Performance Engineering during Information System Development\textsuperscript{1}

Andreas L. Opdahl
Information Systems Group
Faculty of Computer Science and Electrical Engineering
The Norwegian Institute of Technology
The University of Trondheim

November 17, 1992

\textsuperscript{1}This version was reformatted on January 5, 2000. The page numbers do not match those of the original thesis.
Abstract

Performance engineering of information systems aims at predicting and improving the performance of projected applications during development. The need for performance engineering of information systems is pointed out and discussed. The thesis also presents the state of the art in the related fields of information system engineering, computer performance evaluation, and software performance engineering. Requirements for successful performance engineering of information systems are pointed out. In particular, performance engineering must be closely integrated with the information system engineering process.

A basic framework for performance prediction is presented. The framework is based on separate models of computer hardware, existing and projected software applications, and the organisation using the applications. In particular, workload modelling of projected applications is focused on. Workload models for projected applications are derived during development from design specifications annotated with performance parameters. The framework is extended with support for capturing these performance parameters. Sensitivity analysis is provided for identification of important design parameters, and for pointing out where in the design specification optimisation effort should be focused. A performance prediction technique for distributed information systems is also provided.

The basic framework has been implemented in a CASE tool. The framework and the tool have been used in a practical case study. The conclusion points out that further work is particularly needed in the areas of practical case-studies, tool support, database performance prediction, software synchronisation and blocking, as well as distributed systems.
Contents

Abstract i

Contents iii

Preface xi

Terminology and Notation xi
Credits to Adapted Figures and Tables xii
Acknowledgements xii

I Introduction and Motivation 1

1 Introduction 5

1.1 Problem Statement 5

1.2 Approach 5

1.3 Major Results Achieved 7

1.4 Related Doctoral Works 8

2 The Need for Performance Engineering of Information Systems 11

2.1 Information Systems and Computers 11

2.2 Informatics and Computer Science 12

2.3 The Importance of Information System Performance 14

2.4 The Insufficiency of Contemporary Practice 18
II State of the Art and Requirements

3 Information System Engineering

3.1 Software Engineering .................................................. 28
  3.1.1 The waterfall paradigm ........................................... 28
  3.1.2 Process and data modelling .................................... 29
  3.1.3 Integrated modelling approaches ............................... 29
3.2 Information System Engineering ..................................... 30
  3.2.1 Information system engineering methods ....................... 30
  3.2.2 Information system modelling .................................. 31
3.3 The PPP Environment ................................................. 34
  3.3.1 A PPP overview .................................................. 35
  3.3.2 A PrM overview .................................................. 37

4 Computer Performance Evaluation .................................... 47

4.1 Performance Measurement ............................................ 48
  4.1.1 Measurement instrumentation .................................... 48
  4.1.2 Measurement interpretation ..................................... 49
4.2 Workload Modelling .................................................. 50
  4.2.1 Workload characterisation ...................................... 50
  4.2.2 Workload analysis ............................................... 51
4.3 Performance Modelling .............................................. 52
  4.3.1 Simulation models ............................................... 52
  4.3.2 Analytic performance modelling ............................... 53
4.4 Queueing Network Modelling ....................................... 54
  4.4.1 Queueing networks ............................................... 55
  4.4.2 Operational queueing network analysis ....................... 57
  4.4.3 Bounds analysis ................................................ 61
4.5 Petri-Net Modelling ................................................ 61
  4.5.1 Petri-nets ...................................................... 62
  4.5.2 Generalised stochastic Petri-nets ............................. 64
  4.5.3 Generalised stochastic Petri-net analysis .................... 68
# Contents

4.6 Other Performance Modelling Techniques ........................................ 69
  4.6.1 Performance model aggregation ............................................. 69
  4.6.2 Performance model sensitivity analysis ................................... 72
4.7 Performance Modelling Tools ...................................................... 75
4.8 The IMSE Environment .................................................................. 78
  4.8.1 An IMSE overview ................................................................. 79
  4.8.2 An sp overview ..................................................................... 81

5 Performance Engineering of Software Systems .................................. 85
  5.1 Software Performance Engineering ............................................... 85
  5.2 Software Performance Engineering Methods ................................ 88
  5.3 Software Execution Graphs ......................................................... 88
  5.4 Computation Structures ............................................................. 92
  5.5 Database Transaction Design ....................................................... 94
  5.6 Software Performance Engineering Tools ..................................... 98

6 Requirements for Performance Engineering of Information Systems .... 99
  6.1 Problems with Software Performance Engineering ....................... 99
  6.2 Cost-Efficiency ......................................................................... 101
  6.3 Integration with Information System Engineering ......................... 102
  6.4 Integration with Capacity Management ....................................... 103
  6.5 Parameter Capture Support ......................................................... 105

III The Framework ............................................................................ 109

7 Baseline ......................................................................................... 113
  7.1 View of Information Systems ....................................................... 113
  7.2 View of Design Specifications ..................................................... 115
  7.3 View of Performance Modelling .................................................. 117
  7.4 Towards an Integrated View ........................................................ 121

8 Performance Prediction .................................................................. 125
8.1 Performance Parameters .................................... 125
  8.1.1 Entry matrices ........................................ 127
  8.1.2 Activity matrices ....................................... 128
  8.1.3 Visit matrices for primitive components .......... 129
8.2 Workload Derivation ....................................... 130
  8.2.1 Internal visit matrices for components .......... 131
  8.2.2 Complexity matrices .................................. 132
  8.2.3 Visit matrices for non-primitive components .. 133
8.3 Workload and Performance Models ....................... 134
  8.3.1 Workload scenarios ................................... 135
  8.3.2 The projected workload module ..................... 137
  8.3.3 The existing workload modules ..................... 138
  8.3.4 The workload model .................................. 138
  8.3.5 The performance model ............................... 138
  8.3.6 Performance measures ............................... 141
  8.3.7 Performance analysis ............................... 142
8.4 Performance Prediction Overview ......................... 143

9 Parameter Capture Support .................................. 145
  9.1 Parameter Validation ..................................... 146
    9.1.1 Internal parameter validation .................... 146
    9.1.2 External parameter validation .................... 147
  9.2 Residence Time Analysis ................................ 148
    9.2.1 The need for residence time analysis ............ 149
    9.2.2 Component layers and chains ..................... 149
    9.2.3 Sojourn times of layer operations ............... 151
    9.2.4 Complexity matrices under specifications ...... 152
    9.2.5 Per operation residence time analysis ........... 154
    9.2.6 Average residence time analysis ................. 155
    9.2.7 Use of residence time analysis ................... 157
  9.3 Software Platform Modelling ............................ 158
9.3.1 The need for software platform modelling .................................. 159
9.3.2 The software platform model ....................................................... 159
9.3.3 Composite workload model analysis ............................................. 161
9.3.4 Parameter estimation support ...................................................... 162
9.3.5 Database optimisation ............................................................... 162
9.3.6 Target platform synchronisation and blocking .................................. 164
9.3.7 Use of target platform modelling .................................................. 164

9.4 Bounds Analysis ................................................................. 165
9.4.1 The need for bounds analysis ...................................................... 165
9.4.2 Performance parameters ........................................................... 166
9.4.3 Workload derivation ................................................................. 169
9.4.4 Workload and performance models ............................................. 172

9.5 Parametric Analysis ............................................................ 175
9.5.1 The need for parametric analysis ................................................ 175
9.5.2 Performance parameters ........................................................... 176
9.5.3 Workload derivation ................................................................. 176
9.5.4 Software platform modelling ...................................................... 177
9.5.5 Workload and performance models ............................................. 178

10 Sensitivity Analysis ......................................................... 181
10.1 The Need for Sensitivity Analysis ............................................... 181
10.2 Sensitivity Analysis Overview .................................................... 182
10.3 Performance Model Sensitivity Analysis ........................................ 183
  10.3.1 Open queueing networks ....................................................... 184
  10.3.2 Closed queueing networks ..................................................... 185
  10.3.3 Mixed queueing networks ...................................................... 186
  10.3.4 Product-form queueing network sensitivity analysis ..................... 186
10.4 Design Specification Sensitivity Analysis ...................................... 187
10.5 Average Sensitivity Analysis ..................................................... 190
10.6 Validation ............................................................................ 191
10.7 Use of Sensitivity Analysis ....................................................... 192
11 Distributed Information Systems

11.1 The Need for Performance Engineering of Distributed Information Systems  193
11.2 Overview of Distributed Information System Analysis  195
11.3 Distributed Performance Submodel Aggregation  196
  11.3.1 Distributed hardware platforms  196
  11.3.2 Primitive subspecifications  199
  11.3.3 Performance submodel aggregation  200
11.4 Top-Level Subspecification Analysis  204
  11.4.1 Non-primitive subspecifications  204
  11.4.2 The hierarchy of subspecifications  205
  11.4.3 PrM annotations  207
  11.4.4 GSPN generation  208
  11.4.5 GSPN annotations  211
  11.4.6 GSPN analysis  212
11.5 Intermediate Subspecification Aggregation  214
11.6 Software Synchronisation and Blocking  215

IV A Realisation of the Framework  217

12 Integrating the Basic Framework with the PrM CASE Tool  221

12.1 The PrM Tool  221
12.2 Annotating PrM Specifications  222
  12.2.1 Organisation specification  223
  12.2.2 Execution characterisation  228
  12.2.3 Demand description  230
12.3 Interfacing the Basic Framework  233
  12.3.1 Deriving entry matrices  233
  12.3.2 Deriving activity matrices  233
  12.3.3 Deriving visit matrices  236
  12.3.4 Interfacing the IMSE  236

13 Integrating the Framework Extensions with the PrM CASE Tool  241
13.1 Parameter Validation .............................................. 241
13.2 Residence Time Analysis ........................................ 242
  13.2.1 Analysis initiation ........................................ 242
  13.2.2 Analysis outputs ........................................ 243
13.3 Software Platform Modelling .................................... 243
  13.3.1 Analysis initiation ........................................ 244
  13.3.2 Analysis outputs ........................................ 244
  13.3.3 Tool support ............................................. 244
13.4 Bounds Analysis .................................................. 245
  13.4.1 Analysis initiation ........................................ 246
  13.4.2 Analysis outputs ........................................ 246
  13.4.3 Tool support ............................................. 246
13.5 Parametric Analysis ............................................. 247
  13.5.1 Analysis initiation ........................................ 248
  13.5.2 Analysis outputs ........................................ 248
  13.5.3 Tool support ............................................. 248
13.6 Interfacing Sensitivity Analysis with PrM ....................... 249
  13.6.1 Analysis initiation ........................................ 250
  13.6.2 Analysis outputs ........................................ 250
13.7 Interfacing Distributed Analysis with PrM ....................... 251
  13.7.1 Analysis Initiation ........................................ 251
  13.7.2 Analysis outputs ........................................ 252
  13.7.3 Tool support ............................................. 252

14 Conclusions and Further Work .................................... 255
  14.1 Achievements .................................................. 255
  14.2 Requirements Met ............................................ 256
  14.3 Further Work .................................................. 260
    14.3.1 Basic framework ....................................... 260
    14.3.2 Parameter capture support ............................. 260
    14.3.3 Sensitivity analysis .................................... 261
14.3.4 Distributed information systems ............................. 261
14.3.5 The PrM tool ................................................. 261

Appendices 265

A List of Contributions 265
A.1 Statement of Originality and Sources .......................... 265
A.2 Implementation and Case Study ................................ 267

B Operational Queueing Network Analysis 269
B.1 Open Queueing Network Analysis ............................... 269
B.2 Closed Queueing Network Analysis ............................. 271
B.3 Mixed Queueing Network Analysis .............................. 274

C Queueing Network Bounds Analysis 277
C.1 Open Queueing Networks ....................................... 277
C.2 Closed Queueing Networks ..................................... 278
C.3 Mixed Queueing Networks ...................................... 279

D Sensitivity Analysis 281
D.1 Performance Model Sensitivities with Respect to Service Times ................................. 281
D.2 Performance Model Sensitivities with Respect to Think Times ................................. 282
D.3 Design Specification Sensitivities for Non-primitive Components ............................ 283
D.3.1 Analysis with respect to entry counts ............................. 284
D.3.2 Analysis with respect to activity counts ............................. 284

List of Figures and Tables 284

List of Tables 288

Bibliography 290
Preface

This thesis has been written at the University of Trondheim from the spring of 1988 to the autumn of 1992. Most of the work was carried out at the Norwegian Institute of Technology where I was a scientific assistant in 1988 and became a university scholarship holder from 1989 throughout 1991. In the spring of 1992 I became Assistant Professor of Informatics at the College of Arts and Sciences, and in the autumn I moved to a similar position at the Department of Information Science at the University of Bergen.

