. To distinguish traces which successfully complete and traces which prematurely end, all successfully completed traces are terminated with a **wEnd**. If a trace does not complete with a **wEnd** it is assumed to be incomplete and discarded.

4.2.2 Concurrent Execution

Concurrent threads of execution that are associated with the same scenario must be distinguished, otherwise event interleaving will prevent further trace analysis. The event type **wSpawn** is used for this purpose: to causally associate concurrent sequences of events (subtraces) with other subtraces of the same scenario. When a concurrent thread of execution is about to begin:

- A child dye is spawned with a unique dye name. It is used to record the execution of the new concurrent thread of control. The child dye execution is recorded as a subtrace.
- A **wSpawn** event type is recorded in the parent trace, along with event parameters to link it to the spawned subtrace.

It is assumed that the prototype run-time environment, and selected prototype statements which can enable two or more tasks to execute concurrently, are instrumented to automatically generate the **wSpawn** event type.

Figure 3 is an example of concurrent execution which spawns a child dye. It illustrates the way traces, and the spawning of subtraces, are shown. The illustrations

borrow from the graphical notation of Timethreads [15]. Dye propagation during prototype execution is shown as a line which connects events recorded by the dye. A dye and (sub)trace begin with the diamond symbol $\blacklozenge$ (**wBegin**). They end at the slash (**wEnd**). The spawning of a child dye (**wSpawn**) is shown as the beginning of the child dye touching the parent. For example, Figure 3 shows the child dye “AR.1” being spawned where the beginning of the child dye touches the parent dye (“AR”). Tasks are shown as parallelograms.

In the example of Figure 3, a child dye is spawned when an asynchronous message is sent from the Task AP and received by the Task CM. The potential for concurrent execution exists after an asynchronous message send because the sending task continues execution after sending the message, and the receiving task can execute once the message is received. It is not known until after further prototype execution whether the potential concurrency is realized. The prototype run-time statement to send an asynchronous message (“**Async**”) is instrumented to spawn a subtrace (e.g. “AR.1”).

Figure 3: wSpawn Example

Several statements can cause a **wSpawn** event type, aside from sending a message asynchronously. A similar event would be recorded if a fork by Task AP created the Task CM. A third, rather non-intuitive example of a **wSpawn** event type occurs when a serving task continues execution after replying to a client task; a subtrace would be spawned in the server task to record its post-rendezvous execution.

A naming convention, for the spawned dye, is used to keep track of parent/child dye relationships. A unique spawned dye's name is constructed using the following rules:

- 1) The child dye's trace name is the same as the parent dye's trace name, except it has a subscript number appended to it. The subscript is derived from the parent dye's *spawn count*, which is the number of child dyes that have been spawned during the parent dye's execution. The spawn count is a value maintained by the prototype run-time during execution.
- 2) The child dye inherits the occurrence number of its parent dye.
- 3) The new dye is injected to record the execution of the new concurrent thread of control. The **wBegin** event type of the new dye is recorded as starting in the task which initiates the spawning. For example, in Figure 3, the child dye "AR.1" begins in the task "AP" and not in Task "CM". However, the next event of the child dye is recorded as executing in Task "CM". This convention is important because it allows the recorded subtraces to be hierarchically related.
- 4) The parent dye records the event type **wspawn**, including the child dye's trace name as an event parameter.

This naming convention provides for an unlimited number of levels of concurrency.

Figure 4 illustrates the manner in which the spawned dyes and their corresponding subtraces are named. The scenario begins with the dye "W", which spawns two child dye's called "W.1" and "W.2". The child dye "W.2" also spawns a child dye that is called "W.2.1". In each case, the value that is appended to a dye's trace name is its spawn count.

An implementation of the naming convention for multicast or broadcast communication would assign a unique dye to each receiving task.

4.2.3 Task, Phase, and Service Identification

Event types are also used to record software component information for the workthread class components. These components include the task identifier, service entry

Figure 4: Naming Convention Example

identifier, and phase identifier of an entry. The event types which are associated with each of these structural components are:

- **taskEnter**: Indicates a task has started/resumed execution and records the task identifier for trace analysis.
- **taskService**: Associates subsequent statements, executed by the current task, with a service entry name which is supplied by the analyst. The event type **taskService** and entry name are recorded in the trace.
- **taskPhase**: A change in the phase of the current task's service entry has occurred. For example, when a server task replies to a client to end a rendezvous the trace records a **taskPhase** event type for the server.

To reduce the analyst's effort, this information can be generated by the prototype system at run-time (e.g. task identifier) or found during post-analysis of the traces (e.g. task phase information). Using the scenario name as the default entry ID reduces instrumentation effort but it may limit the types of performance modeling experiments that could be conducted.

4.2.4 Delay

The event type **wDelay** records the passing of a specified amount of time. Typically, prototyping environments have statements to delineate the passage of time (e.g. "sleep()"), which can be used to model external delays for resources or users. Such

statements would be instrumented to emit the **wDelay** event type, with an event parameter that is the duration of the delay.

4.2.5 Resource Management

Should the analyst wish to record resource use directly to the trace and not use resource functions, the event type **resUse** is provided. It records a resource selection with demand(s) that are determined during prototype execution. In this manner, resource selection can be delayed until prototype execution (e.g. selecting the link for a data packet transmission). The resource demand of an event can also vary based on the state of the prototype and type of request (e.g. the size of a data packet to transmit).

4.2.6 Documentation

Although not necessary for model generation, an event type has been added which allows a text string to be recorded in the trace. The event type **wReport** will record user specified information in a trace. It is useful for debugging and testing purposes.

4.2.7 Event Type Summary

Table 3 summarizes the event types, and their parameters, that are used in constructing the performance model. Having presented the event types, some observations can be made:

- *One statement may produce several different event types which are recorded with the same event.* For example, when a server task replies to a client, the statement can be instrumented to spawn a new subtrace (i.e., **wSpawn**) and increase the phase of the server task (i.e., **taskPhase**). Analysis of the trace assumes these event types occur at the same point in time.
- *A statement may have several different trace interpretations.* This is resolved by the analyst providing additional information when constructing the prototype. For example, receiving a message can be considered to be: 1) the beginning of an

execution cycle for a server task, or 2) a synchronization point for two or more tasks. To differentiate the two cases, the analyst provides more information to describe the intent of the prototype statement, such as a Boolean parameter to determine what is recorded in the trace.

- *Event types can be assigned at post-execution analysis.* A statement identifier may provide enough information so that post-execution trace analysis can derive the event type (e.g. **wDelay**). In these instances the event type need not be recorded and could be stored with the resource function.

An implementation of angio tracing is now considered.

Event Type	Event Type Parameters
wBegin	None
wSpawn	Name and occurrence of the (sub-)trace which is spawned.
wEnd	None
wDelay	Duration that the task is delayed.
taskEnter	Identifier for the task that has become active.
taskPhase	None.
taskService	Name of the service the task is providing.
resUse	Resource ID used, as well as the resource demand.
wReport	Text line for output in the angio trace for documentation purposes.

Table 3: Angio Event Type Parameters

4.3 MLog Angio Trace Instrumentation

This section describes the implementation of angio probe instrumentation in the MLog prototyping environment. This involved modifications to MLog's tracing facilities and the instrumentation of it's system methods. This angio instrumentation has three layers: *angio primitive methods*, *angio probes*, and *statement instrumentation*. The primitive methods provide the ability to construct angio probes for each event type. The angio probes for the event types **wBegin** and **wSpawn** will be used as examples. The primitive methods and angio probes are then used to instrument the MLog system methods to record event type information. Synchronous and asynchronous task-interactions are presented as

examples of statement instrumentation. Other important implementation issues conclude this subsection.

4.3.1 MLog Angio Instrumentation

The MLog prototyping environment's tracing facilities have been modified to record the following information for each executed statement:¹

- *The dye's trace name and occurrence number.*
- *The class and "task ID" of the active object which generated the event.* It has been observed [79] that identifiers of terminated operating system processes can be reused in traces. In this research, this means a terminated active object's identifier could be reused which could confuse the trace analysis. However, by combining the object's class and "task ID" to form the task identifier the difficulty is eliminated.
- *The currently executing method, its arity, and class.* If two methods in the same class have the same name, MLog will select the method to use based on the number of arguments to the method, or its *arity* [22]. By including the class and arity of the currently executing method the object oriented language features of inheritance and polymorphism in MLog are accommodated, illustrating that the tracing methodology can be applied to object-oriented environments.

The state information of each active object was augmented to include the trace name, occurrence number, and spawn count of its current dye. The angio tracing was disabled by default, and enabled by the command "**angio.**"

New primitive methods were added to MLog to build higher level probes. These primitive methods manipulate the variables for each dye's trace name, occurrence number, and spawn count. They are:

1. This work was done in collaboration with Gerald Karam, the developer of MLog.

- **injectDye(traceName, OccNumber)** A dye with the trace name *traceName*, and the occurrence number *OccNumber*, is associated with subsequent statement execution by the task that executes this primitive method. Another version of this method is available for which the run-time automatically assigns an occurrence number.
- **absorb** Stops the dye from progressing further.
- **currentDye(CurrentName, CurrentOccNumber)** Returns the current trace name (*CurrentName*) and occurrence number (*CurrentOccNumber*).
- **outputAngio(text)** Records *text* in the trace with the current trace name and occurrence number.
- **spawnCountGet(CurrCount)** Assigns the variable *CurrCount* to the spawn count value of the current dye.
- **spawnCountSet(NewCount)** Assigns the spawn count of the current dye to the value of *NewCount*.

These methods are used to construct the higher-level interface of the angio probes.

There are three *tracing contexts* which an angio probe must deal with: 1) a dye is not active; 2) a current dye exists for which events are being recorded; and 3) a dye is being injected, which will replace an existing dye. The angio probe instrumentation hides the dye manipulation and is responsible for recording the event type information. The MLog implementations of a **wBegin** probe and a **wSpawn** probe are shown in Figure 6 and Figure 7. To construct the probes, only string manipulation and the angio primitive methods are used. Before examining these methods an explanation of the MLog syntax is needed.

For this research, an MLog method can be considered to be a function where: the left-hand side of the “:-” is the function declaration, the right hand side is the function body, and the statement separator is a comma. Other syntactic details are that the function is terminated by a period and a “%” indicates the rest of the line is a comment. Figure 5 is an

example MLog method called "**AMethod**" which has the parameter "**AParm**". Each method in MLog sends a message to an object to perform a requested operation. Messages are addressed by prepending the receiving object's identifier to the method, separated by a dash ("-"). For example, the statement "**Line-draw(5,100)**" sends the **Line** object the message to execute the method **draw(5,100)**. If no receiving object is specified, the message is sent to the currently active object. If the currently active object does not know how to satisfy the message, it will pass the message to its parent object in the inheritance hierarchy (called a *super-class*) for resolution. An object can send a message directly to its super-class using the special destination "*super*". For example, the statement "**super-doSomething**" sends the super-class of the current object the message to execute the method "**doSomething**". As with most 4th generation languages, identifiers (e.g. variables) may be used as needed. Variables start with a capital letter.

```

1  AMethod(AParm):-                % Method declaration
2    Line-draw(5,100),             % Message to the line object to draw at (5,100)
3    super-doSomething(AParm),     % Message to the super-class to doSomething
4    .                             % Method ends

```

Figure 5: Example MLog Method

The **wBegin** angio probe (Figure 6) injects a dye to begin recording a trace with the specified trace name (**ReqName**). The probe is designed to handle the three trace contexts mentioned above. If a current dye exists, a **wEnd** probe finishes the current dye before the new dye is injected (line 2). If this is not done, then the current dye's trace will be rejected (section 4.2.1). The **wEnd** probe also records the reason why the trace is ending ("**wBegin**"), which has proven useful when debugging the implementation. After the **wEnd** probe is executed no dye is active, so further executed statements will not be recorded until a new dye is injected. The new dye is injected in line 3 with the new trace name. An occurrence number for the new dye is assigned by the prototype run-time system. The last step in this probe is to record the event of beginning the new trace (line 4). The period of line 5 ends the **wBegin** method.

```

1  wBegin(ReqName):-
2      wEnd("wBegin"),           % End current dye if it exists
3      injectDye(ReqName),       % Begin the new dye
4      outputAngio("wBegin"),    % Record the beginning of the new trace
5      .

```

Figure 6: wBegin Angio Probe

The **wSpawn** probe of Figure 7 produces the **wSpawn** event type (section 4.2.2), including the spawned dye and other information. It does not take any parameters because the current dye's trace name is used to construct the spawned dye's trace name. The **wReport** statements are used to illustrate the changes in trace context (i.e. which dye to record events for). For example, line 2 of the **wSpawn** probe (Figure 7) starts in the parent dye's context but ends in the child dye's context in line 12. The probe begins by storing the parent dye's trace name and occurrence number in the variables **DyeName** and **OccNumber** (line 3). The method **wSpawnCountGetInc** retrieves the parent dye's spawn count, stores it in variable **NewCount**, and then increments the parent dye's spawn count (line 4). The spawned dye's trace name is created by concatenating the parent dye's trace name and spawn count together (line 5, line 6, line 7). The parent trace records the event type for spawning the dye in line 8. The trace context changes into the spawned dye by: ending the current dye (line 9), injecting the spawned dye with the occurrence number of the parent dye (line 10), and recording the beginning of the spawned trace (line 11).

```

1  wSpawn:-
2      wReport("Parent trace event"),           % Event recorded in parent trace
3      currentDye(DyeName,OccNumber),           % Get the current dye name
4      wSpawnCountGetInc(NewCount),             % Get parent dye's spawn count and increment
5      convert(NewCount,"string",CountString), % Convert the spawn number to a string
6      concatenate(DyeName,CountString,NewDye), % Add spawn count to the trace name
7      concatenate("wSpawn",NewDye,Str2),       % Construct the angio event record
8      outputAngio(Str2),                       % Record the angio event record
9      absorb,                                  % Stop the current dye
10     injectDye(NewDye,OccNumber),              % Inject the child dye
11     outputAngio("wBegin"),                    % Record the begin event of the child dye
12     wReport("Child trace event"),             % Event recorded in child trace
13     .                                         % End of the method

```

Figure 7: wSpawn Angio Probe

Prototype statements are instrumented using the angio probes and primitive methods to transparently record trace events. This transparency is accomplished by using the object-oriented features of inheritance and polymorphism to overload the normal MLog statements with those that have been instrumented for angio tracing. The same result can be achieved in a non-object oriented language by providing system libraries with, and without, the angio probe instrumentation. The examples of statement instrumentation which will be discussed are the **waitToAccept** and **performAsync** task-interaction.