My doctoral work has been supervised by Professors Arne Solvberg and Peter Hughes in connection with the RHAPSODY and IMSE\(^1\) projects.

Terminology and Notation

In the introduction and motivation part I — as well as in the state-of-art part II of the thesis — terminology and notation will be informally introduced whenever needed. A precise terminology and notation for parts III and IV will then be introduced in chapter 7. New concepts will be \textbf{boldfaced} when first introduced or defined. The exception is when related work is presented in the state-of-art part II. These presentations have mostly been kept in the authors’ own terminologies. Some terms have been “quoted” to avoid conflicts with defined terms.

Simple variables will be denoted with lower-case roman letters \(x\). Vectors of simple variables will be denoted by upper-case roman letters \(X\), while matrices will be denoted by bold-fonted upper-case letters \(X\). More complex structures, such as hierarchies and layers in hierarchies will usually be denoted with calligraphic letters \(\mathcal{X}\). When there is no danger of confusion, upper-case roman letters will also be used to denote sets and lists. The use of greek letters has been kept at a minimum. However, for lower-case greek letters \(\tau\) for which no upper-case letter exists, the corresponding vectors will be denoted \(\tau\).

\(^1\)The IMSE project was a collaborative research project supported by the CEC as ESPRIT project no 2143. It was carried out by the following organisations: BNR Europe STI, Thomson CSF, Simulog A.S., University of Edinburgh, INRIA, IPK (Berlin), University of Dortmund, University of Pavia, SINTEF (University of Trondheim), University of Turin and University of Milan.
Credits to Adapted Figures and Tables

Some of the figures and tables have been adapted from other sources: Fig. 3.3 has been adapted from [144], figs. 5.2 and 5.3 have been adapted from [167], and figs. 5.4 and 5.7 have been adapted from [162].

Acknowledgements

I would like to thank professor Arne Solvberg for initiating and supervising this work. His ideas about information system engineering and the need for performance evaluation as an integrated part of system development was the starting point of this thesis. He has continuously criticised and suggested improvements in the draft versions of the thesis, and his contribution to the final result is large.

Professor Peter Hughes first introduced me to the field of computer performance evaluation. His ideas of composite workload modelling are very important in this thesis. He has supervised the work, and contributed to numerous improvements in the earlier drafts.

Vidar Vetland and Gunnar Brataas have been my fellow doctoral students and co-workers on the IMSE project. A lot of the material of this thesis has reached its final form only after hours of discussions with them. Also, Gunnar Brataas has contributed significantly as a user of the PrM tool presented in part IV.

Erlend Leganger and Peter Wieland have both written their Diploma Theses in fields related to this thesis. They have both contributed to my understanding of some basic aspects related to this work.

Dr. Guttorm Sindre has been contributing numerous ideas to the thesis, especially in the field of information system engineering and with regard to the PrM language introduced in sec. 3.3.2.

The RHAPSODY and IMSE project groups have been very useful and inspiring environments for discussion throughout the thesis work.

Finally, I would like to thank the edp-personnel at The Regional Hospital in Trondheim, Tandem Computers in Trondheim and Oslo, and Twinco in Oslo for their continuing support and supply of information in connection with the case study mentioned in chapter 1.
Part I

Introduction and Motivation
This part will introduce the thesis and motivate the problem considered in it.

- Chapter 1 “Introduction” presents the problem statement, the chosen solution approach, and the major results of the thesis, while
- chapter 2 “The Need for Performance Engineering of Information Systems” motivates the thesis from several important points of view.
Chapter 1

Introduction

This introduction chapter will state the problem addressed in the thesis, the chosen solution approach, as well as the major results achieved. An outline of the thesis is given and related theses pointed out.

1.1 Problem Statement

The problem addressed in this thesis is stated as:

How can the performance of information systems be predicted and improved during development.

In this statement, the “performance” of information systems is the response times and throughputs seen by their users, as well as observable quantities associated with the corresponding computer, i.e. resource utilizations and lengths of resource queues in the operating system. An “information system” is a system for representation and dissemination of information. “Predicted” means that estimates of information system performance should be available during development, while “improved” indicates that the process of bettering the predicted performance should also be supported. Finally, “development” means the process of creating new information systems, from early problem analysis to implementation, testing, and maintenance.

In the next chapters, the above problem and the solution presented will be named performance engineering of information systems, or sometimes information system performance engineering. However, the definition of these two terms will be generalised slightly in the introduction to chapter 5.

1.2 Approach

The approach chosen to solve this problem resides on three pillars:
• a *framework* for performance engineering of information systems;

• a *CASE-tool* realising the framework, and

• a practical *case-study* validating and suggesting improvements of the framework and CASE-tool.

Establishment of a *framework* was chosen to make the results of the thesis widely applicable. The alternative would have been to study performance prediction and improvement in context of a *single* (or a few) information system development methodology(-ies) and a *single* (or a few) specification language(s) only. The results of such a study would have been less relevant for other methodologies and languages. The framework, on the other hand, attempts to make as few assumptions about methodology and language as possible. The potential drawback is lack of concreteness. Therefore, the CASE-tool has been implemented to demonstrate the possibility of realising of the framework. Feedback from the practical case-study has also ensured concreteness. In addition, it has turned out that the abstractions introduced by the framework (vide infra chapter 7) has added *clarity* to the approach, because only relevant aspects of information systems have been included. (I.e. *relevant* with respect to performance engineering of information systems.)

The thesis and the framework focuses on *modelling* as opposed to *methodology*. While the modelling problem is related to *representation* of the information system performance engineering domain and *analysis* of such representations, methodology focuses on the process of *establishing* representations and *interpreting* analysis results. Hence methodology considerations rely on a well-understood framework for modelling and model analysis. Since no such framework existed when the thesis work was initiated, performance modelling of information
1.3. MAJOR RESULTS ACHIEVED

systems was focused on. Again, the decision supports generality in that few assumptions about methodology have had to be made in the framework. One assumption that has implicitly been made, however, is that the information system development methodology must essentially be modelling oriented. An example to the contrary is the prototyping oriented family of development methodologies discussed in sec. 2.3.

The modelling framework recognises and distinguishes between three levels of computing in a computerised information system. Fig. 1.1 shows how an organisation uses applications which in turn run on a computer. The applications are projected while in development and existing after they are put in production. The computer is represented as a performance model, while applications are represented by workload models. The organisation defines possible workload scenarios and performance goals for the computerised information system. While hardware performance modelling is relatively well-understood, the organisation level can be represented simplistically. Since establishment of workload models for existing software systems is treated in a related doctoral work (Vetland [190]), the framework therefore focuses on workload modelling of projected applications.

Introductions of the framework and its CASE-tool realisation constitute the two main parts (III and IV) of this thesis. The case study [39] has been carried out in cooperation among several doctoral students, and is therefore not reported here. However, the practical experience gained from the study has continuously influenced and served as a test-bed for the ideas of this thesis.

1.3 Major Results Achieved

The major result of the thesis is therefore the framework for performance engineering of information systems. The framework in its basic form supports performance prediction from information system design specifications which have been annotated with performance parameter estimates. In the course of the case-study, it became clear that capturing these parameters was a main effort. Hence, the basic framework was extended with a battery of parameter capture support techniques, for:

- parameter validation which compares early parameter estimates with later measurements or analysis results;

- residence time analysis analysis which tells the user which parts of the design specification to annotate;

- software platform modelling which lets the user annotate the design specification in terms of development concepts like programming language sentences, screens, and database accesses rather than CPU operations and bus transfers, and which frees the users from the responsibility of estimating operating system overhead when annotating the design specification;

- bounds analysis as an alternative to analysis of means early in the development, and

- parametric analysis in case one or a few parameters are infeasible to obtain.
• sensitivity analysis which tells the user which parameters to estimate with most care.

In particular, the sensitivity analysis technique has provided results that are useful beyond the scope of the framework.

Since the development towards distributed computer systems has spawned research trends in both the information system and performance evaluation fields, the framework has also been extended with the capability of predicting the performance of distributed information systems during development. Finally, the work of this thesis has resulted in a working CASE-tool for performance engineering of information systems.

A list of contributions is presented in appendix A, where it is clearly stated what is — and what is not — the author’s own contributions to the thesis. Furthermore, the main sources that the thesis builds upon are identified for each part.

An overview of the thesis is given in table 1.1.

1.4 Related Doctoral Works

The thesis has been written in connection with two forthcoming doctoral theses about related subjects:

• One thesis is about establishment and maintenance of composite workload models for existing software systems (Vetland [190]), and

• another thesis is about performance parameter capture as part of the information system development process (Brataas [41]).

Therefore, the thesis will not cover these important aspects of performance engineering during information system development. In particular, the work presented here makes as few assumptions about overall development and performance engineering methodology as possible. Instead, the appropriate, forthcoming theses will be referenced.
Outline of the thesis

The thesis is about how the performance of information systems can be predicted and improved during development.

Part I introduces and motivates the thesis:

- an introduction is given (chapter 1), and
- the need for performance engineering of information systems is discussed (chapter 2).

Part II presents the state of the art in the fields of:

- information system engineering (chapter 3);
- computer performance evaluation (chapter 4), and
- software performance engineering (chapter 5),

as well as

- the requirements for the remainder of the thesis (chapter 6).

Part III presents a framework for performance engineering of information systems, comprising

- a baseline view of the related fields (chapter 7);
- a performance prediction technique (chapter 8);
- techniques for parameter capture support (chapter 9);
- support for sensitivity analysis (chapter 10), and
- analysis of distributed information systems (chapter 11).

Part IV realises the framework in terms of the PrM language, interfacing

- the basic framework (chapter 12) and
- the framework extensions for parameter capture support, sensitivity analysis, and distributed information systems (chapter 13);

Finally, the thesis is concluded and possible and necessary paths for further work are outlined (chapter 14).

Table 1.1: A brief outline of this thesis.
Chapter 2

The Need for Performance Engineering of Information Systems

This chapter will discuss the need for performance engineering of information systems and thus motivate the remainder of the thesis. First, sec. 2.1 points out that a main aspect of computers is their ability to perform work. Sec. 2.2 then discusses why quantitative work assessments are needed in the fields of computer science and informatics, while sec. 2.3 shows how the lack of such assessments are problematic in contemporary computer practice. Finally, sec. 2.4 demonstrates the insufficiency of the existing alternatives to performance engineering of information systems.

2.1 Information Systems and Computers

For the last one or two decades, the world has been entering the “age of information.” Production equipment and production resources are no longer the determining factor in industry and commerce. The world is moving from the industrial society into an age were access to and control of information has become the main success factor. This new situation has been coined “the information society,” in which information has become a primary concern of all organisations, whether commercial or non-profit, and whether in the private or public sector.