Figure 8 shows the **waitToAccept** MLog system method with its angio instrumentation. This method causes an active object to stop executing until it receives a message. The **wEnd** event type must precede the reception of a message (line 3), otherwise the current trace will be disqualified because it did not end properly (similar to the **wBegin** probe). Once the current dye has ended, the normal system method **waitToAccept** is executed by the super-class (line 4) causing the executing object to wait until it receives a message. When a message is received, it will have a new dye with it. The run-time system will automatically replace the waiting object's dye with the dye which was received with the message.

```

1  waitToAccept(Msg):-
2    wReport("Last dye's event"),% Event recorded with current dye
3    wEnd("waitToAccept"),      % End the current dye
4    super-waitToAccept(Msg),    % Super-class executes the "real" waitToAccept method
5    wReport("Rx dye's event"), % Event recorded in dye from received message
6    .

```

Figure 8: Angio Instrumentation of the waitToAccept Method

Figure 9 gives the instrumentation of MLog's **performAsync**, which sends an asynchronous message to an active object. This method has to spawn a subtrace because it generates concurrent activity which is attributed to the same scenario. This is achieved by storing the parent dye (line 3), spawning the child dye (line 4), executing the normal system method for **performAsync** (line 5) which starts the child dye, and then restoring the parent dye (line 7, line 8).


```

1 performAsync(Msg):-
2   wReport("Sender dye event",, % Event recorded by parent dye
3   currentDye(DyeName,OccNumber), % Get the current dye name of the parent trace
4   wSpawn, % Spawn the child dye for the receiver
5   super-performAsync(Msg), % Super-class sends the message to the receiver
6   wReport("Receiver event"), % Event recorded in receiver's child subtrace
7   absorb, % Absorb the child dye
8   injectDye(DyeName,OccNumber), % Restore the parent dye
9   wReport("Sender dye event"), % Event recorded by parent dye
10  .

```

Figure 9: Angio Instrumentation of the performAsync Method

4.3.2 MLog Angio Trace Filtering

Before further analysis is done, the generated traces are filtered to convert the recorded events into a standard format and remove unnecessary information. and Redundant **wEnd** events and events with no resource functions are removed. Figure 10 is an example of a recorded MLog angio trace.

```

## termInit.0 1:: <#91{#terminal}> call: #91{#object}-stringLength(termInit.0 1,_2203)
## termInit.0 1:: <#91{#terminal}> call: #91{#object}-$stringLength(termInit.0 1,_2203)
## termInit.0 1:: <#91{#terminal}> exit: #91{#object}-$stringLength(termInit.0 1,12)
## termInit.0 1:: <#91{#terminal}> exit: #91{#object}-stringLength(termInit.0 1,12)
## termInit.0 1:: <#91{#terminal}> call: #91{#object}-1 > 12
## termInit.0 1:: <#91{#terminal}> call: #91{#object}-arith_compare(>,1,12)
## termInit.0 1:: <#91{#terminal}> fail: #91{#object}-arith_compare(>,1,12)
## termInit.0 1:: <#91{#terminal}> fail: #91{#object}-1 > 12
## termInit.0 1:: <#91{#terminal}> redo: #91{#object}-stringLength(termInit.0 1,_2203)

```

Figure 10: MLog Angio Trace Example

The MLog angio traces are filtered before the trace analysis occurs. Events without resource functions and which fall into four categories (Table 4) are removed:

- Methods that are specific to the prototyping language (e.g. "!").
- Methods that are used for operational purposes (e.g. arithmetic operators).
- Methods that provide input or output prototyping environment operations.
- Methods used for string manipulation.

These methods are pertinent to the prototyping environment but not to the system being modelled.

Operation Type	MLog Method	Description
Prolog execution	!	Cut to parent goal of executing method
Operational	$+X \backslash == +Y$	Term inequality
String	concatenate(+X,+Y,?Z)	Concatenate string Y to X and yield Z
System input and output.	nl(+Num)	Write Num end-of-line characters to the receiver's output stream

Table 4: MLog Methods that are Ignored

It is possible to generate superfluous **wEnd** event types in the same trace which are filtered out. These superfluous event types have two causes: **wEnd** event types are associated with prototype statements that cause a context switch during run-time (e.g. **waitToAccept** in Figure 8); and a single statement can produce several event types. For example, the **wBegin** angio probe of Figure 6 generates a **wEnd** to end a subtrace before another dye is processed. This poses two difficulties:

- ***wEnd** events can be recorded out of sequence with observed execution.* It has been found that interleaving can occur between the task which is ending the recording of a trace and another task recording events to a trace (e.g. **waitToAccept** method). This interleaving can be misinterpreted as an additional call from one task to another during the trace analysis. To prevent this, the **wEnd** events which occur in the middle of a trace are removed.
- ***Detecting successful completion of a trace.*** A trace is considered to successfully complete if the last event it records has a **wEnd** event type. However, if execution stops prematurely and the last recorded event is a **wEnd** event type, it may incorrectly be interpreted as a valid trace. The solution is to discard the traces which were executing when prototype execution halted.

4.3.3 Backtracking

MLog has a Prolog [77] inference engine which provides for *backtracking*. Backtracking solves a problem by systematically generating all possible solutions. It is analogous to “try something, and if it doesn’t work then try something else” [71]. When an MLog method executes, it is possible that a statement in its body does not complete successfully (i.e it fails). Since MLog methods try to complete successfully, backtracking occurs and an alternative method may be executed in place of the failed statement. Backtracking is a powerful feature of MLog which can be used in three fashions:

- *Backtracking can be intentional and explicit, where failure to continue is intended.* For example, it provides a way of constructing a “repeat” loop in MLog. An example of a repeat loop is shown in Figure 11, where the “**fail**” method (line 5) will result in a backtrack to the “**repeat**” method (line 2).
- *Backtracking allows a method to be used for assignment and selection.* For example, the “**Length**” method can be used to assign a length to a variable. It can also be used like an “if” statement, which evaluates to true when the length of a specified string matches another variable’s value.
- *Backtracking is used when a search for a solution occurs.* When a method’s parameter variables are not fully defined, a search is made for the parameter values which would successfully satisfy the method. This may involve many instances in which the statements of several methods fail and backtracking will occur.

```
1 MessageLoop: -  
2   repeat,  
3   waitToAccept (Message) ,  
4   processMessage (Message) ,  
5   fail.
```

Figure 11: Example of Loop Using Backtracking

Backtracking is a problem for trace analysis because all events which are executed by MLog are recorded to the trace, including the events from methods which do not succeed

and cause backtracking. When examining a trace it is not obvious which events are valid due to backtracking. Backtracking cannot be eliminated because the system methods will backtrack during execution. There are two approaches which solve this problem:

- The inference engine and tracing facilities of MLog could be modified to record goals that succeed, and the prototype statements that execute to satisfy these goals.
- The prototype statements are recorded using the **wReport** event type and the prototype statement information is ignored (i.e., succeed, fail). The prototype is also constructed so that only intentional failures (e.g. repeat loops) result in backtracking. This was the approach which was undertaken for the case study, where the special events were instrumented by hand. This instrumentation could have been automatically added by a file preprocessor.

4.4 Trace Generation Summary

Event types have been presented for generating performance models with synchronous interactions between tasks. The current angio tracing capabilities retains the causal relationships for asynchronous task interactions, even though the models do not currently make use of this. This section has described the composition and generation of angio traces. The MLog prototyping environment's implementation of angio tracing has proven the feasibility of angio tracing. It has also shown that prototyping environments which support backtracking complicate the trace analysis.

The angio trace preserves the causal information of events at a specification and analysis level. The analyst uses the trace name to specify scenarios. For analysis purposes, a trace is defined to be a set of subtraces which are causally ordered by event type information. This allows a (child) subtrace to be considered on its own, independent of the parent trace. An important consequence of this flexibility is that angio traces can be developed in a top-down, bottom-up, or iterative fashion.

The causal properties of the angio trace have two interesting attributes. Angio traces retain causal relationships when concurrent threads of control are spawned, which is an extension to the causal trace of Helmbold, McDowell, and Wang [35]. Also of interest is that the causal ordering of angio events is constructed from the dye and the sequential execution of tasks. This differs from other tracing techniques for ordering events in a distributed system, such as a global clock, timestamp vectors [35], or clock synchronization mechanisms [59].

Angio tracing is not only limited to prototyping environments or performance models. Angio traces can be constructed by hand or generated by actual systems in real environments. To provide for inter-operability between different angio trace sources and allow for further trace analysis, recorded events should follow the guidelines presented in section 4.2.

Chapter 5.0 Model Generation

"We have little experience in using higher-level [program] transformations, but the idea is so appealing, and the alternatives so inadequate, that it is certainly deserving of some enthusiasm." P. Zave [87]

This chapter describes steps #3 through #5 of Figure 2 on page 29, which translate the angio traces into a performance model. The first step is to convert an angio trace into a workthread. Then each workthread is analyzed to generate a workthread class. The last step generates a performance model by combining several workthread classes with environment information. Each of these steps is discussed in detail in the following subsections. An algorithm for converting a workthread into a workthread class is presented.

5.1 Angio Trace(s) to Workthread

A *workthread* characterizes the sequence of resource demands for a scenario. An angio trace is converted into a workthread by:

- i) Obtaining the resource demands of each angio event.
- ii) Summing certain sequences of resource demands together using the *service aggregation property*.
- iii) Associating demands and visits with the appropriate entry of a task.
- iv) Filtering out unnecessary event type information.

Each event in a workthread has the fields shown in Figure 12.

Resource demand estimation uses a *resource function*. A resource function estimates the *service demand* as a service time, or in units that can be converted into a

T_n	The task associated with event n
E_n	The task entry associated with event n .
Y_n	The event type of event n
J_n	The set of resources event n uses.
V_{nk}	The number of visits made to resource k due to event n
D_{nk}	The total service demand required from resource k due to the event n

Figure 12: Workthread Event Components

service time. It is identified using a table lookup, with the statement ID being the search key. The analyst has three approaches to using resource functions, which are:

- i) A *complexity measure*. Event parameters appear in the trace and may be used as arguments for a complexity measure that estimates the resource demands. For example, the data set size for a search algorithm can be used to estimate the cost of the search. The complexity measure is similar to the complexity function of Vetland [81]. This approach can be taken if a complexity measure is known.
- ii) A *distribution function*. The mean value of the distribution is used as the resource demand. No event parameters need to be recorded, except those for resource selection.
- iii) *Demand recorded at run-time*. The resource demand is explicitly indicated in the trace using the event type **resUse**. No resource function is needed. This approach can be used when the other two approaches are not possible.

Since prototype statements are the smallest recorded event, they are the smallest granule for a resource function that is considered. The coupling between the prototype, resource functions, and traces is different for each approach, with the first case having the least coupling. The preferred choice is to minimize coupling so that information binding is deferred. This permits the greatest flexibility in the analytic modelling stage.

The *service aggregation property* is used to sum resource demands for a better characterization of serial resource use. Initially it is assumed that the resource demand of a resource function constitutes at least one visit to the resource. Then the number of visits is reduced using the service aggregation property. When a resource with the service aggregation property is accessed in sequence by the same task, the consecutive visits are reduced to a single visit with the total service time. The analyst identifies resources with this property in the environment information. The service aggregation can also be applied during the workload conversion into a workload class (e.g., the algorithm of section 5.4).

Each workload event is associated with a task and service entry. This information can be recorded in the log implicitly (as is done in the MLog prototyping environment). The **taskService** and **taskEnter** event types (section 4.2.3) can explicitly record the information. By default, the entry name is the scenario name. After this step, the event types **taskEnter** and **taskService** can be discarded.

Execution overhead that is not part of a resource function can be incorporated at this step in the trace analysis. For example, a context switch can be assumed to occur when the executing task changes in a trace. To incorporate the costs of the context switch overhead, appropriate resource demand are inserted in the trace whenever the executing task changes.

The detailed structure of a workload can be exploited by a performance model which utilizes the sequence in which resource demands are made (e.g., detailed trace simulation). However, the performance evaluation techniques that are considered here use average values for resource demand and service requests. The information content of the workload is used to determine these values and is discussed next.

5.2 Workload to Workload Class

The *workload class* characterizes the average resource demands and task interactions of a scenario. It is an abstraction of the software system execution that provides

the structure and parameters used in the a performance model. As discussed in the next section, a performance model is generated by combining several workthread classes together, along with other model configuration information. The discussion will now focus on performance modeling details.

The workthread class has the parameters of Figure 13. It maintains the blocking relationships between tasks to reflect the impact on performance. The blocking relationships are structured into task entries and phases. This is a workload characterization principle that is in addition to those provided by Ferrari in [24].

I_{tepk}	The aggregate number of visits entry e of task t , in phase p , makes to resource k
D_{tepk}	The aggregate demand entry e of task t , in phase p , makes on resource k
$totalCalls[w, t, e]$	The total number of times task t 's entry e will be called for the workthread w .
K_{te}	Set of resources used by entry e of task t .
T_w	Set of all tasks in scenario w .
$C_{t'e'p'te}$	The average number of calls entry e' of task t' , in phase p' , makes to entry e of task t
V_{tepk}	The average number of visits entry e of task t , in phase p , makes to resource k
S_{tepk}	The average service time per visit entry e of task t , in phase p , requires from resource k

Figure 13: Workthread Class Parameters

The differences between a workthread class and LQM are subtle (compare Figure 14 to the corresponding LQM of Figure 1 on page 17). A workthread class has several degrees of freedom which can be used in performance experiments to generate different LQM parameters. This is discussed in Chapter 6.0.

The workthread class has a *scenario task* which is a model abstraction to characterize the external view of the scenario. It does not correspond to a customer but

models the overall control flow of the scenario. The scenario task has the same name as the scenario, with a single phase and its own “processor”.

Figure 14: Workthread Class Components

The workthread class parameters of Figure 13 are obtained by:

- *Aggregating resource demands.* This is based on the requesting task entry's phase which made the resource demand. This is straightforward, as long as resource demands are averaged taking into account the aggregate number of visits to each entry. This is discussed further in section 5.4.6.
- *Estimating the entry-entry calls.* This involves categorizing the task interactions, so that the blocking relationships are maintained. For example, the rendezvous and forwarding interaction are represented differently in an LQM. These task interactions are recognized by the event type and the flow of events in the workthread.

An informal description will now be given of how the entry-to-entry calls for a rendezvous (section 5.2.1), task phase change (section 5.2.1), and forwarding (section 5.2.3) are deduced. An algorithm for converting a workthread into a workthread class is given in section 5.4. The techniques that will be presented for deducing task interaction can be applied recursively across spawned subtraces of a workthread.

5.2.1 Recognizing a Rendezvous

As described in section 2.3.3, a rendezvous task interaction occurs when a client task is blocked until its serving task replies to its request. Figure 15 shows a trace generated during a rendezvous execution, the associated workthread (resource demands are suppressed), and the resulting workthread class. After the server task has replied to the client task (section 4.2.2), the server's post-rendezvous execution is recorded as a spawned subtrace (e.g., "W.1"). A nested rendezvous between Task B and Task C occurs during the Task A-to-Task B rendezvous.

Figure 15: Nested Rendezvous Example

A rendezvous interaction is recognized by a change in the task identifier of the workthread events, without the spawning of a child subtrace. For example in the second event (i.e. #2) of the workthread “W” of Figure 15, Task A is recorded as executing. In the third event Task B is found to be executing. This change in executing task means Task A has initiated a rendezvous with Task B and is waiting for a reply. The change in executing task of the fourth event of workthread “W” indicates Task B initiated a rendezvous with Task C. In the sixth event Task B is executing again, so, its rendezvous with Task C is over. Then Task A executes in the last event, which means Task B has completed Task A’s rendezvous. By examining the order of task execution a rendezvous can be deduced. The rendezvous is supported by the workthread class as a call from a phase in the entry of the client task to an entry of the server task. If a rendezvous occurs using asynchronous inter-task communication, it can be detected and represented in the LQM. This is discussed further in section 5.4.1.

5.2.2 Recognizing a Task Phase Change

Server post-rendezvous execution is captured in its second phase and is detected by examining the trace. A task’s phase change is recognized by the three conditions:

- 1) A **wSpawn** event type occurs in the parent trace.
- 2) In the subsequent event of the parent trace, either: the next event is executed by a different task than initiated the spawn event, or, the next event ends the trace.
- 3) The child task identified in the subtrace’s second event is the same as the task that initiated the spawning. Task execution always begins with the second event of a subtrace since the first event identifies the beginning of the workthread.

For example, in Figure 15, task C in trace “W” spawns the subtrace “W.1” which begins execution in task C (i.e., the second recorded event, #5.2). This indicates a phase change has occurred for the task that initiated the spawn (i.e., Task C). The resource demands of

the spawned subtrace are assigned to the initiating task entry's next phase. For example, the resource demands of subtrace "W.1" are assigned to the second phase of task C's, entry "W".

A task phase change is also recognized when a **taskPhase** event type is found in the workthread. The subsequent resource demands are then associated with the next phase of the task's entry. If a subtrace is spawned that has no other events than a **wBegin** and a **wEnd**, it can be ignored because it has no resource demands. Such a subtrace is "W.2" in Figure 15. In future diagrams, these subtraces are not shown.

5.2.3 Recognizing Forwarding

It can be the case that a server task completes its servicing of a client's request, and then forwards the request to another server without returning a result to the blocked client: a result is then returned to the client by the receiving server task. This client-server interaction is called *forwarding*. Forwarding only occurs during the first phase of a server task; otherwise it is considered to be an asynchronous interaction. Nested forwarding is also possible.

Forwarding causes a child subtrace to be spawned. An example of prototype execution that involves forwarding is shown in Figure 16. In this example, the User sends a request to the Dispatcher and waits for a reply. The Dispatcher services the request, and then asynchronously sends it to the Server task for completion. A trace is spawned ("W.1") because the Dispatcher and Server can execute concurrently after the **Async** statement on behalf of the same scenario (section 4.2.2). However, in this example the parent trace ("W") ends because the Dispatcher receives another request that has its own dye. The Server satisfies the User's request and replies to the user with the result.

There are two tests that are used to recognize a forwarding interaction: 1) the parent trace ends after a **wspawn** event type and 2) the child subtrace *starts execution in a different task* than initiated the spawn event. For example, in Figure 16, the trace "W" ends

Figure 16: Forwarding Example

after a **wSpawn** is initiated by the Dispatcher, and the spawned subtrace ("W.1") begins in the Server. This indicates a forward has occurred from the executing task (Dispatcher) to the task that begins in the spawned subtrace (Server).

The workthread class supports forwarding by altering the assignment of entry-entry calls. Forwarding is characterized by the workthread class as two rendezvous:

- A rendezvous from the most recently blocked task still waiting for a reply (User), to the task that performs the forwarding (Dispatcher).
- A rendezvous from the most recently blocked task still waiting for a reply (User) to the task that received the forwarded request (Server).

By representing the forwarding interaction in this fashion, the workthread class preserves the ratio of calls to each task and the task blocking relationships.

Forwarding can occur where control never returns to the task that initiated the scenario. This execution is exhibited by a pipeline of tasks using asynchronous communication, such as shown in Figure 17. This situation is modelled as forwarding, where the scenario task (e.g., "Scenario W Task") makes a call to all of the other tasks in the pipeline.

Figure 17: Task Pipeline

5.3 Workthread Classes to Performance Model

To convert a set of workthread classes into a performance model the analyst provides environment information. The environment information is used to:

- i) Identify the workthread classes that will form the workload for the model.
- ii) Assemble the model structure by combining these workthread classes into a *workthread class model*.
- iii) Use the environment information to fix the resource parameters for the entries in the workthread class model.

After the resource parameters have been assigned the workthread class model can be used as an LQM, or mapped to another type of performance model (e.g. Information Processing Graph). Other performance experiments can be constructed using the workthread class model as if it was developed from a single workthread (discussed further in Chapter 6.0). If a different type of performance model is being constructed some additional step may be needed to map the workthread class model to it.

The environment information includes the workload intensity of the model to be built. A customer class in the workload intensity is correlated with a scenario and its corresponding workthread class. An open customer class is the rate at which a scenario, and its workthread class, are submitted to the system. A closed customer class is a fixed set of customers which execute the scenario and have a (think) time between each scenario execution. For example, the two closed workthread classes "Y" and "W", of Figure 18, would each have the number of scenarios specified.

Once the workload scenarios and the corresponding workthread classes are specified, a workthread class model is assembled. This model is easily constructed since the workthread classes are independent and service entries characterize the resource demands of each one. The workthread class model is the union of the tasks and entries that are part of the workload. For example, consider constructing an LQM from the workthread

Figure 18: Individual Workthread Classes

classes of Figure 18. The first step is to develop the workthread class model by merging the individual workthread classes. Since Task A has an entry for workthread classes “W” and “Y” in Figure 18, it has two entries in the workthread class model of Figure 19. Having assembled the structural components, the resource parameters need to be assigned.

Figure 19: Workthread Class Model and LQM

Additional environment information is used in developing the resource parameters, such as: the properties of resources used in the model, task characteristic information, and resource assignment. The information needed for each resource in the model are the scheduling discipline, service rate, and the service aggregation property (section 5.1). The service rate of each resource will convert a service demand into a service time for the resource. The information needed for each task is the number of client requests that can be serviced concurrently (*concurrency level*), the assignment of task to resource (*resource assignment*), and the task priority. The population of the scenario tasks is not indicated in the task information because it is specified with the workload parameters.

The workload specification, task characteristics, and resource characteristics provide the minimum information necessary to generate a performance model. The LQM of Figure 19 is the result of the model generation process.

5.4 Workthread to Workthread Class Algorithm

An algorithm is presented in this section to convert a set of workthreads into the corresponding workthread classes. It detects rendezvous, forwarding and task phase interactions. The inputs are the scenarios (i.e. workthreads) to be converted. The algorithm expects the workthreads to be normalized, which is discussed in section 5.4.1. The operations of the algorithm are presented hierarchically, with the highest level view of the algorithm presented in section 5.4.2. The algorithm can generate a workthread class model or LQM, which is used in the case study of section Chapter 7.0.

The notation used by the algorithm is found in Figure 20. The resulting workthread classes are defined by the parameters¹ of Figure 21. The event types that this algorithm considers are: **wBegin**, **wEnd**, **wDelay**, and **wSpawn**.

The algorithm makes use of the following points:

- Each scenario has a *scenario task*.

1. The notation is reproduced from Figure 13 on page 55.

T_{bn}	The task associated with event n of (sub)trace b .
E_{bn}	The task entry associated with event n of (sub)trace b .
Y_{bn}	The event type of event n of (sub)trace b .
J_{bn}	The set of resources used by event n of (sub)trace b .
V_{bnk}	The number of visits made to resource k due to event n in (sub)trace b .
D_{bnk}	The total service time required from resource k due to event n of (sub)trace b .
L	The scenarios specified by the analyst that constitute the workload.
L'	The subtraces whose ancestor traces are due to a scenario specified in L .
w	Identifier for the scenario that is being analyzed.
b	Identifier for the subtrace that is being analyzed.
n	Index for the event that is being examined.
P_{bc}	The phase of the current entry of task c , of subtrace b .
$isClient[c]$	TRUE if task c is acting as a client of another task, FALSE otherwise.
$length[b]$	Returns the number of workload events in (sub)trace b , which start at event 1.
$absorb[j]$	Returns TRUE if resource j has the service aggregation property, FALSE otherwise.
$readWorkload(L)$	Assigns the workload scenarios specified by the analyst to L .
$push(Q, a)$	Push the element onto the top of the stack Q .
$pop(Q)$	Remove the element on the top of the stack Q .
$empty(Q)$	Return TRUE if the stack Q is empty. Otherwise, return FALSE.
$top(Q)$	Return the value of the top element of stack Q .
M_{bn}^w	Name and occurrence of the (sub)trace which is spawned by event n of subtrace b .
M_{bn}^d	Duration that the task is delayed, as recorded by event n of subtrace b .

Figure 20: Algorithm Notation

- A service entry's name does not change during the processing.
- The name of a service entry is the same as the corresponding *trace name* of the scenario. As a result, the notation for the number of calls from the entry and phase of the client task to an entry of a server task (i.e. $C_{t',e',p',te}$) is simplified. The client and server entries are the same, so the workthread class parameter $C_{t',e',p',te}$ becomes $V_{t'wp't}$.
- Service demands made by a scenario task are assumed to be requests to a delay server. So, if w is a scenario, then D_{cwpw} is the demand by the entry w (the trace name) of task c , in phase p , to a delay server. This is used to model the think time for a closed class.
- Tasks and resources are treated in the same manner, with the following exceptions: i) tasks do not make use of the absorption property; ii) a task, other than the scenario task, cannot have a service demand made on it, and iii) resources cannot be clients.
- Each workthread event can use more than one resource, as well as make multiple visits to the same resource.

When there is no distinction between a task and resource, the term "*component*" shall be used. Also, the term "*visit*" will be used in place of call. It is not necessary that the workthread has had the service aggregation property applied in advance because this algorithm will perform it.

Before the traces are analyzed they are normalized, which is the subject of the next subsection.

5.4.1 Trace Normalization

Traces are manipulated into a standard format to simplify the analysis. There are two steps to the *trace normalization*:

$I_{tep k}$	The aggregate number of visits entry e of task t , in phase p , makes to resource k
$D_{tep k}$	The aggregate demand entry e of task t , in phase p , makes on resource k
$totalCalls[w, t, e]$	The total number of times task t 's entry e is called by the workthread w .
K_{te}	Set of resources used by entry e of task t .
T_w	Set of all tasks used in scenario w
$V_{tep k}$	The average number of visits entry e of task t , in phase p , makes to component k
$S_{tep k}$	The average service time per visit entry e of task t , in phase p , requires from resource k

Figure 21: Resulting Model Parameters

- Merging the parent and child (sub)trace which are involved in a forwarding interaction. This helps to deduce entry-to-entry calls.
- Rewriting traces of a rendezvous that uses asynchronous communication into a standard format.

These two steps are now examined.

Trace normalization merges the two subtraces that are involved in a forwarding interaction into a single subtrace. The new event type **wForward** is added to the resulting trace to indicate a forward has taken place. If this simplification is not made, then a client task's identity must be maintained across multiple subtraces. As an example of normalization, the angio trace of Figure 16 is reconsidered and shown in normalized form in Figure 22. When the forwarding is detected, the traces are manipulated in the following fashion:

- i) The parent trace removes the **wSpawn** event type for the forwarding interaction, but leaves the resource demands of the statement intact.
- ii) The **wEnd** event type of the parent subtrace is removed.
- iii) The **wBegin** event type of the child subtrace is replaced with a **wForward**.