One dominating phenomenon occurring with this development is the widespread use and overwhelming importance of computers. Computers have become the main tool for accessing and managing information. The fundamental asset of computers is their ability to free the human participants in organisations from the mechanical tasks of information processing. Hence humans are freed to concentrate on utilising this information. This enables the organisation to provide new information services to its clients, and increases the speed with which such services can be provided. The latter point is important as internationalisation rapidly increases market competition.

Thus one of the main aspects of computers is their ability to perform work that was previously carried out manually. Computer work should be understood in a wide perspective, including aspects such as the function performed and its quality — both with respect to process and
result. Additional aspects are reliability, cost, usability, ease of use, serviceability, security, availability, applicability, accessibility, and speed. This thesis will focus on the latter aspect: the speed with which computers perform work, or in other words their performance. The next two secs. 2.2 and 2.3 will argue why this aspect is important. Examples are problems such as:

- Comparing the strength of different computers by comparing their ability to perform work,
- comparing different realisations of a computation task by comparing the work they induce, and
- comparing the match between a computer and a computation task by comparing the power of the computer, the work of the computation task, and the time available to solve the task.

Yet, the commonly accepted measures of the performance aspect of computer work are not absolute, in the sense that they are expressed in terms of resource demands on some abstract machine. Since such measures are only valid for that particular abstract machine, it is difficult to apply them in a general way. Therefore, quantitative work assessments are uncommon. New computer and software systems are being created without thoroughly founded considerations on time, capacity, and resources. This thesis motivates and introduces such considerations, with emphasis on information system development. To prepare for this, the next sec. 2.2 discusses the science theoretic importance of computer performance, while the following sec. 2.3 as well as the remainder of this chapter motivates computer performance from a practical point of view.

2.2 Informatics and Computer Science

To explain why performance is a fundamental aspect of computer work, the fundamentals of natural sciences — and engineering — upon which computer science was initially based must be surveyed. According to Kolence [115], a scientific theory must

1. simplify and unify a diverse set of empirical observations;
2. quantitatively predict the empirically observed values of known phenomena, and
3. predict the existence of and the quantitative values of hitherto unobserved phenomena.

A field of engineering is an application of such a science for a practical purpose.

Thus computer science may be regarded as the “scientific theory” whose purpose is to support the technical aspects of automated data processing, while informatics is the “science” that supports the use of such automated processing in context of the humans and organisations surrounding them.

The key point in the above definition is the predictability requirement of point 3. This point is related to the Popperian falsification requirement of science [150]. According to Popper, a
scientific theory must be falsifiable in order to qualify as a "proper" science. Using Popper's criterion, most branches of natural science — such as physics, chemistry, and biology — are proper sciences, while life and social sciences — in general — are not.

Computer science and informatics today satisfy the first half of the above point 3 in that the predict the existence of phenomena. However, invalidating the quantitative values of these phenomena is not supported to the same extent. Computer science and informatics are therefore not proper sciences in their present form. Hence if computer science shall become a branch of the natural sciences, quantitative falsification criteria should be provided. The thesis claims that one fundamental shortcoming in this respect is

- the lack of a universally accepted definition of the work of computers with respect to performance, and
- the resulting lack of practice of such a definition.

If such a definition existed and was an integrated part of computer practice, the computer science and informatics fields would be on their way to become "proper" sciences in the Popperian sense. As a result, the computer engineering field would be pervaded by quantitative assessments of concepts such as "time," "speed," "capacity," "power," "efficiency" and "work."

In common engineering this is the case, and speed, capacity, power, efficiency, and work measures are routinely applied both in theory and practice. When designing a bridge, assessments of strength and capacity are made continuously throughout construction in order to ensure that the finished bridge will satisfy a set of quantitative requirements. These quantitative requirements are necessary if the bridge shall meet its qualitative requirements: connecting two points on different sides of the river. This is the heart of the problem. Since traditional fields of engineering aim at comparatively simple qualitative requirements, all effort can be focused on meeting the quantitative ones.

Computer science turns this situation around: The complexity of contemporary information systems, their surrounding organisations, as well as their users, make the qualitative requirements on computers very complex. Thus focus in the field has traditionally been on satisfying these qualitative requirements, almost completely neglecting quantitative assessments. While traditional engineering disciplines are based on continuous mathematics (suitable for quantitative assessments), the computer science and informatics fields are therefore based on discrete mathematics (suitable for qualitative assessments) — most notably 1. order predicate logic. Although this has traditionally been close to being acceptable, it will be less so in the future, as the next section will point out.

The lack of universal performance considerations in the computer science field should also incarnate itself as shortcomings in contemporary computer engineering practice. Such shortcomings exist, and they will be described in the next two sections. In fact, they have been the main initiators of the work of this thesis. These initial comments serve as an explanation why the problems to be described have come into existence.
2.3 The Importance of Information System Performance

As pointed out in sec. 2.1, efficient utilisation of information technology has become a decisive competition factor in industry and business. Although the price of computers is steadily decreasing, the total hardware expenses are increasing along with the demand for computing power. As a result, hardware expenses have become visible at the organisation level, and therefore manifest themselves as a limited resource. At the same time, the consequences of unacceptable performance have become more serious due to tightened market competition, e.g., loss of customers because of slow system responses. New methods must therefore be applied to utilise the organisation’s computing resources more efficiently.

Hence, the most important motivation behind performance engineering of information systems is that information system performance is attracting attention from management, because [141, 173]:

1. the total cost of computing equipment is increasing;
2. computer hardware expenses have become visible at the organisational level and therefore subject to constraints;
3. the proliferation of computer in society leads to increased complexity in the information system infrastructure;
4. balancing and tuning problems grow as organisations change more rapidly than before;
5. information systems are becoming increasingly important to organisational efficiency;
6. the penalties for slow responses are increasing because of market competition;
7. increased workload on existing information systems has revealed previously hidden performance problems.

Throughout the history of computers, hardware costs have fallen dramatically. Although the total computing costs in the organisation have increased because more computers and more software is now needed by more users than before, the price of the computing resources needed by an average user has fallen. This seemingly paradoxical situation is explained by examining three key parameters:

- $R$ — the price of the computing resources (hardware, software, networking, and management) needed by an average user;
- $N$ — the total number computer users in the organisation, and
- $C = NR$ — the total computing expenses of the organisation.

Because the dramatic fall in hardware costs has partially been counterbalanced by increased software, networking, and management costs, parameter $R$ has decreased less than the number $N$ of computer users, which has sky-rocketed by orders of magnitude. Hence the total computing expenses $C$ have increased while $R$ has fallen.
Today, each employee is therefore an expensive resource compared to the computing hardware and software the employee requires. This situation is the opposite of what used to be only 15 or 20 years ago. The performance evaluation field should change its orientation in response to this: From being a utilisation and throughput oriented business in the batch processing era, it must become increasingly concerned with response time considerations [138]. In the age of interactive computing, performance evaluation should be oriented towards the users of systems. This suggests its intrusion into the user-oriented business of developing information systems. Also, Ferrari [79] and Kolenec [117] have argued that computer performance evaluation and capacity management should be more closely interfaced with software development.

Three trends have however been thought to counter the need for performance evaluation of information systems [141]:

- **the steadily dropping cost of computer hardware:**
  Since computers have become so cheap it is believed by some that the additional cost of performance evaluation during software development should rather be invested in computing equipment. Additional computer equipment will increase the possibility that available computing resources are sufficient for the projected information system. Some may also state that computer hardware is a lasting commodity, as opposed to a performance study which is valuable within the scope of a particular development project.

- **the steadily shortened software development time due to prototyping:**
  Since (rapid or evolutionary) prototyping makes it possible to create an initial version of the projected information system quickly, some may argue that a performance study cannot be successfully completed until the projected system already exists. When a system already exists, quicker, cheaper, and more accurate measurements can replace performance predictions.

- **the trend towards automated code-generation tools:**
  Since production quality information systems may soon be derivable from design specifications, it is sometimes claimed that performance prediction techniques will become obsolete. Again, cheaper, quicker, and more accurate performance predictions can be obtained by measuring the generated code itself, as soon as the design specification has been established.

Either of these three points may be relevant for certain projects. However, this thesis argues that they are all doubtful, and will be so even more in the future. Each one of them will therefore be discussed briefly.

**The dropping cost of computer hardware** Although the price “per bit” of storage capacity and “per gate-switch” of processing capacity has fallen dramatically, the total cost of computing is increasing in most organisations, as has already been pointed out. The main reasons are [140]:

- the total demand for computing power has been growing more than hardware cost per power has fallen;
• the amount of basic software that must be purchased along with the computer hardware has increased greatly, and

• the amount of application software that must be purchased has also grown.

As a result, computer hardware expenses have become visible at the overall level of the organisation and therefore subject to constraints. This makes controlled management of computer software and hardware performance mandatory.

Furthermore, it is untrue that a software performance study is valuable only within the scope of a particular development project. Instead, the study will result in high-performance software whose added value will accumulate over the years in terms of computing equipment expenses saved. Also high-performance software is a better investment than large-capacity hardware, as its life-time is usually longer. Furthermore, sec. 6.2 will argue that the workload and performance models created as part of a performance study is a lasting commodity since it can be reused in later performance studies.

Prototyping The field of prototyping is commonly partitioned into rapid as opposed to evolutionary prototyping, depending on whether the early prototype is thrown away or evolves into a production system. This paragraph makes an alternative distinction based on the purpose of the prototype, whether rapid or evolutionary. The three most important kinds of prototypes are therefore [124]:

• User-interface prototypes are used to give users early experience with the user interface part of the projected information system. They are useful when user requirements are not easily expressed. Such prototypes are typically throw-away, as they concentrate on a limited system aspect. In particular, they contain no system functionality;

• Functional prototypes are used to give users an initial idea of the functions provided by the projected information system. Such prototypes are also used to fix user requirements at an early stage before development proceeds. Functional prototypes are less likely to be thrown away than user-interface prototypes. A functional prototype may evolve into a production system through a process of stepwise refinement.

• Performance prototypes are used to ensure that the performance of the projected system will be acceptable. The user-interfaces and functionality of these prototypes need not bear any resemblance to those of the projected information system. Performance prototypes are therefore always throw-away.

Hence, user-interface and functional prototypes provide little insight into the potential performance problems of the projected system. In addition, the final design and implementation will be carried out more quickly than in projects where no early prototypes were made. This means that the possibility of correcting performance problems during design and implementation is even smaller than usual. Rather than discouraging performance evaluation during design, user-interface and functional prototypes therefore make it mandatory.

Performance prototyping, on the other hand, is an alternative to the modelling-oriented framework for performance engineering of information systems presented in this thesis. Performance
prototypes are more realistic than models, and therefore they are likely to provide more accurate results. On the other hand, performance prototypes must be designed and implemented before they can be run. Since performance prototypes are always throw-away, the high cost of design and implementation has no value beyond that of the performance study. In addition, the computer on which the projected application will run in production must be available for prototype testing and measurement. This is not always the case, and it is not sufficient to run the prototype in isolation on a dedicated pilot computer, since the effects of interaction with existing applications must also be studied.

Finally, the potentially higher accuracy of performance prototypes may be useless, because:

1. In the early stages of information system development the important questions are: “Will performance be clearly good enough?” and “Will performance be clearly unacceptable?” rather than “What exactly will the performance be?”, as will be pointed out in sec. 9.4.

2. The degree of accuracy with which important prototype parameters (e.g. resource demands and execution profiles) can be estimated limits the accuracy with which the performance prototype predicts the performance of the projected application.

When discussing the requirements for performance engineering of information systems, chapter 6 will point out that cost efficiency is the most important factor limiting its use. From this point of view, performance modelling is preferable to performance prototyping.