The task associated with the **wForward** is the task initiating the forwarding (i.e., Server).

iv) The child subtrace ("W.1") is concatenated to the parent trace ("W") and has the trace name of its events rewritten to that of the parent trace.

The result of the trace normalization for forwarding is shown in Figure 22.

Figure 22: Normalized Trace of Forwarding

A prototype environment may not have a statement that implements a rendezvous directly. To overcome this, a rendezvous interaction could be composed of two asynchronous communications. The manner in which the trace reduction analysis supports this will now be discussed. Figure 23 is an example of prototype execution where asynchronous inter-task communication is used to perform a rendezvous.

A rendezvous interaction that uses asynchronous communication generates a different set of subtraces than recorded by a normal rendezvous. A non-obvious point is that clarification of these subtraces is needed to generate a correct performance model. Confusion can arise whether trace "W" of Figure 23 should be interpreted as a forward interaction or as a rendezvous. The distinction is necessary because different contention effects arise in the performance model: a forward would allow Task A to service client tasks while Task B is busy, whereas a rendezvous would mean Task A is blocked while

Task B is processing its request. The trace analysis is augmented to distinguish these cases by: using additional instrumentation of the prototype and normalization of the traces into a standard rendezvous form.

Figure 23: Execution of Rendezvous Using Asynchronous Communication

The additional instrumentation records a tag in the angio trace to indicate that the asynchronous communication is intended to form a rendezvous. In Figure 24, the trace “W” has recorded such a tag using a **wReport** event type, with the parameter “DoingRnz”. This indicates that the communication is intended to be a rendezvous, although asynchronous communication is used as a vehicle. It is assumed that a prototype statement has been constructed in this manner, to perform a rendezvous that uses asynchronous communication. When this “tag” is recognized the traces are normalized into a standard format for a rendezvous.

In the standard rendezvous format (Figure 23), a rendezvous begins and ends in the same trace. If server post-rendezvous execution occurs, a subtrace is spawned to model the phase change. It is assumed a rendezvous using asynchronous communication occurs when a parent trace (“W.1”) does not end immediately after a **wSpawn** event type. The asynchronous rendezvous trace normalization manipulates the trace as follows (Figure 24):

- i) The tag in the trace identifies that a rendezvous implemented using asynchronous communication has begun. The subtraces involved in this interaction are normalized.

- ii) The trace that is the reply to the client (W.1.1) has its **wBegin** event type removed.
- iii) The reply to the client is specified by a **wSpawn** event type in the parent trace, which is where the subtrace for the reply should be attached. The parent trace is split into two traces after the **wSpawn** event, a past and future subtrace.
- iv) The past subtrace is concatenated with the subtrace that completed the rendezvous (W.1.1) and its events take the parent trace's name (W.1).
- v) The future subtrace takes on the trace name of the original spawned trace ("W.1.1"). Then the **wSpawn** event does not need to change the event parameter for the spawned trace.

The result of normalizing the angio trace of Figure 23 is shown in Figure 24.

Figure 24: Traces of Rendezvous Using Asynchronous Communication

Having discussed the trace normalization, the algorithm for generating the workload class can be discussed.

5.4.2 Algorithm for Generating the Workthread Class

The algorithm for generating the workload class is shown in Figure 25, and has the following steps:

- i) Read the trace names for the scenarios to examine. This operation also initializes the *absorb* [*j*] and *length* [*b*] data.
- ii) Initialize the set of tasks involved in the workload (τ_w) to the empty set (Eq 25.2) and create the scenario task.
- iii) subtraces are analyzed without interruption. The set of subtraces to examine (L') is initially set to the scenario (w) (Eq 25.4). As the events are examined, a **wspawn** event type may indicate a subtrace to be evaluated. If so, the subtrace is identified by the event parameter $M_{b_n}^w$, which is added to the list of subtraces to be evaluated later (Eq 25.6).
- iv) Each subtrace of the scenario has its workload events analyzed by the operation *ProcessEvent*. Events are examined in sequence. This operation aggregates resource demands and entry-entry calls by scenario, task, and phase.
- v) Once aggregation is complete for each scenario, the parameters are averaged to generate the workload classes (Eq 25.7).

Having presented the high-level algorithm, the constituent operations are explained further. The first operation is task creation (*CreateTask()*), which will introduce the notation. This is followed by an explanation of the operations for the resource aggregation (*ProcessEvent()*), the entry-entry call aggregation (*EntryCallsAggregate()*), and parameter averaging (*ResourceDemandAvg()*).

```

BuildWorkthreadClass()
begin
    ReadWorkload(L)                                % Initialize workload and variables   Eq 25.1
    For  $\forall w \in L$ 
         $T_w \leftarrow \emptyset$                         % Initialize set of tasks to empty       Eq 25.2
        CreateTask(w,w)                             % Create scenario task                   Eq 25.3
         $L' = w$                                        % Set initial subtrace to trace name    Eq 25.4
        For  $\forall b \in L'$ 
            For  $n = 1 \dots \text{length}[b]$              % Examine each event in subtrace
                ProcessEvent(n, w, b)                % Analyze the sequence of events       Eq 25.5
                if ( $Y_{bn} = w\text{Spawn}$ )                % Is this a spawn event type
                     $L' = L' \cup M_{bn}^w$             % Add subtrace for later analysis      Eq 25.6
                endif
            endFor
        endFor
    endFor
    ResourceDemandAvg(L)                            % Average resource demands &          Eq 25.7
visits
end

```

Figure 25: Model Generation Algorithm

5.4.3 Task Creation Algorithm

The algorithm for creating a task is shown in Figure 26. The steps for creating a task are:

- i) If the task was not created previously, then create it and allocate the data structures.
- ii) The analysis keeps track of tasks that act as clients using the array *isClient*. Tasks will become clients when they visit other tasks. This is not the case for a newly created task so the value is initialized to FALSE (Eq 26.2).
- iii) Two variables are initialized to allow tasks to be treated as resources. The absorption property does not apply for a task so it is set to FALSE (Eq 26.2). Since tasks cannot make demands on other tasks, the aggregate demand made by this task to any component is initialized to zero (Eq 26.6).
- iv) The visits from the newly created task to all components are set to zero (Eq 26.5 and Eq 26.7).

v) The new task is then added to the set of active tasks (T_w) for the scenario (Eq 26.8).

5.4.4 Resource Demand Aggregation

The analysis of each workthread event is shown by Figure 27. The operation begins by creating the task of the current event if necessary ($CreateTask(w, c)$). If the current event type is a **wDelay** then a visit from the current task's entry to the scenario delay server (Eq 27.4) is made and its demand is increased by the value of the event parameter (M_{bn}^d). If the event type is not a **wDelay** the resource demands for the event are aggregated, which is explained below. This is followed by the estimation of entry-entry calls ($EntryCallsAggregate(w, b, n)$) which is explained in the next section.

The aggregation of resource demands begins at Eq 27.5 by adding the resource used by the event (j) to the set of resources used by the task in that scenario (K_{wc}). The visits to

each resource are aggregated, using the service aggregation property (section 5.1).

Applying the service aggregation property has one of three results:

- *Service aggregation does not apply to the resource.* In this instance the number of visits to the resource (v_{bnj}) are added to its aggregate number of visits (Eq 27.4) for that entry's phase;
- *Service aggregation applies with boundary conditions.* When a boundary condition exists a single additional visit is made to the resource (Eq 27.4). Some boundary conditions are: there have been no visits to that resource yet, a task change occurred between the last event and the current event which requires a visit to the resource (e.g., context switch), a phase change occurred, or the last event did not visit that same resource.
- *Service aggregation applies and the previous event made a visit to the resource.* No additional visits are necessary.

In all three cases, the service time for the event is added to the service time of previous visits to the resource. The operation of aggregating the entry-entry calls is explained next.

5.4.5 Entry-Entry Call Aggregation Operation

The operation described by Figure 28 detects the task interactions and phase changes. Phase change detection will be discussed first. Then the rendezvous detection is presented, followed by the detection of forwarding.

The steps for detecting a task's phase change are Eq. 28.4 to Eq 28.8 which uses two separate tests. The first test is that a **wSpawn** event type occurs in the parent trace. The second test is that the parent trace's next event is not a **wForward** and the child subtrace starts in the same task that initiated the spawn event. If the second test fails, then trace analysis cannot continue (Eq 28.8) and an error is reported (e.g., asynchronous execution is occurring that is not post-rendezvous execution). If the tests succeed, then the task's

```

ProcessEvent (n, w, b)
begin
    CreateTask (w, Tbn)                                % Create task if needed           Eq 27.1
    c = Tbn                                                % Task of the current event       Eq 27.2
    p = Pbc                                                % Phase of current task in subtrace Eq 27.3

    if (Ybn = wDelay)                                     % Delay event type
        Icwpw = Icwpw + 1                                % Visit to delay server          Eq 27.4
        Dcwpw = Dcwpw + Mbnd                            % Aggregate time at delay server  Eq 27.5
    else
        For  $\forall j \in J_{bn}$                                 % Do for each resource used
            Kwc = Kwc  $\cup$  j                            % Add to set of resources used by task Eq 27.6
            Dcwpj = Dcwpj + Dbnj                        % Aggregate resource demand      Eq 27.7
            if ( $\neg \text{absorb}(j)$ )
                Icwpj = Icwpj + Vbnj                % No absorption property          Eq 27.8
            else
                if ( $(T_{bn} \neq T_{b(n-1)}) \vee (I_{cwpj} = 0) \vee (j \notin J_{b(n-1)})$ )
                    Icwpj = Icwpj + 1                % Boundary condition             Eq 27.9
                endif
            endif
        endFor
        EntryCallsAggregate (w, b, n)                    % Aggregate entry-entry calls     Eq 27.10
    endif
end

```

Figure 27: Workthread Event Analysis Operation

phase is incremented (Eq 28.6) and the task's new phase is associated with the spawned trace (Eq 28.7). Execution then returns to a higher-level operation.

The steps for detecting rendezvous and forwarding are Eq. 28.9 to Eq 28.21. It is only when the current task changes (i.e., $c \neq T_{bn-1}$) that a call to another task is made and a rendezvous or forwarding interaction occurs. The analysis of forwarding makes the use of a stack.

Rendezvous detection occurs by analyzing the manner in which the executing task changes. As discussed in section 5.2.1, when task identifiers of sequential events in the subthread change there are two possibilities. These are distinguished by a task's Boolean flag ("*isClient*") to indicate if a reply is being sent to a client, or a request is being sent to a server:

```

EntryCallsAggregate(w, b, n)
begin
    c = Tbn                                     % Task of the current event                      Eq 28.1
    if ( (n = 1) ∧ (w = b) )
        Iwwlc = 1                                % Initial visit from scenario task                Eq 28.2
        push(Q, w)                                % To model task pipeline, scenario task is first Eq 28.3
    else
        if (Ybn = wSpawn)                        % Potential phase change
            b' = Mbnw                            % Name of spawned subtrace                        Eq 28.4
            c' = Tb'2                             % First executing task of action in the subtrace Eq 28.5
            if (c = c') ∧ (Yb(n+1) ≠ wForward) % Is it a phase change?
                Pbc = Pbc + 1                    % Increase task phase in current subtrace        Eq 28.6
                Pb'c = Pbc                        % Increase task phase in spawned subtrace        Eq 28.7
                return                             % Done event analysis
            else
                ERROR                             % Asynchronous, concurrent execution            Eq 28.8
            endif
        endif
    endif

    if (c ≠ Tb(n-1))                             % Continue analysis only on a task change
        if (isClient[c])                          % Current task is a client
            isClient[c] = False % Return to client from server Eq 28.9
            pop(Q)                               % Waiting task changes                      Eq 28.10
        else
            if (Ybn = wForward) % Forwarding event
                if (empty(Q)) % Check for task to assign forward to
                    ERROR % Forwarding not in first phase Eq 28.11
                endif
                c' = top(Q) % Client task is from forwarding stack Eq 28.12
                if (isClient[c]) % Current task is a client
                    isClient[Tb(n-1)] = False Eq 28.13
                    pop(Q) % Return to client Eq 28.14
                    return; Eq 28.15
                endif
            else
                c' = Tb(n-1) % Client task is last active task Eq 28.16
                isClient[c'] = True % Last task was a client Eq 28.17
                push(Q, c') % Remember client task in case of forward Eq 28.18
            endif
            p' = Pbc' % Phase of client task Eq 28.19
            Ic'wp'c = Ic'wp'c + 1 % Add a call from the client to the server Eq 28.20
            Kc'w = Kc'w ∪ c % Set of components used by the task Eq 28.21
        endif
    endif
end

```

Figure 28: Entry-Entry Call Aggregation Operation

- *Task of the current event is not a client:* The change in the task identifier means the task of the last event (c') made a call to the current event's task (c). So c is now acting as a server task. The *isClient* flag of the previous task (c') is then set to TRUE because it has become a client (Eq 28.17). The aggregate number of visits from the client task to the serving task is incremented, taking into account the entry and phase of the client task (Eq 28.20). The serving task is also added to set of components used by the client task (Eq 28.21).
- *Task of the current event is acting as a server:* The task change is a return of control to a client task (c) from a serving task (c'). This is not considered a visit to the client task, because it was accounted for in Case 1. Since the current task is no longer a client, its *isClient* flag is cleared (Eq. 28.9). This is also true when a server task forwards the reply to the client task waiting for a result (Eq 26.2). No further action is necessary.

If there was no task change then there is no reason to check for a rendezvous.