**Automated code generation** Although automated code generation is likely to reduce the need for performance engineering of information systems somewhat in the future, it will not be made obsolete today and for a long time, because:

- understanding and practice of manual design for performance is a necessary prerequisite for successful automated code generation; [85, 29] points out that the real distinction between a true automated-code generator and a prototyping tool is the ability to create efficient code. Until the manual process of performance design is well-understood, proper automated code-generators can therefore not be made.

- the very large complexity of automatically generated information systems will create serious performance problems in the future; This necessitates new techniques for managing the performance of automatically generated information systems. In addition, automatically generated software systems are likely to exhibit worse performance than the ones implemented manually for a long time to come.

- automated code generation and software component reuse makes overall information system performance unpredictable. The tendency today is to build new information systems on top of already existing systems [178] or by reuse of already existing software components [68, 97]. Interactions between software components whose individual performances are acceptable, is likely to
create new performance problems. This further strengthens the need for performance considerations to become an integrated part of the information system development and management process.

Thus the increase in the complexities of contemporary information systems makes continuous performance assessment during development mandatory, either in its manual or in an automated form. In addition, current trends in the direction of reuse-based composite software systems call for advanced performance management techniques.

In addition, the provision of automated code generation will shift development effort towards the earlier stages of development. Accordingly, important design decisions will be made in the initial design stages. Performance considerations early in the life-cycle will therefore still be needed.

Finally, automated code generation is still at the experimental stage. It is not clear if and when code generation of production-quality commercial systems will be possible.

2.4 The Insufficiency of Contemporary Practice

According to [140], successful resource management should lead to high degrees of utilisation for the organisation’s computers, while their response times satisfy the users. The balance between hardware investments and performance should be good with regard to efficiency and satisfaction in the organisation. Correctly sized hardware expansions should be made when growth in the use of the computers make them necessary. New software systems should show acceptable performance from when they are put in production. This implies that the software must have been appropriately designed, and that the computer hardware has been upgraded sufficiently in advance.

Nevertheless, existing methods for efficient utilisation of computer hardware resources are not widely employed. Capacity planning, aimed at matching supply and demand for hardware resources, is left to intuition and rules of thumb. The result is overdimensioned computer installations “to be on the safe side.” This ties up valuable investment capital and conceals bad resource utilisation. The alternative is unacceptable response times leading to weakened competitiveness in the worst case, as a result of inefficiency and dissatisfaction in the organisation. In addition, purchasing computing equipment earlier than needed ties up capital and may fail to take advantage of the decreasing price/performance ratio in the marketplace [140].

New software systems are often introduced without concrete knowledge about expected performance and resource demands, and without knowledge of how the new system will degrade the performance of other, already existing software systems when put in production [173, 140].

The conventional “fix-it-later” response [167, 18] to this challenge dominates in the software development communities. Such development projects aim at designing easily modifiable application code and then “redesigning in case of trouble.” Good performance is only ensured through rules of thumb and intuition. This approach is becoming insufficient because [138]:

1. implementation patches cannot remove design errors;
   In “fix-it-later” projects, performance problems are not detected prior to system testing,
Implementation patches at the user site sometimes resolve smaller problems, while severe ones leads to delayed deliveries. However, performance problems are more often caused by bad designs than by inefficient implementations. In development projects where the performance aspects were continuously evaluated throughout the development, the responsiveness of the final application has been reported to be as much as two orders of magnitude better than in projects applying the “fix-it-later” strategy [169].

2. **redesigning takes effort and leads to development project overruns**;
Redesigning or reimplementing a finished system takes a lot of effort, and is costly for developers in itself. In addition, the additional time and manpower costs have rarely been accounted for in development project plans.

3. **the cost of redesign becomes prohibitive as organisations come to depend on the availability of their computerised information systems**;
Due to the increased importance of information systems in modern organisations, the cost of performance redesigns and delayed deliveries has become considerable [168].

4. **ad-hoc changes to the code make it less maintainable**;
The many small, ad-hoc changes made to a system make it more difficult to maintain, without improving performance as much as could have been achieved at earlier design stages. This is again because the causes of problems more often are bad designs than inefficient implementations.

5. **patching the code creates serious maintenance problems in case of automated-code generation**;
Additional problems with implementation patches is introduced by automated code generation. In such cases, changes in the code make the production code inconsistent with its higher-level specification. There is no way to ensure that validations and verifications made on the design specification apply to the application in production. Furthermore, if changes to the design specification must be made and the code regenerated, all the performance patches must be redone by hand [141].

6. **there is a need to undertake cost-benefit studies before initiating an application development**;
Another requirement making performance analysis necessary early in information system development, is the need for cost-benefit studies before initiating an expensive development project. Such studies strongly benefit from predictions of application performance. The cost of developing and introducing an information system in an organisation must always be weighed against its benefits. An important aspect of this “benefit” to the organisation is its improved efficiency. Without considering the performance of the information system, no thorough analysis of its impact on organisational efficiency can be made.

7. **the “fix-it-later” strategy provides no integration with capacity management**;
Performance predictions during information system development are needed in order to take advance actions in case predicted response times are unacceptable. One possible action is hardware upgrades, thus linking information systems performance evaluation with capacity planning.
8. the “fix-it-later” strategy provides no integration with database design.
As most information systems depend heavily on a database management system, database performance is crucial to overall information system performance. Design of a physical database that is optimal with respect to the performance of a particular application requires a thorough understanding of its design as well as of how it will be used by the surrounding organisation. Such information is typically captured as part of a software performance engineering study. However, the “fix-it-later” strategy provides no means of utilising this information. (The presence of a database “optimiser” whose non-optimal optimising algorithm is usually unknown severely complicates this picture, as will be pointed out in sec. 9.3.5).

As a consequence, time and manpower schedules are overrun when response times finally turn out to be unacceptable. Thus “fix-it-later” is no longer an effective, nor a cost-efficient strategy.

In some cases [140], application performance is tested on an isolated pilot-computer during development. This is expensive, and does not take into account that the software will later run alongside other systems. In addition, pilot testing cannot take place until the implementation phase. In both cases, it has become too late to correct the design flaws causing the bad performance.

Other approaches are not sufficient in coping with the above problems either [141]:

- **buying additional computer hardware on demand;**
  Since the ratio of hardware cost to power is dropping sharply, simply buying additional equipment on demand has been believed to be more cost-effective than using expensive personnel and tools to analyse the situation in advance.

  But although the price-to-power ratio is falling, the amount of power needed by the organisation is increasing even more, making the (price × needed-power) product grow. As a result, computer expenses attract increasingly bigger attention at the management level.

- **balancing out performance problems between different computers;**
  Since most larger organisations have several computers, it has been believed that applications may be moved between them in order to “balance out” system bottlenecks:

    When one of the systems reaches peak load e.g. for its I/O configuration, an I/O-bound suite of applications may simply be swapped with a set of CPU-bound applications from a machine where the central processing unit is becoming critical, giving performance improvements in both systems. Thus it is sometimes stated that there is always “a way out” of problems occurring because of a new, inefficient application.

    First of all, it is untrue that balancing “always” is possible: It requires that several compatible computers exist in the organisation, and that the software systems in question are at all possible to move. The latter point is not at all trivial, especially when older, patched-up transaction processing systems are concerned. In particular, large and complex system-dependent data structures are a problem. Also, balancing is only a way of **postponing** software performance problems. The technique uses unexploited (and hence unnecessary) computing resources to hide performance problems elsewhere. The key problem remains: inefficient applications.
In addition, there are numerous options of how to perform the balancing. Which applications should be moved? To which machine should they be moved, etc.? Also, balancing without strong tool support is very difficult because the workload at the installation may vary its character greatly throughout a day. Modelling techniques are needed to choose the best balancing option, in particular because the cost of moving an application may be high.

- **applying rules of thumb, and**
  It is claimed that experienced information systems engineers have got an innate “feel” of how to design and implement efficient information systems.
  But while experience is an invaluable resource, it is far too scarce. This is particularly true in the field of information systems engineering with regard to designing efficient systems. Tools can never replace experts, but they are needed to help novices reach a higher level of professionalism.

- **extrapolating from past trends.**
  It has been argued that the workload growth on large systems should be smooth, because the total growth will be the sum of numerous smaller and largely independent increases. Examples are: small increases in number of users on a system, small increases in the load each old user puts on the installation, small increases from new applications installed, etc. Simple trend analysis techniques on accounting logs or measurement data can be used to ensure that new computing resources are bought in advance when system workload increases, in order to avoid unacceptable response times or throughputs.
  But this technique does not take into account the installation of large new applications. In such cases, after months of analysis, design, implementation, and training, system managers still have no clue of how well the computer will perform in a live environment, or what the overall impact on system performance will be. This is clearly not good enough.
Part II

State of the Art and Requirements
This part will present the state of the art in the field of information system performance engineering as well as in the information system engineering and computer performance evaluation fields upon which it builds.

- Chapter 3 “Information System Engineering” presents the state of the art in the information system engineering field;
- chapter 4 “Computer Performance Evaluation” presents the state of the art in the computer performance evaluation field, and
- chapter 5 “Performance Engineering of Software Systems” presents the states of the art in the software and information system performance engineering fields.

Since this thesis is oriented towards performance modelling of information systems, each chapter will focus on modelling, as opposed to methodology and tools. Finally,

- chapter 6 “Requirements for Performance Engineering of Information Systems” summarises and elaborates on the requirements for successful performance engineering of information systems.
Chapter 3

Information System Engineering

This chapter will describe the state of the art in the information system engineering field. Since the thesis is oriented towards performance modelling of information systems, the chapter will focus on information system modelling, as opposed to tools and methodology.

An information system can be defined as “a system for representation and dissemination of information.” This definition relies on such general concepts as “system” and “information” which will be left undefined. It indicates that an information system is

- a system for management of information;
- (commonly) based on computer technology, and
- related to manual information processes.

In chapter 7 it will be made clear that the term information systems — as used in this thesis — does not include these related manual processes. More specifically, the information systems considered in this thesis will typically

- have large numbers of homogeneous users;\(^1\)
- be interfaced through terminals and PC’s;
- be based on a database and a screen management system, and
- run on a single or a few mainframes or mini-computers.

Computerised information systems are therefore distinguished from other kinds of software systems, such as real-time or embedded systems; scientific or technical software systems; computer-aided manufacturing and design systems; expert systems and artificial intelligence systems, as well as basic and operating software systems.

Information system engineering can be defined as “the practice of developing and maintaining computerised information systems.”

\(^1\) Sec. 8.3.1 will explain why this is necessary.
The above definition and the title of the thesis indicates that the other types of software systems listed above are not considered in this thesis. However, this does not mean that the framework to be presented may not be applicable to other types of software systems also. Instead, it means that the question is left open for now, and in sec. 3.3.2 it will be indicated that the framework is at least likely to be useful in the context of real-time systems.

3.1 Software Engineering

The field of software engineering — from which information system engineering later evolved — was motivated by the introduction of problem-oriented programming languages in the late fifties and early sixties (FORTRAN in 1957, COBOL in 1960, and ALGOL in 1960). These languages grew out of the need to create larger and more manageable computer programs quicker than was possible using assembly programming languages. Although early programming languages soon introduced such concepts as “subprograms” and “datatypes,” programs quickly grew unmanageable just like assembly programs had done before them.

3.1.1 The waterfall paradigm

The software engineering field can be said to have started with the introduction of the now so-called “waterfall” paradigm (Royce [157]). The waterfall paradigm was introduced to manage the complexity of software systems during design, viewing software development as a process going through phases such as analysis, design, and coding. The paradigm emphasises

- clear definitions of the goals of each phase, as well as
- focus on transitions between phases through the concept of milestones.

Although the original waterfall paradigm was continuously revised through the seventies and eighties (e.g. by Boehm [32]), it still dominates the contemporary practice of software and information system engineering.