As discussed in section 5.4.1, forwarding is detected by the event type **wForward**. Analysis of forwarding is similar to that for a rendezvous, except care is taken to remember which task should be considered a client. Since forwarding can occur several times in a workthread, one after the other, a stack (Q) is used to manage the identification of clients when forwarding is detected. The top of the stack ($top(Q)$) stores the client task identifier that is assigned the call to the new server task. Forwarding is concerned with three possibilities when a task change occurs:

- *The last executing task is a client.* Then push it on the stack because it is involved in a rendezvous (Eq 28.19) and it will have forwarding calls associated with it.
- *The last executing task is a server.* This event signifies a return in control from a server task to a client task. The task on top of the stack is the task is the server

task, so it must be changed to the client task. So pop the server task off of the stack so that the client becomes the top of the stack (Eq. 28.10).

- *The last executing task performed a forward.* The task performed a forward so it is not stored on the stack. Since this is a forward, the aggregate number of visits from the client task ($top(Q)$) to serving task is incremented (Eq 28.20). No other stack operations need to be performed.

The algorithm reports task interactions that are beyond its current analysis capability. These limitations fall under two categories and they are reported by the algorithm as analysis errors. The first limitation is when there is asynchronous execution which is not a phase change (Eq 28.8). The second limitation is when a forward occurs that is not in the first phase of a task. This is detected by the top of the client stack being empty (Eq 28.11). In both cases processing halts and a message is reported.

The initialization of $EntryCallsAggregate(w, b, n)$ is now discussed. To permit the modeling of task pipelines the scenario task is pushed onto the client stack during initialization (Eq 28.3). Also, if this is the first event of trace that initiates a scenario (i.e., the scenario name and subtrace names match), then a visit is made from the scenario task to the first task in the subtrace (Eq 28.2).

This concludes the processing of events. The last step is to average the workthread class parameter values.

5.4.6 Workthread Class Parameter Averaging

Once the visits and demands have been aggregated over all (sub)traces, averaging is applied for the following parameters. The averaging is done for each scenario (Figure 29) and it involves averaging the: visits from entries to resources, calls from entry-to-entry, and service time per visit. This step takes into account the total number of times a task's entry is called, which is stored as $totalCalls[w, c, e]$. The value for $totalCalls[w, c, e]$ is found using a depth-first search algorithm [44] for the workthread class structure.


```

ResourceDemandAvg(L)
begin
  For  $\forall w \in L, \forall c \in T, \forall p \in c, \forall j \in K_{wc}$ 
    
$$V_{cwpj} = \frac{I_{cwpj}}{totalCalls[w, c, w]} \quad \% \text{ Average visits and calls} \quad \text{Eq 29.1}$$

    
$$S_{cwpj} = \frac{D_{cwpj}}{I_{cwpj}} \quad \% \text{ Average service time per call} \quad \text{Eq 29.2}$$

  endFor
end

```

Figure 29: Workthread Class Parameter Averaging Operation

5.5 Model Generation Summary

This chapter has presented the transformations to automatically generate a workthread class and performance model from an angio trace. An algorithm for generating a workthread class from an angio trace has been given. A benefit of this algorithm is that it gives the structural parameters for an LQM. An analyst can then provide the environmental information to complete the LQM.

Some possible enhancements to the algorithm are:

- To estimate the coefficient of variation for resource visits and the service time per visit. Service times are currently assumed to be exponentially distributed with a geometrically distributed number of calls.
- Provide support for explicit entry names and phase changes.
- Extensions to the types of interactions that can be accommodated could be (e.g., asynchronous execution).

Chapter 6.0 Structural Transformations

“The goal of every performance study is either to compare different alternatives or to find the optimal parameter value.” R. Jain [42]

This chapter considers how the workload classes generated in the previous chapter can be used to conduct performance experiments by varying a model's structure, without returning to the prototype environment. This allows the analyst to bypass steps #1 to #4 of Figure 2, reducing the modeling time and effort. This can be used to quickly evaluate the performance of several designs which vary only in structure. These new experiments are developed by manipulating the structure of a previously generated workload class and regenerating the performance model. This section is a high-level description of some of the modifications which can be used to construct alternative experiments. It is assumed that a graphical tool, such as Timebench [64] or DASS [36], is used to alter the workload class structure in an intuitive fashion.

The structure of each workload class can be changed since workload classes are independent of each other. There are three manners in which the structure of a workload class can be manipulated for experimental purposes: *workload class clustering*, *entry movement* from one task to another task, and *workload class cloning* with reassignment of resource demands. The workload classes from the previous chapter will illustrate the approach, and are reproduced in Figure 30.

6.1 Workload Class Clustering

A new workload class can be created by clustering a set of existing workload classes in a way that the analyst wants (e.g. weighted averaging). The resulting workload class characterizes the combined behavior. This may be done to construct a workload

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

Figure 30: Individual Workthread Classes

class for a transaction that incorporates several scenarios, where each scenario describes the execution of a single user command. An example of this is found in Figure 31, where the user issues two scenario “Y” commands and three scenario “W” commands per transaction. Other reasons for clustering are to accommodate performance evaluation tool shortcomings (e.g. limited number of customer classes), or to combine workthread classes with the same scenario name but different options (e.g. input parameter values or system state).

Figure 31: Clustered Workthread Classes

6.2 Movement of Entries

Entries can be moved from one task to another task. This allows the analyst to estimate the effect on performance of moving a service entry up, down, or horizontally in the client-server hierarchy. This is analogous to moving functionality from one task to another, for conducting experiments about the partitioning of functions between tasks. It is assumed that the movement of a service is feasible and that little resource demand changes occur due to the movement. Entry movement can occur in three fashions:

- *Vertical movement:* The task entry is moved up or down in the client-server hierarchy. Figure 32 is an example of vertical movement, where entry Y of Task A in Figure 30 has been moved to Task B. When an entry is moved the clients' calls to that entry do not change, neither do the entry's parameters. Some resource demands may become invalid (e.g. inter-task communication is unnecessary) and must be compensated for by analyzing the workthread or angio trace to adjust the costs.
- *Horizontal movement:* A task entry is moved horizontally from one task to another, which may involve the creation of a new task. For example, entry Y of Task A and entry W of Task A (Figure 18 on page 63) are combined when generating the composite Workthread class in Figure 19. However, they are separated into the different tasks Task A and Task D in Figure 33 by the horizontal movement of entries. To accomplish this, Task D was added. The environment information for Task D would also have to be supplied.
- *Service fusion:* Entries within the same task are combined together. For example, the entries of Task C in the clustered workthread class of Figure 31, have been fused together in Figure 34.

The vertical or horizontal movement of an entry is limited to the scope of a workthread class. When entries are moved and fused, averaging of resource demands and device visits is needed to account for the structural changes (section 5.4.6).

Figure 32: Vertical Movement of Entries

Figure 33: Horizontal Movement of Entries

6.3 Class Cloning and Resource Reassignment

If the workthreads of several scenarios are the same except that they use different resources, then workthread class cloning with resources reassignment can be used to construct the model. An example of this is shown in Figure 35, where there are two scenarios that differ only in the disk that is used. To model the scenarios, the workthread class Y is cloned to produce two separate workthread classes “Y1” and “Y2”. To model the differing disk usage, resource reassignment occurs for workthread class “Y2”.

Figure 34: Service Fusion of Entries

Resource assignment binds resource demands in the workthread class to a particular resource in the model. An example of resource assignment that has been presented was the assigning of a task to a CPU (section 5.3). Reassignment of an entry's resource demands for one device to another device also be done. For example, the entry "Y1" of Task C use a different resource (e.g. Disk1) than entry "Y2" does. Resource reassignment is limited to the scope of a task's entry because the entry represents a particular service.

Figure 35: Resource Assignment of Multiple Workthread Class Instances

6.4 Structure Transformations Summary

The workthread class has been found to be a flexible abstraction of software system execution, with many degrees of freedom for structural and resource parameter experimentation. Structural manipulations on the workthread class can be equated with modifications to the software structure or with the external environment (e.g. clustering).

Chapter 7.0 Workthread Case Study

“Our aim is to show the essential elements of the language in real programs, but without getting bogged down in details, formal rules, and exceptions. At this point, we are not trying to be complete or even concise (save that the examples are meant to be correct).” Brian Kernighan and Dennis Ritchie [13]

This chapter will illustrate the model generation technique by developing a Layered Queueing Model automatically from a prototype. It will produce a LQM with the same model structure and parameters as if it had been manually constructed.

This case study considers an On-Line Transaction Processing (*OLTP*) system. In this system, a user will enter commands to update entries in a database and wait for a response. An example of such an application is an air-line reservation system. To estimate the response time of a user's command a performance model is generated using the technique described in chapters four and five (steps #1 through #6 of Figure 2 on page 29). The model type chosen is a Layered Queueing Model (LQM), since it best reflects the class of application (i.e., layered client-server type system). This is followed by examples of altering the workthread class structure to predict the performance of alternative software system configurations.

7.1 Prototype of an OLTP System

The case study is based on the real-life system presented in [12], [7], and [61]. The software task architecture is organized as client tasks making requests to server tasks. The inter-task communication occurs in a manner similar to a Remote Procedure Call [11]. Each server task provides a single service such that it retains no memory from the servicing of one request to the next. Server tasks can be replicated to provide concurrent execution of

the same service. If there are replicated servers, the operating system manages the sending of a request to a free server.

The software task architecture is layered; tasks at lower layers act as servers to tasks above them. The tasks in the prototype are shown in Figure 36. The following operations are satisfied by the tasks:

- *User*: Submits a command to the OLTP system.
- *Communication Task (CM)*: Interface for a single user's terminal input and output communication.
- *Dispatcher (DP)*: The Dispatcher is responsible for three operations: field validation of user input, sending of application commands to application servers, and formatting of data for presentation on the user's terminal.
- *Name Server (NS)*: Map an identifier to the message address of an available server.
- *Application Task (AP)*: Decomposes a user's command into primitive Structure Query Language (SQL) operations on the database. The primitive SQL database operations are sent to an SQL disk task for execution.
- *SQL Disk Task (DT)*: Translates low level SQL database operation into read and write operations on a particular disk volume. In our example, the low level SQL database operations are the most detailed statements that are represented.

Users can submit two types of commands: 1) a validation command (VC), which manipulates entered data and does not need a database access, and 2) an application command (AC) which always adds a record to an SQL table that is partitioned across two disks. Both commands will be discussed in detail.

Figure 36: OLTP System Configuration

The case study uses the workload intensity given in Table 5. The user commands are independent and correspond to separate scenarios.

Scenario	Trace Name	User Population	User Think Time
Validation Command	VC	50	3.0 seconds
Application Command	AC	60	4.0 seconds

Table 5: OLTP Workload Intensity

The system configuration that is modelled is illustrated in Figure 36. It has two CPUs that each have a disk drive. The resource characteristics for this system configuration are given in Table 6.

Resource	Scheduling	Service Aggregation	Service Rate per Visit
Disk1, Disk2	FIFO	No	0.018 (secs)
CPU1, CPU2	PS	Yes	1/(5 Mips)

Table 6: OLTP Performance Model Resource Definition :

To complete the environment information, data is needed about the software task architecture. Each task needs to have the following information defined: its CPU, priority, and concurrency level. The CPU that each task executes on is shown in Figure 36. It is also assumed that all tasks have the same priority. The concurrency level for each task is presented in Table 7. The number of "User" tasks is not indicated in the task information because they are specified as the workload intensity (Table 5). To provide for the one-to-

one correspondence between the User and Communication Tasks, the communication tasks are separated based on the scenario.

Task Identifier	Concurrency Level
Communication Task Application Command	60
Communication Task Validation Command	50
Dispatcher (DP)	1
Name Server (NS)	1
Application Task (AP)	3
Disk Task 1 (DT1)	3
Disk Task 2 (DT2)	3

Table 7: OLTP Task Information

The case study prototype was constructed using the MLog prototyping environment (section 4.3). MLog was used because it was a general purpose prototyping environment that was available. MLog was modified to generate angio traces.

The server tasks of the OLTP system were active objects and provided the appropriate services. The prototype did not emulate resources since this is not needed by the technique (i.e., no disks or CPUs in the prototype). The methods of MLog were overloaded to record the event type information. The statements which had resource functions are shown in Table 8, along with a brief description of their operation. The resource function for the "Acc" statement is of particular interest because it uses information from the angio event (shown as arguments with "\$" signs) to identify the disk used and to estimate the number of disk accesses.

Statement	Resource Function	Description
Send	(CPU, 5000 instructions)	Synchronous message send to another task.
Reply	(CPU, 5000)	Reply to a received synchronous message.
In	(CPU, 10000)	Input from user's terminal.
Out	(CPU, 10000)	Output of formatted data to user's terminal.
Async	(CPU, 5000)	Asynchronous message send to another task.
Rcv	(CPU, 5000)	Block until reception of a message from another task.
Cvt	(CPU, 5000)	Convert user command to inter-task message
Fmt	(CPU, 10000)	Format a screen of data for output to the user.

Table 8: Statement IDs and Resource Functions for the OLTP System

Statement	Resource Function	Description
Lkup	(CPU, 2500)	Lookup a task's message port address.
SqlAdd	(CPU, 12000)	Perform an SQL add operation on a table.
SqlPrepare	(CPU, 5000)	Preparation for servicing an SQL command.
Acc	(\$1, \$2)(CPU, 1500)	Access a disk: \$1 is the disk used, and \$2 is the number of disk accesses.

Table 8: Statement IDs and Resource Functions for the OLTP System

As discussed in Chapter 3.0, a motivation for using a prototype is to capture the effect of data dependencies on behavior and performance. A scenario's input data and the system state may affect looping and branching, and therefore, the resource demands. In this example, the logical and physical aspects of data storage provided by the SQL interface create data dependencies that affect disk usage [30]. The logical representation insulates the application from the location and physical representation of the data. The physical characteristics (e.g., file partitioning, file placement) influence how SQL statements are translated into resource selection and demands. Typically, an SQL optimizer uses this information to minimize disk accesses. The prototype can include SQL optimizer heuristics, such as are found in [78] and [46], to determine the resource demands of each SQL statement at run-time. The performance of different database configurations could then be explored by making data storage characteristics a scenario option. Although one particular database configuration is presented in this example, many alternative configurations could be examined.