A “modern” waterfall paradigm might comprise the following phases, which any waterfall development is likely to proceed through, whether explicitly or implicitly:

**Problem analysis**: A model of the organisation is created, preferably explicitly, and analysed. The model may be changed to reflect necessary changes in the organisation.

**Requirements specification**: The problem analysis may indicate that the organisation would be better off with parts of its information system computerised. An “automation border” is drawn inside the organisation model to determine which part of the system to computerise.

**Design**: The task of this phase is to refine the part of the organisation model that is inside the automation borders, until it can be implemented on a computer. This is typically done using the method of stepwise refinement. In each step, the previous model is being refined through decomposition. Every refinement step requires selecting between
3.1. SOFTWARE ENGINEERING

several decomposition alternatives. System performance and responsiveness are among
the criteria in this selection.

**Implementation:** The design phase halts when the lowest-level models are so detailed that
each model primitive corresponds to a piece of code that is easy to implement.

**Testing, production & maintenance:** The implemented application must be tested be-
fore being put in production, and maintained to stay in production. It is generally
agreed that too much of the development effort in information systems engineering is
put here. The work of this thesis is an attempt to reduce part of this amount.

3.1.2 Process and data modelling

As soon as the waterfall paradigm had been established, attention was shifted to the problem
of representing the projected software system during each phase and in milestones.

The Turing and von Neumann paradigms from the earliest days of computing, introduced a
separation between dynamic/activity/behaviour/process modelling on one hand and static/passive/phenomenon/
modelling on the other. This separation was mirrored both in

- the construction of computer hardware as a CPU manipulating a memory;
- the view of assembly programming as instructions manipulating data, and
- the view of higher-level programming languages as algorithms manipulating data struc-
tures.

The advent of software representation languages followed the same separation, and further
widened the gap by separating the process and data models in two largely different communities of research and practice.

**Data oriented modelling** arrived first, with the introduction of the relational model
(Codd [67]), the entity-relationship model (Chen [60]), and NIAM (Nijssen [134, 188]).

**Process oriented modelling** followed with the arrival of flow-charts and structure charts,
dataflow diagrams (DFD) (DeMarco [71], Gane & Sarson [81]), Yourdon charts (Yourdon [193]),
behaviour modelling (Kung & Solvberg [179]) and dataflow diagram extensions (Ward [191],
Adler [1], Solvberg et al. [20]).

3.1.3 Integrated modelling approaches

However, the data and process oriented modelling communities continuously interacted in
software development practice. This interaction soon made it clear that the two were comple-
mentary approaches to software development. This led way to a number of approaches integrating the two:
loosely integrated approaches aimed at providing links between distinct static and dynamic modelling paradigms. Examples are early applications of dataflow diagrams and entity-relationship models in the same development projects, as well as

closely integrated approaches aimed at encompassing both static and dynamic modelling capabilities within a single language formalism and graphical syntax. These closely integrated static and dynamic approaches all have in common a view of the projected software system as a hierarchical organisation of entities, this hierarchical organisation being mirrored in both data and processing models. Examples are Jackson Development Method (JSD) [108], ACM/PCM (Brodie [42]). More recent approaches along this path are State-charts (Harel [90, 89]), Hicons (Sindre [165]), as well as the currently emerging family of object-oriented development methods [22].

Although closely integrated approaches seems a promising approach, it is not certain that a single graphical formalism can be made versatile and strong enough to represent all aspects of a complex software design, and to support establishment of such a representation. The alternative, loosely integrated approaches, will therefore be considered in sec. 3.2.2.3 on CASE tool integration.

### 3.2 Information System Engineering

While software engineering focuses on development of software systems in isolation from their users, information system engineering considers both the software and its environment in common. It is not clearly defined exactly when information system engineering grew out of software engineering. In general, three factors were important:

- The widespread use of software platforms consisting of database management systems (DBMS’s), programming languages, and screen handlers distinguished information systems (or information processing systems or transaction processing systems) from other types of software systems (such as embedded systems, real-time systems, basic software systems, scientific software, expert systems, etc.)
- Major improvements of computer-supported software design tools made it possible to shift developers’ attentions to the earlier phases of software development, such as problem analysis and requirement specification.
- Advances in workstation technology made a new generation of graphics-oriented software development tools possible.

From a backbone of already established software design methodologies and techniques, these developments gave rise to a separate field of information system engineering.

### 3.2.1 Information system engineering methods

In addition to the waterfall methodology, a number of alternative methods have been suggested. Some of them are already well-established, but have recently become attractive in
practice because of new computer-supported tools. Others are novel alternatives to the waterfall paradigm, or extensions to overcome some of its limitations. The most well-known alternatives and extensions are:

**Reverse engineering** is introduced as a supplement to the waterfall paradigm to meet the problems of 1) maintenance in the case of automated code generation, and 2) introducing CASE technology into existing software infrastructures [92];

**Prototyping** is introduced as an alternative or a supplement to the waterfall paradigm, to counter the problem of erroneous, incomplete, and/or unstable requirement specifications, as has been outlined in sec. 2 [124];

**Enterprise modelling** is introduced to interface the information system development process with the organisation development and business goals [164];

**Object-oriented development** attempts to introduce object-oriented techniques as early as possible in the requirement analysis and design phases [22], and

**Spiral development** is a further development of the waterfall paradigm [33].

In addition, several authors have attempted to provide expert system support for information system engineering, e.g. [57].

### 3.2.2 Information system modelling

Many types of tools to support information systems engineering are today commercially available or can be expected to become so in the near future. These include diagrammatic modelling paradigms for data-, data-flow-, process-, object-, and structure-description, and model editors for these paradigms. Consistency checkers, interpreters, animators, simulators, and code-generators can be linked with such editors, as well as translators for transformation from one model type to another. Other related tools are version controllers, project management support systems, documentation tools and report generators, as well as modelling and method advisors.

#### 3.2.2.1 Drawing tools

Drawing tools emerged, supporting creation and management of graphical models of the projected information system [92]. The drawing tools typically supported only one or a few static or dynamic modelling paradigms. In addition, some drawing tools added features like

- integrated storage and retrieval of models;
- version management, and
- consistency checking of models.
Graphical support systems have emerged to support the creation of such graphical drawing tools. These systems make it possible to create drawing tools rapidly from high-level descriptions of the graph type to be supported. Typically, graphical support systems view the graph as a network of typed nodes and links [186].

Information repositories accordingly emerged to support the storage of retrieval of models, as well as version management. These can be regarded as more advanced and higher-level versions of the data-dictionary systems of the initial generation of software engineering tools. They allow models to be stored in a uniform manner, and allow relationships to be maintained between related models, possibly of different paradigms [92].

Hence, this development mirrors the emergence of graphical support systems.

3.2.2.2 CASE tools

Computer aided software engineering (CASE) tools combine the graphical support of the drawing tools, with the code-generation facilities of 4th generation languages [92]. Other features that can be supported are

- formal model verification;
- prototype generation, and
- model animation,

as well as the advanced drawing tool features of 1) integrated storage and retrieval of models; 2) version management, and 3) consistency checking of models already mentioned. These tools are also typically limited too

- one or a few phases of software design;
- a single modelling paradigm, and
- a single or a narrow range of application domains.

3.2.2.3 Integrated CASE tools

Single CASE tools only support one or a few phases of development and only a single aspect of the projected software. Hence, two gaps arise:

Phase gaps arise because the models developed in earlier phases of development are not directly applicable in the next. For instance, specifications which are output from the requirement analysis may not be useful for requirement specification, or more commonly, requirement specifications are not useful for design and design specifications are not useful for coding.

This necessitates manual model translations between phases. This is both cumbersome and error-prone.
aspect gaps arise because the models developed within the same phase are not closely enough linked with one another. For instance, a conceptual (real-world) model of the static aspects of a problem domain during requirement analysis, may not be connected with its complementary dynamic model other than through naming. During design, a DFD-type model of the processing structure may not be properly integrated with the functional hierarchy.

This prohibits consistency checking and formal verification across model boundaries, making it impossible to ensure the internal correctness of milestones. Furthermore, the aspect gaps make milestones more difficult to establish, as they dispense closely related pieces of information and make it more difficult for the software developer to get an overall picture of the state of development. This problem is strengthened by the size of most information system development projects, requiring cooperation among several people in a development project.

To meet these problems, a generation of integrated CASE (iCASE) tools are currently emerging [175, 176, 92, 13]. Examples include the Information Engineering Facility (IEF [110, 111, 183]) and the Information Systems Factory (ISF [106]). Such integrated tools include

- **multiple CASE tools** for 1) most phases of development; 2) most aspects of modelling, and 3) a wide range of application domains;

- a **common graphical interface system** supporting creation and manipulation of models in different languages through a similar set of commands and windows;

- a **common repository** for 1) model storage and retrieval; 2) managing versions of models; 3) maintaining relations between interdependent models; and 4) maintaining relations between interdependent internals of models.

- a **paradigm integration** providing well-defined and well-understood links between several modelling languages by 1) the CASE tools supporting the modelling languages; 2) the repository which maintains relations among the models, and potentially by 3) a set of auxiliary tools performing multiple-paradigm functions such as a) consistency checking and verification of sets of models of different paradigms, and b) model-to-model translations.

**CASE tool frames** To support the creation of integrated CASE tool environments, a generation of CASE tool frames has emerged [92], providing a combined graphical support and repository into which stand-alone CASE tools can be introduced and integrated. This development is a continuation of the graphical support systems and repository developed to support the generation of stand-alone CASE tools.

**Integrated project support environments** With the convergence of traditional modelling and modelling support tools in paradigm integrated CASE tool environments, attention is gradually shifted towards the non-modelling activities of an information system development project. Integrated project support environments [92] are extensions of iCASE tools with additional support for such tasks, including
Figure 3.1: An integrated information systems engineering environment.

- planning and keeping track of milestones, deadlines, project dependencies, man-power assignments, as well as budgets, and
- explicitly representing and maintaining a model of the development process itself inside the environment.

Tools to be included in the emerging generation of integrated environments are:

- model editors supporting different diagrammatic system specification paradigms, both operational and declarative,
- consistency checkers for formal verification,
- interpreters, animators, and simulation modules for informal validation, as well as
- code generators.

Future tools to include are aggregation and decomposition aids, version controllers, project management systems as well as modelling and method advisors. This list of tools conforms to the major activities of information systems engineering as described in the previous section.

Some major components of an integrated toolset for information system engineering are depicted in fig. 3.1.

### 3.3 The PPP Environment

The particular information systems engineering toolset used in this thesis is based on the PPP (Processes, Phenomena, and Programs) modelling paradigm [21, 20] already mentioned in sec.3.1.2.
This section introduces the PPP environment for information system engineering with particular emphasis on its PrM tool for dynamic information system modelling. The presentation will be illustrated with an example from a bank. (The bank example has been adapted from [88]. The complete example also contains PhM and PLD models, as well as Ada code and database schemas automatically generated by the PPP tool.)

### 3.3.1 A PPP overview

#### 3.3.1.1 PPP modelling

PPP (processes, phenomena, and programs) is an experimental integrated CASE tool for software development, developed by Sölvberg et al. [88]. It focuses on 1) *formality* to facilitate early verification and validation as well as automated model-to-model translation and code generation, and 2) *integration* between the modelling tools used, and between different phases of development, and between the problem analysis and system design phases in particular. PPP builds on well-known approaches to software development, such as top-down design and the DFD and ER paradigms of sec. 3.1.2.

PPP is intended for use in all phases of information system development. A tool has been implemented to generate Ada code directly from PPP specifications, as part of the TEMPORA project. Other work on PPP include case studies, method support [123], work on automatic aggregation [166], and a formal definition [145].

PPP comprises four diagrammatic modelling tools:

1. the **process modelling (PrM) tool** supports a formalised extension of ordinary DFD modelling used to represent *functional decomposition structures*;

2. the **phenomenon modelling (PhM) tool** supports a formalised extension of ER-diagrams used to represent *static objects*;

3. the **process life description (PLD) tool** supports low-level diagrammatic *algorithm* representations, and

4. the **user interface description (UID) tool** supports low-level *screen definitions*.

These four partial modelling languages are strongly interrelated. In particular, the static (PhM) and dynamic (PrM) aspects of modelling are closely coupled.

**Example**

A schematic overview of the PPP functionality is shown in fig. 3.2. The tools provide drawing tools for the four modelling approaches supported, as well as consistency checking, Ada code and database schema generation, and some minor auxiliary functions. The environment is interfaced through a common PCE Prolog [6] based graphical user-interface, and the tools share a common SYBASE database [70]. □
3.3.1.2 PPP methodology

The development strategy advocated by the PPP methodology is top-down, incremental and iterative, as shown in fig. 3.3 which is adapted from [144]. Development starts with establishing a model of the real-world system in question. The resulting real-world model describes both the existing manual information system and its environment. After refining and possibly altering the real-world model, it is decided which part of the real-world organisation to automate. This is expressed using automation boundaries on the refined real-world model. The part of the model inside the automation boundaries becomes the first, abstract specification of the computerised information system. This approach couples the analysis phase closer
with design as emphasised above. The initial design specification is incrementally refined by iterated decompositions. This proceeds until executable code can be automatically generated from the lowest level of specification.

The development team utilises the various tools provided in accordance with a method, organised around the phases of problem analysis, requirements specification, design, implementation, testing, and maintenance. This method is top-down in the large, although smaller bottom-up corrections are occurring all the time. The problem analysis creates an organisation model. The requirements specification then defines automation-boundaries around this model. The design phase gradually refines the part of the organisational model within these borders. This is done by decomposition, where each component of an existing model is refined — rather than replaced — by a more detailed submodel. This proceeds until the lowest level of the design specification has become a program description that can be implemented and tested.

The information system design phases as defined in sec. 3.1.1 can be recognised in fig. 3.3. Problem analysis corresponds to the task of establishing and possibly altering a refined real-world model of the enterprise. Requirements specification consists of defining the automation borders and formulating additional requirements, such as performance goals for the projected application. The design phase corresponds to the stepwise refinement of the design specification. Since PPP generates code from the lowest level of specification, the implementation phase is vaguely resembled in the lowest-level process life descriptions (PLD) of PPP. Testing, production and maintenance are not represented in the above figure.

This thesis focuses on the design phase where a design specification of the projected application is incrementally refined through decomposition.

3.3.2 A PrM overview

The PrM language for software specification has been developed by Svolberg et al. [21, 20, 145].

The process modelling (PrM) paradigm is a formalised extension of DFD modeling [81, 71]. The PrM paradigm was established to avoid several unclear issues in conventional DFD models [122]:

- PrM flows may denote flow of control as well as data flow.
- PrM introduces the concepts of process triggering and termination to define the dynamics of PrM specifications.
- PrM introduces connectives to define clearly what is consumed and produced by each process per execution.
- PrM also introduces operations to distinguish between different ways of triggering a process or a model.

Each of these extensions will be elaborated upon in sec. 3.3.2.1. Hence the PrM approach is related to some of the real-time extensions to DFD that has been introduced (e.g. Ward [191]
and Adler [1]). (As the framework of part III will be realised in terms of the PrM language in part IV, this is an indication that the results of the thesis are also interesting in the context of real-time systems.) Also, PrM flows and stores are linked with PhM entity classes in order to strengthen the typing of the dynamic models and to integrate static with dynamic modelling.

The PrM language covers several phases of the design cycle from real-world modelling in the problem analysis phase to detailed functional descriptions in the late design phase. This has lead to the introduction of additional modelling constructs such as resources and timers. In this thesis, only the PrM constructs that are relevant to design are considered.

![Diagram](image)

Figure 3.4: PrM constructs: process (a), store (b), agent (c), and flow (d).

3.3.2.1 PrM modelling

A PrM model is a graph of processes and stores connected by flows, as in traditional dataflow diagrams (DFD) [81, 71]. In addition, PrM introduces agent nodes, representing something that is external to the specification and therefore not modelled in detail. The processes, stores and agents in a model exchange PrM items through the flows.

A PrM process represents some transformation from inputs to outputs. Inputs and outputs are represented as items. Per execution the process consumes input items and produces output items, respectively. Processes are depicted by round-cornered boxes as in fig. 3.4a.

A PrM store represents storage of items. Stores are produced to and consumed from by processes and possibly by agents. Stores are depicted by flat boxes that are open on the right side as in fig. 3.4b.

A PrM agent represents something that is external to the model and is therefore not modelled in detail. Agents may produce and consume items like processes do, depending on whether they are input or output agents, or both. Agents are represented by cones with a head on as in fig. 3.4c.

Thus while processes and agents represent active entities in PrM, stores are passive.

A PrM flow represents a path that an item may follow between production and consumption.

Flows constitute the edges of the PrM model graph, whose nodes are processes, stores, and agents. A flow edge may not connect two store or two agents. Flows from or to agents are the external input and output flows of the model. Flows are depicted by arrows as in fig. 3.4d.
Figure 3.5: Top-level PrM model for an Customer transaction processing system.

**Example**

The PrM model for a Customer transaction system for a bank is shown in fig. 3.5. In this simple example, a number of Clerks access the software system through four functions. In each interaction, a customer account is initiated or terminated, or a deposit or withdrawal is made to or from an existing account. The Customer transaction processing system will be used as a main example throughout this thesis.

The PrM model for an Accounting system for the same bank accordingly is shown in fig. 3.6. This example demonstrates some other aspects of the PrM specification language. A number of Clerks access the software system through a screen menu giving two choices. A set of Balances in a database are then either retrieved or updated. Afterwards, the result is presented to the clerk on the screen. In some cases, the Accountants are automatically notified when an update attempt is made. The Accountants may then check the Balances through a simpler line-oriented interface. The Accounting system will be used as the main example in chapter 11 about distributed systems.

The main PrM extensions to DFD are

- flow of control;
- process triggering and termination;
Figure 3.6: Top-level PrM model for an Accounting system.

- process connectives, and
- process operations\(^2\).

The four will be treated in separate.

**Flow of control**

The flows of PrM diagrams may denote **control flow** as well as **data flow**. This extends conventional DFD modelling, in which flows carry only data.

**Example**

In fig. 3.5, several PrM flows carry both data and control information. Examples are the four external input flows, *Initiate*, *Terminate*, *Deposit* and *Withdrawal*. The presence of control information is denoted with a T on the consuming side of the flow. These flows both trigger PrM model execution and provide the input data to that execution. □

Furthermore, a PrM flow is typed through a link with the PhM specification of the projected application. A control flow is a flow with empty type, i.e. a flow with no information apart from its presence. In information system specifications, such flows are untypical.

\(^2\)Later, PrM model and submodel operations will also be introduced.
Process triggering and termination

PrM input flows to processes may be triggering or non-triggering. Accordingly, output flows from processes may be terminating or non-terminating. Triggering and terminating flows are denoted with a T.

Example

In fig. 3.5, the Make withdrawal process is triggered by the receipt of an item on the Withdrawal flow. Process execution terminates when an item has been produced to the Write wrvl stm flow. While executing, it consumes from and produces items to the Balances store through its other input and output flows, according to the connectives that will be introduced in the next paragraph. □

Hence triggering and terminating flows in PrM specify process dynamics down to the level of triggering and termination. Items must always be consumed from triggering flows first and be produced to terminating flows last. However, the sequencing of item receipts from and productions to non-triggering and non-terminating flows is not defined. This is a compromise between ease of use and the necessity of precise specifications.

For this thesis, only triggering flows are relevant and process termination and terminating flows will not be elaborated more on here.

(a) ![Flow diagram]  (b) ![Flow diagram]

Figure 3.7: PrM connectives: AND and XOR (a), and REP, COND, REPCOND, and CON-DREP (b).

Connectives

Flows are connected with processes through connectives. Each process has one input connective and one output connective. PrM input and output connectives define the combinations of input and output items that may be consumed and produced per process execution, respectively. A connective is either

1. **AND**, meaning that all flows inside it are consumed or produced, or

2. **XOR**, meaning that one and only one flow inside it is consumed or produced

per selection of the connective as shown in fig. 3.7a. The OR-connective construct in [88] can be regarded as a composition of XOR- and AND-connectives for the purposes of this thesis. Moreover, a connective may also be
1. **repeating** (REP), meaning that the flows inside it are consumed or produced several times during one process execution;

2. **conditional** (COND), meaning that the flows inside it need not be consumed or produced, or

3. a *combination* of the two in either order (REPCOND or CONDREP)

per selection of the connective as shown in fig. 3.7b for an AND-connective. The corresponding XOR-connectives are obvious.

Connectives may be **composite**, with the obvious semantics. However, a triggering input flow cannot be conditional, for obvious reasons. This restriction will become important in chapter 11. The nesting of connectives lead to outermost, outer, inner, and innermost connectives in the obvious sense.

**Example**

The *Make withdrawal* process of fig. 3.5 therefore consumes exactly one *Withdraw* and one *Old balance* item per execution. In addition, it *may* consume *several* *Additional balance* items. On the output side the process produces exactly one *Write withdrawal* and *may* also produce a *New balance* item (in case the withdrawal was successful). □

Thus PrM connectives specify process dynamics down to the level of what can be consumed and produced per process execution. In combination with the PrM concepts of flow of control and triggering and termination, this makes the PrM language dynamically strong. In the late design phases, at the lowest levels of modelling, this is helpful. However in the early design phases, for very abstract models, the connectives may be difficult to use. Therefore they do not become mandatory until the lowest levels of specification.

When an input or output is inside some input or output connective, they are ANDed, XORed, CONDed, or REPed on input or on output, correspondingly.

**Operations**

A **PrM operation** is a way of triggering a process. Due to the presence of triggering flows that are XORed on input, a process can be triggered by more than one set of triggering flows. Each of these *cooperating sets* represent some *function* provided by the process. Although a well-designed process is likely to incorporate functions that are similar, each such function is likely to have slightly different execution characterisations and resource demands. The operation concept is introduced to represent these differences.

All PrM processes provide at least one operation.

**Example**

The *Initiate account*, *Terminate account*, *Make deposit*, and *Make withdrawal* processes of fig. 3.5 provide one operation each. (In chapter 12 it will turn out that these operations *may* be given
the same name as the flow that triggers them. This is a useful convention for operations triggered by a single flow only.)

The Write statement process of fig. 3.5 also provides just a single operation. This is a simplification made for the purpose of this section, since the four operations — one per XORed triggering input flow — that are really provided are very similar and can more conveniently be treated as a single one. The PrM tool to be presented in part IV does not allow several operations to be grouped like this. □

Every operation provided by a PrM process correspond to some way of triggering that process. If a process provides more than one operation, it must be because it can be triggered in more than one way. Each different way must distinguish itself by at least one specific flow from one specific process. This means that every operation can be represented by the flow and the process it is triggered by.

**Figure 3.8:** PrM submodel for the Make withdrawal process of fig. 3.5.

**Figure 3.9:** PrM submodel for the Update balances process of fig. 3.6.
3.3.2.2 PrM methodology

In accordance with the PPP methodology, the PrM specification method is operational, top-down, and oriented towards depth-first specification. This means that an application specification in PrM is developed layer by layer as a hierarchy of models. Any PrM process may be decomposed by a PrM submodel. With a PrM submodel is usually meant a direct decomposition. However, the term indirect PrM submodels will sometimes be used in the obvious sense. A PrM submodel is itself a PrM model as defined already. A process decomposed by a submodel is non-primitive, otherwise it is primitive. A PrM specification, \( \overline{S} \), is therefore a hierarchy of processes \( p \in \overline{S} \) and this submodels \( m \in \overline{S} \). The PrM process chain of a process is a list of processes from a process in some top-level PrM model of the specification down to the process itself.