Having presented the target system and the prototype devised to emulate it, the discussion can proceed to the generation of angio traces.

7.2 Angio Traces of Scenarios

The following sections provide examples of traces for "validation" and "application" command scenarios. The example traces will have rendezvous and forwarding task interactions.

7.2.1 Validation Command Scenario Trace

The validation command scenario trace of Figure 37 illustrates how a dye propagates during execution and the events are recorded.¹ The occurrence number two was automatically assigned by the prototyping environment during execution.

Figure 37: Validation Command Scenario Execution and Angio Trace

The sequence numbers in Figure 37 identify significant steps during the prototype execution of a validation command:

- #1) The User (US) issues the validation command, which causes the trace "VC, occurrence=2" to begin. Then US waits for a reply.
- #2) The Communication Task (CM) receives the validation command, converts it to an inter-task message, sends the message to the Dispatcher (DP), and waits for a reply.

1. Each event in the trace has the form: "<trace name>, <occurrence number>, <task>, <statement ID>, <event type>, <optional event parameters >". In subsequent angio trace figures, the trace name and occurrence number will be recorded only at the top of the trace.

#3) DP receives the message, performs the validation, and then replies to CM. The *angio* statement instrumentation for **Reply** spawns the child subtrace "VC.1".

#4) CM receives the response, and outputs it to the user's terminal.

#5) The User receives the reply which ends the validation command and the scenario.

The validation scenario has examples of a *rendezvous*. The User (client task) makes a request to a CM (server task) and then waits for the reply; CM satisfies the client's request and replies to the User; the User receives the reply and continues with the next command. A nested rendezvous between CM and DP occurs during the User-CM rendezvous.

As discussed in section 4.2.2, whenever concurrent execution that originates from the same dye can occur a child subtrace is spawned to maintain causal information. Since the potential for concurrent activities exists with a rendezvous reply, the subtrace "VC.1" was spawned in task DP by the *angio* statement instrumentation for **Reply**. However, DP did not have post-rendezvous execution so the subtrace "VC.1" immediately ended. It will be deleted in post-processing.

7.2.2 Application Command Scenario Trace

An application command scenario will now be examined (Figure 38), which includes *forwarding* interactions. As discussed in section 5.2.3, forwarding causes a child subtrace to be spawned. The servicing of the application command occurs as follows:

#1) The User issues the application command, which begins the scenario and injects the dye with the trace name "AP". The User waits for a reply.

#2) The Communication Task (CM) receives the application command, sends it to the Dispatcher (DP), and waits for a reply.

- #3) DP requests, by rendezvous, an application server's address from the Name Server (NS).
- #4) NS replies with the address of an application server to DP. The statement instrumentation for **"Reply"** generates the child subtrace "AC.1".
- #5) DP "forwards" the command to the Application server (AP) by asynchronous communication. The angio statement instrumentation for **"Async"** spawns the subtrace "AC.2".
- #6) AP receives the command, sends an SQL command to SQL Disk Task 1 (DT1), and waits for a result. The subtrace "AC.2" is recorded as beginning in DP and the statement execution starts with the next event that is recorded in the task AP. This step illustrates that the task that initiates a spawn begins the trace, and the next recorded event is the first executed statement.
- #7) DT1 executes the SQL command and sends the result to the client task AP. Based on data values included in the request, the number of disk accesses is estimated by the SQL optimizer heuristics, and recorded in the trace as event parameters. In this case, there are two consecutive disk accesses to "Disk1" recorded in the trace.
- #8) After replying, DT1 continues execution in preparation for servicing another SQL command (e.g., returns memory to the free pool). This happens concurrently with continued servicing of the application command, so the subtrace "AC.2.1" is spawned.
- #9) AP receives the results from DT1, sends another SQL command to SQL Disk Task 2 (DT2), and waits for the result.
- #10) DT2 makes three consecutive accesses to "Disk2" using the SQL optimizer heuristics, and replies to the application server.

#11) After replying, DT2 prepares for its next SQL command and the subtrace “AC.2.2” is spawned.

#12) The application command has been satisfied. AP sends a response back to the Dispatcher using asynchronous communication, spawning subtrace “AC.2.3”.

#13) DP acquires the address of the Communication Task from the Name Server, which also spawns the subtrace “AC.2.3.1”.

#14) DP sends the response to CM using asynchronous communication that spawns the subtrace “AC.2.3.2”.

#15) CM sends the reply to the User, and the scenario ends.

Figure 38: Application Command Scenario Execution

The application scenario in Figure 38 has several rendezvous, forwarding, and phase change interactions. Each forwarding interaction is initiated by asynchronous inter-task communication that spawns a subtrace.

The subtraces “AC.2.1” and “AC.2.2” are examples of valid post-rendezvous execution. The subtraces “AC.1” and “AC.2.3.1” do not have any statements with resource

demands so they can be ignored in further analysis. This is also true of the subtrace “v C.1” of the validation scenario.

Figure 39: Application Command Scenario Angio Trace

Having generated the traces for the two types of scenarios, the next step is to convert them into workthreads.

7.3 OLTP Workthreads

Translation of an angio trace to a workthread was discussed in section 5.1, and begins by estimating the resource demands of each statement using resource functions. In the case study, the *statement ID* of each event is used to determine which resource function to apply from Table 8. For example, the statement ID “In” corresponds to 5000 CPU instructions. An example of using event parameters for resource selection is the first disk

access operation “**Acc**” of trace “AC.1”, which records that “Disk1” is selected with one access (Figure 41). Overhead costs which are not included in a resource function could be incorporated in the workthread, such as the context switch cost can be added whenever the executing task changes in the trace.

Figure 40: Validation Command Workthread

The *service aggregation property* is applied to sum resource demands, to the same resource, into a single visit with a total service time. For example, in Figure 40, the communication task uses the CPU for the sequential execution of “**In**” (CPU, 10000), “**Cvt**” (CPU, 5000), and “**Send**” (CPU, 5000) statements, which are summed into a visit with a demand of (CPU, 20000). The service aggregation property applies for those resources identified by the analyst, and only when the same resource is sequentially accessed by the same task. For example, in workthread “AC.2” of Figure 41, Disk Task1 (DT1) does not have all of its CPU demands combined into a single visit because the CPU resource demands do not occur sequentially: disk requests intervene between the CPU requests.

To complete the workthread parameters (Figure 12 on page 53), the entry names need to be identified. The entry names in this example will default to the scenario name. For example, the validation scenario workthread (Figure 40) assigns resource demands to the service entry “VC” for each task.

7.4 OLTP Workthread Classes

The workthreads are converted into the workthread class parameters (Figure 13 on page 55) by aggregating resource demands and estimating entry-entry calls. Estimating the

Figure 41: Application Command Workthread

entry-entry calls involves categorizing the interactions between tasks as a rendezvous, task phase change, or forwarding. An informal description will be given of how the entry-to-entry calls are estimated from the trace. The description of aggregating resource demands is then presented. The reader is referred to section 5.4 for a detailed description.

Before applying the algorithm of section 5.4 the workthreads are first normalized. This trace normalization changes the application scenario workthread as seen in Figure 37, into the normalized trace of Figure 41. Conversion into a workthread class can then begin.

The validation workthread of Figure 40 is nearly identical to the nested rendezvous example of Figure 15 on page 57.² A rendezvous interaction is recognized by a change in the executing task of the workthread, without the spawning of a child subtrace. For example in the first event of the validation workthread of Figure 40, the User (US) task is executing. In the second event the Communication Task (CM) is executing. This change in executing task means US has initiated a rendezvous with CM and is waiting for a reply. The change

2. The explanation in section 5.2.1 for the rendezvous and phase change detection also applies to the validation workthread and is reproduced here for completeness.

Figure 42: Normalized Application Command Workthread

in executing task of the third event indicates CM initiated a rendezvous with the Dispatcher (DP). In the fourth event CM is executing again, so, its rendezvous with DP is over. Then US executes in the last event, which means CM has completed the User's rendezvous. Each call from client to server in the workthread is mapped to a call from the client entry's current phase to the server entry. Following the form of the LQM diagrams, the resulting entry-entry calls for a validation scenario are shown in Figure 43.

Figure 43: Validation Command Workthread Class

The forwarding interaction is recognized as a (sub-)trace ending immediately after a **wSpawn** event type, and the child's subtrace starts execution (i.e., the second event in the subtrace) in a different task than initiated the spawn event (section 5.2.3). For example, in

Figure 41, the trace “AC” ends after a **wSpawn** event type is initiated by the Dispatcher (DP), and the spawned subtrace “AC.2” begins execution in task AP. This indicates a forward has occurred from the initiating task (DP), to the task that begins execution in the spawned subtrace (AP). A forward also occurs when the AP sends the a result to the DP in subtrace “AC.2.3”. Forwarding is characterized in the workthread class as a call from the blocked client task which is waiting for a reply (CM), to the task that has had the request forwarded to it (AP). The application scenario’s entry-to-entry calls are shown in Figure 41.

Figure 44: Application Command Workthread Class

A phase change is recognized as: 1) a **wSpawn** event type occurring in a trace, 2) the trace’s next event is executed by a different task, and 3) the spawned subtrace starts execution in the same task that initiated the spawn event. For example, in Figure 41, task

DT1 in subtrace “AC.2” spawns the subtrace “AC.2.1” which begins execution in task DT1. This indicates a phase change has occurred for the task that initiated the spawn (e.g., DT1). This also occurs for subtrace “AC.2.2” and task DT2. The resource demands of the spawned subtrace are assigned to the task entry’s second phase. For example, the resource demands of subtrace “AC.3.1” are assigned to the second phase of entry AC, in task DT1 (the first and second phase visits are separated by the comma in Figure 41).

Aggregating resource estimates for the workthread class is straightforward, as long as service times and visits are averages based on the number of times an entry is called (section 5.4.6). For example, in the application command workthread of Figure 41, the Dispatcher task will make four visits to the CPU with an aggregate service demand of 50,000 instructions. However, since the Dispatcher is called twice during the scenario, its entry’s service demand is averaged to 25,000 instructions per call.

The result of this analysis phase is the entry-entry calls shown in the figures, and the resource demand parameters summarized in Table 9. Table 9 has used the service rate of the resources to convert the service demands of the workthreads into service times. an LQM can now be constructed.

Task	Entry	Resource	Visits to Resource	Service Time / Visit (milli-sec)	Task	Entry	Resource	Visits to Resource	Service Time / Visit (milli-sec)
CM	VC	CPU2	2	3.5	AP	AC	CPU1	3	2.8
CM	AC	CPU2	2	3.5	DT1	AC	CPU1	4	2.2
DP	VC	CPU1	1	4.0	DT1	AC	Disk1	2	18.0
DP	AC	CPU1	2	2.5	DT2	AC	CPU2	5	2.5
NS	AC	CPU2	1	2.5	DT2	AC	Disk2	3	18.0

Table 9: OLTP Workthread Class Resource Demands

7.5 The OLTP Layered Queueing Model

Having developed the workthread classes, the performance model generation can be completed. The environment information for the model was provided in section 7.1. The workload intensity was given in Table 5, the resource characteristics in Table 6, and the

task information in Table 7. The model structure now needs to be assembled by combining these workthread classes into a *workthread class model*. The resource parameters are assigned to this model structure. In this case study, the workthread class model corresponds to an LQM that will be used to estimate the user response time.

The workthread class model characterizes the structure and resource demands of the workload (section 5.3). Its structure is considered to be the union of all tasks, and their entries, from each of the individual workthread classes in the workload. In this example, only the Dispatcher task has its entries for both workthread classes combined, since it is the only task used by both workthread classes. The communication tasks for each workthread class are modelled separately as specified by the task information, and distinguished by their entry name. Figure 45 illustrates the structure and the entry-to-entry calls of the workthread class model. The resource demands of the entries of the workthread class model are shown in Table 9, and the visits that each entry makes to a resource is found in Figure 45.

To convert the workthread class model into an LQM a simplification was made to combine the disk task phase 2 demands with its the phase 1 demands. This was done to accommodate the version of analytic LQM solver that was used.

The model was solved using an LQM solver [84] [41]. For the workload intensity of Table 5, the validation command response time is 0.018 seconds, the application command response time is 0.41 seconds, and the resource utilizations are shown in Table 10. A non-obvious result of the model evaluation is that the application task is constraining the throughput of the system because its utilization is 2.6 (with a maximum

Figure 45: OLTP Workthread Class Model and LQM

concurrency level of 3.0). This bottleneck could be eased by increasing the concurrency level, which is discussed in the next section.

Resource	Utilization
CPU1	46%
CPU2	46%
Disk1	40%
Disk2	59%

Table 10: OLTP Estimated Resource Utilizations

7.6 Experimenting with the OLTP Workthread Classes

Using the generated workthread classes of the OLTP system, some performance experiments can be conducted without returning to the prototyping environment. Several design alternatives are explored by:

- *Prototype behavioral changes:* To consider the impact of relocating data dependant services from one task to another the prototype is modified, traces are regenerated, and the workthread classes reproduced.
- *Altering the environment information:* To consider the impact of using a faster resource, the resource information is modified. Task information can be changed to consider alternate task placements across CPUs.
- *Workthread class structure modifications:* The structure of the workthread class is altered so that the resource demands for performing a service, are moved from one task to another. The workthread class structure can be modified in three fashions (section Chapter 6.0).

The first alternative is obvious and will not be considered further. An example of altering the environment information, specifically the concurrency level, is discussed next. Structural modifications are more intricate, and three examples will be considered.

The first experiment estimates the effect of varying the application task concurrency level by modifying and evaluating the model. The effect of altering the concurrency level on the response time is shown in Table 11. It suggests that increasing the concurrency level of the application task to at least four reduces the response time.