Two PrM processes may be decomposed by the same PrM submodel, as long as the resulting directed acyclic graph (DAG) has no cycles. This DAG forms the PrM process hierarchy of the specification, where the internal nodes are PrM models decomposing non-primitive processes and the leaf nodes are primitive processes, as depicted in figs. 3.11 and 3.12. This means that a PrM specification consists of one or more top-level PrM models and all their direct and indirect PrM submodels. At any point of refinement, a PrM specification can be depicted as such a hierarchy.

Example

The PrM submodel for the Make withdrawal process of fig. 3.5 is shown in fig. 3.8. The three processes of this submodel are all primitive, as well as the Initiate account, Terminate account, Make deposit, and Write statement processes of the top-level model. The Make withdrawal process of the top-level model is therefore the only non-primitive process in the specification.

Figs. 3.9 and 3.10 accordingly show the PrM submodels for the Update balances and Get balances processes of fig. 3.6. These submodels will be used in chapter 11. □

The external input and output flows of the PrM submodel correspond to the input and output flows of the non-primitive process it decomposes. These are denoted with open circles for input flows and closed circles for output flows, with the same name as the corresponding next-higher level PrM flows. Triggering and terminating flows of the process correspond to triggering and terminating external flows of the PrM submodel, accordingly.
Since PrM submodels represent processes, submodels provide operations as processes do. And since triggering process flows correspond to triggering external submodel flows, the set of operations provided by a PrM submodel corresponds to the set of operations provided by the process it decomposes. The operations provided by a model are the external PrM operations of that PrM model. The operations provided by the processes of a model are its internal PrM operations, accordingly. The PrM model of fig. 3.6 provides exactly two external operations corresponding to each of its triggering external input flows, and seven internal operations corresponding to the operations provided by its processes. The PrM submodel of fig. 3.9 provides one external and five internal operations, accordingly.

A PrM specification itself also provides operations which will be used by the organisation. The set of operations provided by the specification corresponds to the set of operations provided by its top-level models. Since the Accounting software system specification has only one top-level model, it provides the two external operations Initiate session and Check balances of the PrM specification.

Figure 3.11: Overview of the Customer transaction PrM specification of figs. 3.5 and 3.8 with processes and PrM models.

Figure 3.12: Overview of the Accounting PrM specification of figs. 3.6, 3.9, and 3.10 with processes and PrM models.

In the sequel this will be used for performance parameter validation purposes. [166] has developed functional validation techniques for PrM hierarchies along the same lines.
Chapter 4

Computer Performance Evaluation

This chapter will present the state of the art in the field of computer performance evaluation. Since the thesis is oriented towards performance modelling of information systems, the chapter will focus on computer performance modelling, as opposed to tools and methodology.

The performance of a computer can be defined as “the behaviour of the computer with respect to time.” The performance of a computer is

- related to the speed with which the computer processes data;
- a set of quantitative aspects of the computer, and
- dependent on how the computer is being used by the organisation.

Computer performance evaluation can be defined as “the practice of assessing the performance of computers.”

As summarised in [91], the skills required for successful computer performance evaluation include (among others) designing and implementing performance measurement tools; instrumenting existing computers; measuring instrumented existing computers; modelling the performance of projected computers; modelling the workload on projected computers; stochastically simulating models of projected computers; mathematically analysing queueing models of computer performance; analysing and presenting measurement and analysis results, and validating performance models against measurements. These techniques are applicable both to

- system design and development, and
- configuration and capacity planning.

Successful performance evaluation provides additional insights into the structure and behaviour of the system being evaluated. This is particularly valuable during the early development phases in order to detect and correct design flaws [91, 173].

Computer performance evaluation is typically iterative, using several different techniques during development of a single system: In the early stages, inexact, analytic models suffice.
As development proceeds and knowledge of the proposed system increases, simulation models may become necessary. Finally, when the system has been implemented, measurements can be undertaken to facilitate tuning. Thus computer performance evaluation is characterised by the close interplay of several autonomous measurement, modelling, and analysis techniques.

Recently, several authors have attempted to provide expert system support for computer performance evaluation. An overview is provided by Potier [152].

4.1 Performance Measurement

Computer performance measurement is important for computer installation managers and computer hardware manufacturers. It is also a prerequisite for validation of computer performance models. Hence performance measurement is important both in the fields of capacity management and computer performance evaluation.

Since the performance of a computer is only meaningful in context of a particular workload on that computer, performance measurements are always carried out in the context of a “measurement workload.” A measurement workload may be

- an uncontrolled live user workload;
- a semi-controlled environment using (real or synthetic) benchmark programs, or
- a controlled environment using benchmark programs, and terminal interactors to emulate the behavior of interactive users.

The advantage of performance measurement over modelling is that the derived measures are from the computer itself. Performance measurements thus capture subtle interactions in system behaviour that may be impossible to represent in analytic models and infeasible to capture in simulation models due to prohibitively long run times. The drawback of performance measurement is that it requires the system to be properly instrumented, and that it is not applicable until (part of) the system has been implemented. Furthermore, controlled measurements require a dedicated system. The cost of setting up a controlled measurement experiment may be high, especially if a modification analysis is to be undertaken. Finally, available measurement tools are often severely limited.

The two following subsections will survey techniques for measurement instrumentation and statistical interpretation of computer performance measurements. The subsequent sec. 4.2 considers the workloads needed for measurement (and modelling).

4.1.1 Measurement instrumentation

The three basic approaches to computer performance measurement instrumentation are hardware, software, and hybrid monitors:

**Hardware monitors** are based on *hardware probes* attached to the computer hardware. Such probes are able to detect certain states and events at the hardware level, from which
event times and times between events can be collected by the monitor. Measurements
are recorded as counts and averages, or as traces. A trace is a time stamped sequence
of events amenable to post-measurement analysis.

Software monitors are based on software probes. Software probes are instructions added to
the software running on the computer to be measured. Software probes gather data by
reading memory locations and otherwise sensing status. In particular, software probes in
the operating system provide information on per program and per user bases. Powerful
on-line data reduction techniques are available.

Hybrid monitors combine the advantages of hardware and software monitors. Accurate
hardware probes which do not interfere with the system are used when possible, while
software probes are available when necessary. Using hybrid monitors, it is also possible
to record the software causes of hardware events.

4.1.2 Measurement interpretation

Measurement studies should be subjected to sound experimental methods. This implies that
objectives should be clearly stated in advance, that experiments should be carefully designed,
and that the resulting data should carefully analysed and reduced in accordance to the ob-
jectives. This is true both for live and controlled experiments. Measurement interpretation
should be driven by common sense and be based on sound statistical methods.

Although computers are deterministic systems, computer performance measurements usually
have random properties, depending on abstraction level. Reasons for this are:

- random properties of computer workloads;
  If the workload is live, user behaviour is uncontrollable and should be regarded as
  random. In case of synthetic workloads, think times, interarrival rates and job types
  will typically be approximated by statistical distributions.

- uncontrollable factors of repeated experiments;
  Repeated “controlled” experiments may produce different performance measures be-
  cause of factors that are impossible or infeasible to control.

In most measurement studies, measurements have random properties due to a combination
of the above two reasons. The consequence is that the performance measures which are de-
ferred from output sequences with random properties should themselves be considered random
variables.

The stochastic nature of performance measurements make them amenable to a well-developed
set of statistical analysis techniques. Such techniques are used to derive

- point estimates, as well as
- confidence intervals
for the resulting performance measures. Confidence intervals are particularly important when probabilistic workloads are being used. However, [91] points out that the technique is seldom applied in practice.

*Regression analysis* is widely used to obtain functional relationships of one group of variables — the *dependent variables* — on another group of variables — called the *independent variables*. Such a functional relationship, once established, can be used to predict values of the dependent variables for new values of the independent ones.

*Design of experiments* is another general technique also applicable to performance measurement. Statistical design of experiments is useful when multiple controllable factors determine the measurement outputs. Factors are varied *many at a time* to study the interactions among factors. Typically, a simple linear relationship is used to approximate the relationship between the measured responses and the controlled factors.

Although these methods are not yet widely applied, their usefulness is likely to increase along with the complexity of contemporary computers.

### 4.2 Workload Modelling

Since the performance of a computer is only meaningful with respect to a workload on that computer, *workload modelling* is a very important issue in computer performance measurement (and of course in all fields of computer performance evaluation, performance *modelling* in particular). The two following subsections will consider approaches to workload characterisation and some analysis methods needed to establish such characterisations.

#### 4.2.1 Workload characterisation

The main approaches to workload characterisation for measurement studies are live workloads, traces of live user workloads, and synthetic executable workloads. The distinction between “live” and “synthetic” workloads made here, resembles that between “natural” and “artificial” workloads made by Ferrari [78].

**Live user workloads** are not controllable, nor repeatable. Therefore they are not sufficient for scientific purposes. On the other hand they are available at no cost for existing systems.

**Traces of live workloads** may be collected and used in controlled experiments. In such experiments, the computer being measured is driven by a remote *terminal interactor* simulating a number of users executing jobs on the machine according to previously collected traces. Such traces must contain both the commands submitted by users and their think times. Traces may be recorded with little effort from live user environments. However, they may be large and inflexible.

**Synthetic executable workloads** or *benchmarks* are executable programs created specifically for measurement purposes. Standard benchmarks exist, and new ones can be written for specific purposes whenever needed. Benchmarks are typically run under
supervision of a remote terminal emulator. A main advantage is that they can be made parametric and hence flexible. The disadvantage is the potential lack of realism, especially if standard benchmarks are used. Also, writing new benchmarks is time-consuming and costly.

While initially aimed at measurement studies, the two latter workload characterisation techniques are also useful in simulation studies. Simulators can be built to use live traces or synthetic executable workloads as input. A fourth option,

**Synthetic non-executable workloads** are useful *only* in context of computer performance *modelling*. Such models typically represent the workload in terms of stochastic distributions driving a simulation or stochastic analysis.

However, it is important to note that non-executable workloads may also be useful guides for designing executable ones.

### 4.2.2 Workload analysis

A common approach to workload analysis can be summarised in five steps [91]. The WAT tool [55, 53] by Calzarossa & Serazzi supports workload characterisation along the lines of this method:

**Component selection** identifies the type of workload components to be characterised. Examples are batch jobs or job steps, interactive sessions or single commands, database transactions, etc. Decisions must be made about the *coarseness* and *abstraction level* of components, as well as the *purpose* of the workload model.

**Feature/parameter selection** identifies how each workload component will be characterised, i.e.

- by demand for *hardware resources* such as number of I/O’s to a certain device, CPU times, and memory requirements, or
- by demand for *software operations* such as numbers and types of database transactions, programming language statements, and screen handler calls.

While hardware resource demands are more directly related to hardware performance, software demands are less system dependent.

**Workload measurement** of the real workload is typically made for a long interval collecting data for a large number of components, typically several thousand execution instances or more. Hence, the resulting measurements is a collection of multivariate data.

**Exploratory data analysis** simplifies and normalises the data collection. The empirical distributions and moments of measurement features are collected. Highly skewed distributions may be transformed, and certain features may be normalised to obtain comparable ranges. *Outliers* may be removed. Care must be shown, since their impact on performance is typically large [55].
Cluster analysis reduces the number of workload components collected by grouping components with similar resource demands, and averaging the demands in each group. The purpose is to be able to treat each group as a single workload component. Typically, several thousands of job steps can be clustered into only 15–20 clusters. Several clustering algorithms exist. To ensure comparable features, range scaling will typically have been performed in the previous step.