Application Task Concurrency Level	Application Command Response Time (sec)
2	1.30
3	0.41
4	0.27
5	0.25

Table 11: OLTP Application Task Concurrency and Response Time

The workthread class structure can be altered by moving an entry from one task, to a task at the same level in the client-server hierarchy (section 6.2). This is called horizontal entry movement and is shown in Figure 46. In this example, the validation entry of the Dispatcher (in Figure 45) has been moved to the new task "Validator". This moves the resource demands for the validation service from the Dispatcher to the new Validator task. The task information for this new task (Validator) would have to be provided.

Another structural modification is vertical entry movement, where a task's entry is moved to its client task. This is representative of moving the execution, and corresponding resource demands, for a service to the client task. For example, in Figure 46 the service provided by the Name Server has been moved to the Dispatcher. This is accomplished by incorporating the resource demands of the name server's entry for the "AC" scenario, into the Dispatcher's entry for the "AC" scenario. Since this results in fewer inter-task communications, the resource demands would be modified as well.

The last method of modifying the structure of a workthread class is to reassign resources, as shown in Figure 47. If two workthread classes only differ by the resource(s) they use, the second workthread class is created by cloning the first workthread class and reassigning the resource selection appropriately. For example, in the original workthread class of Figure 41, Disk Task 2 used the resource "Disk 2". In Figure 47, the resources are reassigned so Disk Task 2's entry "AC" now makes resource demands on Disk 1 instead of Disk 2. This change does not affect the entry-entry calls of the LQM.

7.7 Conclusions

The technique for generating a performance model has been presented, using an On-Line Transaction Processing system as the case study. The prototype illustrated the model generation technique and could have included more complex behavior, such as features that support fault tolerance, time-outs, transaction management, or audit information. Several performance experiments were discussed, involving changing the

Figure 46: LQM for Horizontal and Vertical Entry Movement

Figure 47: LQM Resource Demands with Reassignment

concurrency level and structural changes to the workthread class. The automated model building significantly reduced the effort to construct a model. The technique should easily scale to larger systems.

Chapter 8.0 Summary and Conclusion

“Wisdom is proved right by her children” Matthew 11:19

This chapter begins by summarizing the results of the research. It then discusses, in turn: general considerations about implementing angio tracing, use of the technique, and the characterization of resource functions. A discussion about future research ends the chapter.

8.1 Summary of Results

The following problem was posed in Chapter 1:

“Given an executable software design description, construct a Layered Queueing Network performance model, which includes structural information and data-dependent frequency parameters.”

The model generation technique presented in this thesis focuses on synchronous interactions between concurrent computer programs, with some limited interpretation of asynchronous interactions. Generalization seems possible and is discussed in the last subsection.

This work provides a technique that combines the empirical and predictive approaches to estimating performance model parameters by recording traces during prototype execution. The traces are then used to generate an analytic performance model. These traces are flexible enough to study a new software system’s performance behavior for alternative designs, system inputs, and environments.

The outline of the flow of information for the model generation technique is repeated in Figure 48. The technique begins once the prototype is operational. The technique has several steps, of which the first six are automated:

- Record angio traces for selected performance scenarios (steps #1 and #2).
- Reduce the traces to workthreads and workthread classes (steps #3 and #4).
- Construct and evaluate a performance model to produce performance estimates (steps #5 and #6).
- Compare the performance estimates with performance objectives and improve the design if necessary (steps #7).

The technique can be used with an operational, iterative software development process, as well as the more traditional waterfall process.

Figure 48: Model Generation Technique

Three novel concepts are used in this technique: the angio trace, the workthread, and the workthread class. The research contributions of this work are:

- *Definition of a form of tracing suitable for constructing performance models.* The angio trace and its event types can be used as a standardized input for automated model generation (section 4.2.7).
- *Automated transformations of traces to models.* An algorithm is presented in section 5.4 to perform this transformation on MLog angio traces. The algorithm detects rendezvous, forwarding, and phase change interactions.
- *Definition of a software system description for model experimentation.* The workthread class characterizes behavioral and structural information of a software system sufficiently for performance modeling purposes, allowing the analyst to manipulate the model structure in an intuitive manner for experimental purposes (Chapter 6.0).
- *Development of tools to support the technique.*
- *Demonstration of the technique (Chapter 7.0).*

Two other contributions are by-products of development of the technique. First, the technique uses a new workload characterization principle that adds to those given by Ferrari in [25]. Namely, software blocking relationships (here it is task blocking) are characterized, as well as the relative frequency of resource demands. To accomplish this, rendezvous and forwarding task interactions are detected, as well as a fork in control due to the reply of a rendezvous. This level of analysis is adequate for building Layered Queueing Models. Secondly, angio tracing extends the causal trace of Helmbold, McDowell, and Wang [35] to include asynchronous activity.

The case study that was presented in Chapter 7.0 considered an On-Line Transaction Processing (*OLTP*) system which serviced user commands. To estimate the response time of a user's command, a LQM of the system was automatically generated

using the technique illustrated in Figure 48. Using the LQM performance estimates a system bottleneck was detected and corrective measures were suggested. Once the initial workthread classes were constructed, examples of modifying their structure to produce alternative models were considered.

This research can easily be adapted to different design methodologies (e.g. Object-Oriented Analysis and Design), design descriptions (e.g. ObjecTime, MLog), software lifecycle phases (e.g. reengineering of legacy systems), and application areas (e.g. databases, real-time systems). Using traces as a standardized input to the model building, a single performance engineering approach can be applied with any software development methodology. The traces provide independence from the language used to express a software system design. For example, an alternative source of angio traces could be embellished Use-Case [40] investigations. Provided the traces can be generated, the technique can be used during any phase of a system's lifecycle including analysis, design, testing, operation, and reengineering. By generating traces and using an appropriate performance model, the technique can be adapted to virtually any kind of software.

8.2 Implementation Considerations

This subsection discusses implementation considerations gleaned from the MLog implementation of angio tracing, which are relevant for other prototyping environments. A description of the recent implementation of angio tracing for the ObjecTime prototyping environment ends the subsection.

Although conceptually simple, instrumenting the MLog statements proved to be tricky. Several times, unexpected interactions between an incorrect ordering of primitive instrumentation probes (section 4.3.1) and the execution point where the angio event is recorded to a file, resulted in an event being recorded with an incorrect subtrace name. This problem was extremely difficult to debug because it can only be detected after prototype

execution by examining the traces. A set of simple MLog prototype models, with corresponding traces, were used to test the instrumentation and detect these problems.

A prototype environment that is suitable for angio tracing must have several characteristics. The first characteristic is that the prototype run-time (e.g. task scheduler) must be able to manage the data for the dyes. Secondly, statements for indicating the beginning and ending of a scenario are needed. Lastly, backtracking (as allowed for in MLog) should only be used if it is part of the system being modelled (section 4.3.3). As angio tracing is implemented in other prototyping environments more constraints may become evident.

The characteristics of MLog that simplified angio tracing implementation were:

- Separate statements to distinguish a rendezvous initiation, rendezvous reply, and asynchronous communication. If only asynchronous communication is available, the number of subtraces spawned easily proliferates.
- Once a task begins execution of a service request, it cannot be interrupted by a higher priority request. Otherwise, the prototype run-time must manage the dye contexts during the pre-emption of a service request.
- Inter-task communication only uses message passing, which simplifies instrumentation. Extending angio tracing for other communication mechanisms (e.g. shared memory) should be possible, but the instrumentation and trace post-processing would be more complicated.
- The over-riding of system statements can be done to instrument the system operations. Then there is minimal impact to normal prototype use since the angio instrumentation would only be invoked when tracing is being performed.

A side-benefit of angio tracing was that the traces provided application contextual information that was useful during the debugging of the MLog prototype. This was

especially useful when isolating the cause of unexpected MLog backtracking during prototype execution.

Recently, angio tracing has been implemented for the ObjecTime [58] prototyping environment. The initial development was easily accomplished over a couple of weeks due to the message passing and object-oriented nature of ObjecTime. The ObjecTime implementations differs from the MLog implementation is several ways:

- The angio events are kept in memory to allow for Message Sequence Charts [18] to be constructed. This also allows some event information to be suppressed until the traces are written to a file (e.g. **wBegin**, **wEnd**).
- The angio events record the state machine transition name and logical clock time associated for each angio events.

Future research on model generation will use the ObjecTime prototyping environment.

8.3 Using the Technique

This subsection provides a starting point for using the technique, which can be expanded as experience is gained. A description of the necessary information is provided. Scenarios which interact are then discussed. This subsection ends with comments on knowing when a model has enough detail.

Aside from a prototyping environment which supports angio tracing, there are several preconditions to be satisfied before the technique can be used. Each of these items is listed below:

- Performance experiments and the associated scenarios must be specified. A requirements document usually provides the basis for performance experiment and scenario identification. Each performance experiment must provide the workload intensity of the scenarios and the performance goals for them. If a scenario is composed of several stimuli (e.g. if several user commands form a transaction), then a trace and corresponding workthread class is generated for each stimulus.

The scenario workthread class is constructed by clustering the workthread classes of several stimuli (section 6.1).

- The instrumented prototype, should have software structure and behavior similar to the final implementation. Even if the prototype does not have structural information, then resource demands for the scenario could still be generated. Typically, a prototype is used to verify the functional correctness of a design. After this step, the prototype can be used for performance analysis purposes.
- The implementation's planned configuration should be available. This includes resource characteristics (e.g. device scheduling discipline), task information (e.g. task population), and resource allocation.
- Resource functions to characterize the resource demands are needed. This is discussed in the next subsection.

Data management tools are also needed to aid the analysis of the potentially large volume of information that can generated.

The technique assumes the independence of execution of separate scenarios, which may not be true. Cases where a scenario's execution behavior, resource selection, and resource demand are not independent of the execution or load imposed by other scenarios are possible. More research is needed in this area.

A concern, when modelling, is understanding when the model has sufficient detail. This concern is mitigated because the technique allows the performance model to be transparently refined, provided it is used in conjunction with an evolutionary prototyping approach (stepwise refinement from abstract models to detailed realistic products [56]), or an incremental prototyping approach (the system is based on one overall design but built incrementally [34]). Intuitively, iteration of the analysis process can be used to converge upon a sufficiently detailed prototype. The iteration begins once an initial set of performance experiments have been performed, and would proceed as follows:

- 1) Using the performance predictions, identify resources which are system bottlenecks.
- 2) Using the workload classes, identify the scenarios which use resources that are bottlenecks.
- 3) Add additional detail for these scenarios which make use of the bottlenecks. This may involve adding more detail to the prototype, environment information, or resource functions.
- 4) Regenerate and reevaluate the performance models. If the performance predictions agree with those generated in the previous iteration, then there is likely little to be gained by adding detail. Otherwise, continue the process.

It should be noted that such an iterative analysis is possible because of the automation provided by the technique.

8.4 Resource Function Characterization

The characterization of a resource function is important for the model generation technique. Three approaches to the characterization and use of resource functions were described in section 5.1. They are:

- i) *A complexity measure.* Event parameters appear in the trace and may be used as arguments for a complexity measure that estimates the resource demands [81]. For example, the USL C++ Standard Components [80] provide time-complexity measures for the execution of the algorithms for array access, set manipulation operations, etc. The resource function parameters are limited to prototype statement parameters and accessible system state information.
- ii) *A distribution function.* The mean value of the distribution is used as the resource demand. This approach can be taken if measurements from a comparable system are available and can be utilized.

iii) *Demand recorded at run-time.* No resource function is provided as the resource selection and demands are bound during prototype execution. The resource demand is also limited to prototype statement parameters and accessible system state information.

The gathering of resource functions represents a cost to the development of the model for types i) and ii). This cost can be amortized if a database of resource functions is maintained, so that reuse of resource functions is possible. Reuse of type iii) resource functions is possible through prototype code reuse.

Future research is needed to provide guidance for developing resource functions.

8.5 Future Research

Scenario independence and resource function characterization have already been identified as areas for future research. Two other streams of research which have not previously been discussed are presented in this subsection. The first stream is to extend the technique to automatically generate performance models other than a Layered Queueing Network. The second thrust is to use the technique to enhance existing software performance engineering approaches. Some further comments follow.

This research focussed, in general, on constructing analytic models for systems with a single thread of control. To construct more complex performance models, the angio trace needs to be extended for recording more exotic system behavior. Examples of such behavior are task synchronization or the initiation of asynchronous execution by a task. Angio trace recording of logical resource management (e.g. semaphores, database locks) would provide further detail for more complete performance models. For this purpose, the angio trace will need to include new event types and extend the naming convention. Additions to the trace analysis capability will also be required to map these more complicated traces to a performance model.

Aside from the automated generation of performance models, other aspects of the technique can be used to enhance Software Performance Engineering (SPE). SPE approaches make use of two or three different models (section 2.1): an implementation or software system description, a software execution model, and a performance model. There is little coupling between these three models that can be taken advantage of by current SPE approaches. This technique produces by-products that can bridge the gaps among the three models, and used to enhance other aspects of SPE. For example, the traces can be used to map performance predictions back to the components of the software system, to the degree that they are identified in the trace.

References

- [1] M. Alford. "SREM at the age of eight; the distributed computing design system." *Computer*, 18(4):36–46, April 1985.
- [2] M. Alford. "Software Requirements Engineering Methodology (SREM) at the age of eleven – requirements driven design." In P. Ng and R. Yeh, editors, *Modern Software Engineering: Foundations and Current Perspectives*, pages 352–377. Van Nostrand Reinhold, New York, NY, 1990.
- [3] R. A. Ammar and C. P. Rosienc. "Visualizing a hierarchy of performance models for software systems." *Software - Practice and Experience*, 23(3):293–315, March 1993.
- [4] G. E. Anderson. "The coordinated use of five performance evaluation methodologies." *Communications of the A.C.M.*, 27(2):119–125, February 1984.
- [5] J. Baker, D. Chester, and R. Yeh. "Software design by stepwise evaluation and refinement." In P. Ng and R. Yeh, editors, *Modern Software Engineering: Foundations and Current Perspectives*, pages 236–273. Van Nostrand Reinhold, New York, NY, 1990.
- [6] R. Balzer. "Draft report on requirements for a common prototyping system." *ACM SIGPLAN Notices*, 24(3):93–165, March 1989.
- [7] J. Bartlett, J. Gray, and B. Horst. "Fault tolerance in Tandem Computer Systems." In A. Avizienis, P. Kopetz, and J. Laprie, editors, *Dependable Computing and Fault-Tolerant Systems*, pages 55–77. Springer-Verlag, New York, New York, 1987.
- [8] P. C. Bates. *Debugging Programs in a Distributed System Environment*. PhD thesis, University of Massachusetts, 1986.
- [9] F. Belina, D. Hogrefe, and A. Sarma. *SDL with Applications from Protocol*

- Specification*. Prentice Hall International, Hertfordshire, UK, 1991.
- [10] V. Berzins and Luqi. "Rapidly prototyping real time systems." *IEEE Software*, pages 25–35, Sept. 1988.
- [11] A. D. Birrell and B. J. Nelson. "Implementing remote procedure calls." *ACM Transactions on Computer Systems*, 2(1):39–59, Feb. 1984.
- [12] A. Borr and F. Putzolu. "High performance SQL through low-level system integration." *Proceedings of International Conference on Management of Data (SIGMOD)*, pages 342–349, 1988.
- [13] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Inc., Englewood Cliffs, N.J. 07632, 1978.
- [14] R. Budde, K. Kautz, K. Kuhlenkamp, and H. Zullighoven. *Prototyping: An Approach to Evolutionary System Development*. Springer-Verlag, New York, New York, 1992.
- [15] R. J. Buhr and R. S. Casselman. "Architectures with pictures." In *OOPSLA'92*, volume 27, pages 466–483, 1992.
- [16] J. Caillet, C. Bonnet, and B. Raither. "High level interpretation of execution traces of Ada tasks." In H. Nichols and D. Simpson, editors, *ESEC '87: 1st European Software engineering Conference*, pages 309–317, New York, New York, 1987. Springer-Verlag.
- [17] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, UK, Ltd., London, England, 1985.
- [18] CCITT. "Recommendation Z.120: Message Sequence Chart (MSC)." Technical report, Geneva.
- [19] C.M. Woodside, 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 & Computer Engineering and School of Computer Science, Carleton University, March 1989.
- [20] Curtis E. Hrischuk and C.M. Woodside. "Prototyping for performance measures." Report SCE-93-16, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada K1S 5B6, January 1994.
- [21] D.L. Parnas and P.C. Clements. "A rational design process: How and why to fake it." *IEEE Transactions on Software Engineering*, 12(2):251–257, 1986.
- [22] T. Dodd. *Prolog: A Logical Approach*. Oxford University Press, New York, New York, 1990.
- [23] E.D. Lazowska, J. Zahorjan, G.S. Graham, and K.C. Sevcik. *Quantitative System Performance*. Prentice-Hall, Inc., Englewood Cliffs, N.J. 07632, 1984.
- [24] D. Ferrari. *Computer Systems Performance Evaluation*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1978.
- [25] D. Ferrari. "A performance-oriented procedure for modeling interactive workloads." In D. Ferrari and M. Spadoni, editors, *Experimental Computer Performance Evaluation*, pages 57–78. North Holland, 1981.
- [26] N. D. Francesco and G. Vaglini. "Description of a tool for specifying and prototyping concurrent programs." *IEEE Trans. on Soft. Eng.*, pages 1554–1564, Nov. 1988.
- [27] D. Garlan and M. Shaw. "An introduction to software architecture." In *Advances in Software Engineering and Knowledge Engineering*, volume I. World Scientific Publishing Company, 1993.
- [28] G.M. Karam. "MLog: a language for prototyping concurrent system designs," September 1989. 19 pages.
- [29] G.M. Karam. "The MLog user's guide." SCE-89-15, Dept. of Syst. and Comp. Engineering, Carleton University, April 1992. 78 pages.

- [30] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1993.
- [31] D. Harel. "On visual formalisms." *Communications of the ACM*, 31(5):514–530, May 1988.
- [32] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and A. Shtut-Trauring. "Statemate: A working environment for the development of complex reactive systems." In *10th International Conference on Software Engineering*, pages 241–250, 1988.
- [33] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtut-Trauring, and M. Trakhtenbrot. "Statemate: A working environment for the development of complex reactive systems." *IEEE Trans. on Soft. Eng.*, 16(4):403–414, April 1990.
- [34] S. Hekmatpour and S. Ince. *Software Prototyping, Formal Methods and VDM*. Addison-Wesley Publishing Co., New York, New York, 1988.
- [35] D. P. Helmbold, C. E. McDowell, and J.-Z. Wang. "Determining possible event orders by analyzing sequential traces." *IEEE Transactions on Parallel and Distributed Systems*, 4(7):827–839, July 1993.
- [36] G. Hills, J. Rolia, and G. Serazzi. "Performance engineering of software architectures for ODP systems." 1995. Submitted to the 3rd International Conference on Open Distributed Processing.
- [37] i-Logix Inc., Burlington, MA. *STATEMATE Analyser Manual - version 3.0*, 1990.
- [38] i-Logix Inc., Burlington, MA. *STATEMATE Kernel - X-Window System 11 - version 3.0*, 1990.
- [39] I.E.E.E. *Threads Extension for Portable Operating Systems*, posix p1003.4a edition, 1990.

- [40] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-Oriented Software Engineering*. Addison-Wesley Publishing, New York, 1992.
- [41] J.A. Rolia. *Predicting the Performance of Software Systems*. PhD thesis, University of Toronto, January 1992.
- [42] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, New York, New York, 1991.
- [43] J.E. Neilson, C.M. Woodside, D.C. Petriu, and S. Majumdar. "Software bottlenecking in client-server systems and rendezvous networks." Report SCE-92-17, Real-Time and Distributed Systems Group, Carleton University, Ottawa, Canada K1S 5B6, November 1992.
- [44] Jean-Paul Tremblay and Paul G. Sorenson. *An Introduction to Data Structures with Applications*. McGraw-Hill Publishing Company, New York, 1984. second edition.
- [45] K. Mani Chandy and Doug Neuse. "Linearizer: A heuristic algorithm for queueing network models of computing systems." *CACM*, 25(2):126–134, February 1982.
- [46] S. Khoshafian, A. Chan, A. Wong, and H. K. Wong. *Client/Server SQL Applications*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1993.
- [47] D. Kourie. "The design and use of a Prolog trace generator for CSP." *Software Practice and Experience*, 17(7):423–438, July 1987.
- [48] J. R. Larus. "Abstract execution: A technique for efficiently tracing programs." *Software - Practice and Experience*, 20(12):1241–1258, December 1990.
- [49] J. R. Larus. "Efficient program tracing." *I.E.E.E. Computer*, 26(5):52–61, May 1993.
- [50] C. H. Leung. *Quantitative Analysis of Computer Systems*. John Wiley and Sons, New York, New York, 1988.
- [51] Luqi. "Computer aided prototyping for a command and control system using CAPS."

- IEEE Software*, pages 56–67, Jan. 1992.
- [52] Luqi. “Real time constraints in a rapid prototyping language.” *Journal of Computer Languages*, 18(2):77–103, 1993.
- [53] Luqi, V. Berzins, and R. Yeh. “A prototyping language for real time software.” *IEEE Trans. on Soft. Eng.*, 14(10):1409–1423, Oct. 1988.
- [54] Luqi and M. Ketabchi. “A computer-aided prototyping system.” *IEEE Software*, pages 66–72, March 1988.
- [55] Luqi and W. Royce. “Software evolution through rapid prototyping.” *IEEE Computer*, pages 13–25, May 1989.
- [56] Luqi and W. Royce. “Status report: Computer aided prototyping.” *IEEE Software*, pages 77–81, Nov. 1991.
- [57] A. Mazurkiewicz. “Trace theory.” In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency*, Lecture Notes in Computer Science: 255, pages 279–324. Springer-Verlag, New York, New York, Sept. 1986.
- [58] ObjecTime Limited, Kanata, Ontario, Canada. *ObjecTime User’s Manual*, 1994.
- [59] D. M. Ogle, K. Schwan, and R. Snodgrass. “Application-dependent dynamic monitoring of distributed and parallel systems.” *IEEE Transactions on Parallel and Distributed Systems*, 4(7):762–778, July 1993.
- [60] P.A. Jacobson and E.D. Lazowska. “Analyzing queueing networks with simultaneous resource possession.” *Communications of the ACM*, 25(2):142–151, February 1982.
- [61] D. C. Petriu, S. Majumdar, J. Lin, and C. E. Hrischuk. “Analytic performance estimation of client-server systems with multi-threaded clients.” In *MASCOTS 94*, pages 96–100, 1994.

- [62] C. Potts, A. Bartlett, B. Cherrie, and R. MacLean. "Discrete event simulation as a means of validating JSD design specifications." In *8th Int. Conf. on Software Engineering*, pages 119–125, London, UK, Aug. 1985.
- [63] M. Rem. "Trace theory and the design of concurrent computations." In J. van Leeuwen and L. Lenstra, editors, *Parallel Computers and Computations*, volume 9 of *CWI Syllabus*, pages 115–132. Center for Mathematics and Computer Science, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands, 1983.
- [64] R.J.A. Buhr, G.M. Karam, C.M. Woodside, R. Casselman, R.G. Franks, H.J. Scott, and D. Bailey. "TimeBench: a CAD tool for real-time system design." In *Proceedings of the 2nd International Symposium on Environments and Tools for Ada SETA2*, Washington, D.C., January 1991.
- [65] J. Rolia and K. Sevcik. "Fast performance estimates for a class of generalized stochastic petri-nets." In *Computer Performance Evaluation '92: Modelling Techniques and Tools*, pages 21–33. Edinburgh University Press, August 1993.
- [66] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Inc., Englewood Cliffs, NJ 07632 USA, 1991.
- [67] P. Scheffer, A. Stone, III, and W. Rzepka. "A case study of SREM." *Computer*, 18(4):47–54, April 1985.
- [68] P. Schweitzer. "Approximate analysis of multiclass closed networks of queues." In *Proceedings of the International Conference on Stochastic Control and Optimization*, Amsterdam, 1979.
- [69] P. J. Schweitzer and I. F. Akyldiz. "Approximate mean value analysis for non-product form networks of queues with job class changes." In G. Balbo and G. Serazzi, editors, *Computer Performance Evaluation*, pages 61–72. Elsevier Science Publishers, 1992.
- [70] Scientific and Engineering Software, Inc., 4301 Westbank Drive, Austin, Texas,

- U.S.A. *SES/workbench User's Manual - Release 2.1*, Feb. 1992.
- [71] R. Sedgewick. *Algorithms in C++*. Addison-Wesley Publishing Company, New York, 1992.
- [72] B. Selic, G. Gullekson, and P. Ward. *Real-time Object-oriented Modeling*. John Wiley and Sons, 1994.
- [73] A. Sinha. "Client-server computing." *Communications of the ACM*, 35(7):77–98, July 1992.
- [74] C. Smith. *Performance Engineering of Software Systems*. Addison-Wesley Publishing Co., New York, NY, 1990.
- [75] C. U. Smith and L. G. Williams. "Software performance engineering: A case study including performance comparison with design alternatives." *IEEE Transactions on Software Engineering*, 19(7):720–741, July 1993.
- [76] S.S. Lavenberg. "A perspective on queueing models of computer performance." *Performance Evaluation*, 10:53–76, 1989.
- [77] L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Massachusetts, 2 edition, 1986.
- [78] Tandem Computers Inc. "Nonstop SQL optimizer: Basic concepts." In *Tandem Systems Review*, pages 14–21. Tandem Computers Inc., July 1988.
- [79] H. Touati and A. J. Smith. "Reducing and manipulating complex trace data." *Software - Practice and Experience*, 21(6):639–655, June 1991.
- [80] Unix System Laboratories, Inc., 190 River Road. *USL C++ Standard Componentets Release 3.0 Programmer's Guide*, 1992.
- [81] V. Vetland. *Measurement-Based Composite Computational Work Modelling of Software*. PhD thesis, University of Trondheim, August 1993.

- [82] R. Vonk. *Prototyping: The effective use of CASE technology*. Prentice-Hall, London, UK, 1990.
- [83] W. Wayt Gibbs. "Software's chronic crisis." *Scientific American*, pages 86–95, September 1994.
- [84] C. M. Woodside, J. E. Neilson, D. C. Petriu, and S. Majumdar. "The stochastic rendezvous network model for performance of synchronous multi-tasking distributed software." *IEEE Trans. Softw. Eng.*, 1994. Accepted for publication.
- [85] N. Xenios. *The Structure and Performance Specification Tool user Guide of the stand-alone version*. BNR Europe Ltd., v1.0 edition, October 1991.
- [86] P. Zave. "An operational approach to requirements specification for embedded systems." *IEEE Trans. on Soft. Eng.*, SE-8(3):250–269, May 1982.
- [87] P. Zave. "The operational versus the conventional approach to software development." *Comm. of the ACM*, 27(2):104–118, Feb. 1984.
- [88] P. Zave. "An insider's evaluation of PAISley." *IEEE Trans. on Soft. Eng.*, pages 212–225, March 1991.
- [89] P. Zave and W. Schell. "Salient features of an executable specification language and its environment." *IEEE Trans. on Soft. Eng.*, pages 312–325, Feb. 1986.

END

3 1 - 0 5 - 9 5

FIN