The above steps can be undertaken in a measurement study, in which case the resulting clusters serve as guidelines when fixing the parameters of parametric benchmarks or developing new ones. However, the resulting clusters are also useful input to the performance modelling studies which will be considered in the following secs. 4.3, 4.4, and 4.5.

4.3 Performance Modelling

The two main approaches to solving performance models are simulation and analysis, which will be outlined in the following two subsections. A large number of performance modelling approaches exist, of which this thesis will be based on two: queueing networks and (generalised stochastic) Petri-nets. The subsequent secs. 4.4 and 4.5 will therefore present these two approaches, as well as analytic solution methods for such models. Parts III and IV will be heavily based on these models and their analytic solutions. Also, sec. 4.6 will present aggregation and sensitivity analysis techniques which are important for the thesis.

4.3.1 Simulation models

Simulation is the most versatile approach to computer performance modelling. It offers a potentially unlimited degree of accuracy and model detail, allowing representation of analytically intractable features. Furthermore, the range of performance measures that can be produced by simulation runs are almost unlimited, including distributions and higher-order moments. Also transient behaviour can be studied whereas analytic models mostly aim at steady-state analysis. However, this increased sophistication comes at a large computational cost. [91] points out that one major application of simulation models is to validate analytic models.

The two major approaches to computer performance simulation are:

*Trace driven simulation* is based on a deterministic simulation model driven by a trace obtained from a live workload. [91] points out that in practice, trace driven simulation has mostly been used for deterministic models, and often of systems without queues. Trace-driven simulation has been used for models of storage hierarchies, paging strategies, processor pipelines, and so on.

*Stochastic discrete event simulation* is based on simulation of a queueing model driven by synthetic, non-executable workload models. Such workloads often consist of sequences of pseudorandom numbers generated from specified distributions. Stochastic discrete event simulation is also sometimes driven by traces.
4.3. PERFORMANCE MODELLING

While trace driven simulation is usually specialised, focusing on low-level evaluation of hardware devices and operating system algorithms, stochastic discrete event simulation has a broader range of application. The latter will therefore be focused on.

*Discrete event simulation* is used for systems whose state does not vary continuously with time [130]. Instead, states change instantly. These states are referred to as events. The discrete nature of computer system states makes them amenable to such analysis, as opposed to the simulation of continuously changing states used in other branches of engineering and science.

*Discrete time*, discrete event models only consider the system at selected moments in time, which are typically evenly spaced. State changes are observed only at these moments, which are not necessarily the moments at which the events themselves occur. Such models are often used to approximate systems with continuously changing states to a chosen degree of accuracy. *Continuous time*, discrete event models, on the other hand, considers the system at the time of each event. Hence the time parameter in such models is conceptually continuous. At levels of abstraction above the hardware clock, these models are usually the appropriate choice for computer performance simulation.

Several special and general purpose continuous time-discrete event simulation languages and tools exist. These languages typically support simulation facilities such as pseudorandom number generation, event handling and scheduling, queue management, statistics gathering and analysis, as well as reporting. Special purpose simulation packages for computer performance modelling include PAWS [43], QNAP2 [151], and PIT [148, 12].

Simulation is an extremely versatile and powerful performance modelling technique. It allows a potentially unlimited degree of structural detail and accuracy. This will become particularly important as more complex computer architectures which are not analytically tractable, become available. The drawbacks of simulation is increased run lengths and statistical analysis problems. However, a wide range of special purpose tools are becoming available to relieve the user from many of the statistical considerations as possible. A key challenge is development of computationally efficient simulation techniques, as well as hierarchical workload modelling techniques similar to those found in analytic modelling. In particular, parallel processing simulators is a promising approach. Future simulators should also provide extensive graphics support for data input and output, as well as real-time animation.

4.3.2 Analytic performance modelling

Analytic performance modelling has become widely regarded as a *cost-efficient* performance prediction technique. Analytic performance models are based on simplifying assumptions about the structure and behaviour of the computer. As a result, analytic models have tractable — exact or approximate — mathematical solutions at the cost of accuracy and representativeness. However, for a very large set of problems they provide useful and sufficient insights into the bottlenecks and key parameters affecting system performance. Analytic models have been widely applied, and they are usually capable of providing throughput and utilisation estimates with about 10% accuracy, as well as response times within 30% accuracy. For a number of purposes this is sufficient:
Capacity planning utilises analytic models to determine the hardware needs of future systems based workload growth projections. Typically, capacity planning requires three steps:

1. measurement and data reduction of the current system;
2. analytic model construction and validation, and
3. model extensions to incorporate new devices and analysing the model for projected future workloads.

The key parameters of the model are varied until a configuration is found which meets future performance demands under future workloads. Analytic models can be quickly solved so that a large number of possibilities can be evaluated. Due to the uncertainties of future workloads and performance demands, the accuracy of analytic models is more than sufficient for this purpose. A number of systems for capacity planning exist, such as the BGS tools that will be described in sec. 4.7. These tools contain interfaces to measurement tools and provide data reduction facilities.

I/O subsystem modelling has become a major application area for analytic modelling due to the big difference between I/O and main memory access times. I/O system performance has become a dominating factor for the performance of many computers. This trend can be expected to continue as electronic devices become yet faster, while mechanical I/O devices cannot be expected to improve much further. Analytic models are therefore commonly used to optimise and improve I/O subsystem performance.

Preliminary design aid employs analytic models to obtain early performance estimates for proposed computers. Such studies are characterised by

- a large set of early design alternatives, and
- a small set of inaccurate parameters estimates.

Therefore, analytic model solutions are cost-efficient trade-offs between efficient solution and accuracy as in the capacity planning case. Analytic models suffice to provide insights into the key factors affecting performance of the system, as well as the sensitivities of main performance measures on key design factors. They guide the design process, and suggest the development of more accurate simulation models. In the latter case, the analytic model will point out where effort should be placed in detailing the simulation model, and limit the range of input parameters for expensive simulation runs.

4.4 Queueing Network Modelling

The computer performance evaluation field can be said to have started in the late sixties based on advances in queueing network theory. According to [102], queueing theory was first used in connection with computers to model time-sharing systems, and more precisely, central processing unit allocation policies. Much work [112, 113] was published on this problem, with focus on

1. various CPU scheduling policies, and
2. various disc management strategies.

The service centres considered consist of two parts: a queue and a server. The queue is typically described by parameters such as queueing discipline and capacity, while the server is described in terms of its service rate distribution (or inversely by service time distribution). In addition, arrival of jobs at the service centre is described in terms of a number of servers and an arrival rate distribution for each service centre.

Analysis of such service centres led to a qualitative understanding of some aspects of operating and disc management system design. The single service centre models used were, however, too restricted to allow performance prediction of systems of interacting computing devices (i.e. computers) [102].

4.4.1 Queueing networks

Later developments in queueing theory therefore studied the interaction between multiple service centres. Contemporary computers could be viewed as a set of loosely coupled hardware components through which weakly interacting “jobs” or “transactions” were circulating. The success of early queueing network theory relied on this view of computers. Hardware components were described in terms of service centres, while circulating jobs were represented by customer classes described in terms of 1) multiplicity or external arrival rates; 2) service demands at service centres, and 3) transition probabilities \( p_{ij} \) of moving on to service centre \( j \) when leaving centre \( i \).

Burke’s theorem [46] stated that a Poisson process driving a service centre with exponential service rates generates a Poisson process of departures [112]. This result implies that each service centre in a chain of simple exponential centres driven by a Poisson process, can be analysed independently using results from single service centre queueing theory.

Jackson’s theorem [107] generalised this by considering an arbitrary network of exponential service centres each of which may be driven by an external Poisson process. Although the presence of feedback paths destroys the Poisson nature of the service centre arrivals, Jackson shows that they still behave as if they were Poisson driven. Again, this implies that each service centre in a queueing network can be analysed independently, according to the equation

\[
p(k_1, k_2, \ldots, k_N) = p_1(k_1)p_2(k_2) \cdots p_N(k_N),
\]

where \( p(k_1, k_2, \ldots, k_N) \) is the probability of finding \( k_1 \) customers in service centre 1, \( k_2 \) customers in centre 2, and so on, and \( p_i(k_i) \) is the solution of the corresponding \( i \)th M/M/m service centre. The network considered by Jackson was open, in that the single customer class considered described in terms of an external arrival rate rather than in terms of multiplicity.

In general, this thesis will use the term product-form queueing networks to denote queueing networks which can be solved by such a product-form equation. The importance of product-form solutions cannot be overemphasised, as they make solution complexity increase linearly with number of service centres in the network, rather than with the number of states in the corresponding Markov chain (which grows exponentially with number of centres).

Jackson also considered more general open queueing networks in which
1. external customer arrival rates may depend on the total number of customers in the system, and

2. the service rate (or time) of any service centre in the network may be a function of the number of customers in that centre.

This generalised system also lent itself to a (rather more complex) product-form solution. Gordon and Newell [86] considered a modification of Jackson’s queueing networks which made the network closed as opposed to open. (In fact, such closed networks could also have been analysed using the state dependent external arrival rate extension of Jackson [78]). Closed queueing networks are not driven by any external Poisson processes. Instead, the number of customers in (or population of) the network is fixed, and the single customer class considered is described in terms of its multiplicity. Each customer requests service from some centre immediately after departure from another one with probability one, hence customers never “leave” the queuing network. Gordon and Newell’s product-form equation has the form

\[ p(k_1, k_2, \ldots, k_N) = \frac{1}{G(K)} \prod_{i=1}^{N} \frac{x_i^{k_i}}{\beta_i(k_i)}, \]

where \( G(K) \) is a normalisation constant given by

\[ G(K) = \sum_{k \in A} \prod_{i=1}^{N} \frac{x_i^{k_i}}{\beta_i(k_i)}, \]

and \( x_i^{k_i} \) and \( \beta_i(k_i) \) are derived from sets of linear equations.

Buzen [48] provided a recurrent method for obtaining normalisation constants \( G(1), G(2), \ldots, G(N) \). This method is called convolution. Buzen showed that many useful performance measures could be derived from these normalisation constants, and extended the resulting recurrent algorithm to cover the case where the service rates (or times) of a service centre depend on the number of customers present at the centre.

As more extensions were made to the product-form equations of Jackson (for open networks) and Gordon and Newell (for closed networks), Baskett, Chandy, Muntz, and Palacios [14] integrated several early results, covering

- multiple customer classes;
- both open and closed customer classes (i.e. potentially mixed queueing networks);
- fixed-probability customer class changes;
- first-come first-served (FCFS) service centres with identical, load-dependent, exponential service rates for all classes, and
- preemptive-resume last-come first-served (LCFS), processor-sharing, and infinite-server service centres with load-independent, per class service times with rational Laplace transforms.
Since then, further extensions have been made to the original “BCMP-form,” e.g. the MSCCC service centre by Balbo et al. [10, 62] and Le Boudec [120, 69].

An alternative approach to obtaining performance measures by convolution, is the mean-value analysis (MVA) algorithm developed by Reiser and Lavenberg [154]. This algorithm computes mean values for main performance measures, avoiding explicit calculation of the normalisation constant $G(N)$. The MVA algorithm is intuitively appealing as it is based on easily understandable relations between basic performance measures. It also provides a basis for approximation algorithms for unfeasibly large product form networks or for non-product form networks. This thesis will be heavily based on mean-value analysis. However, it will be used in the operational setting introduced in the next section 4.4.2, rather than in the conventional stochastic setting of this section.

### 4.4.2 Operational queueing network analysis

Operational analysis provides an alternative — non-stochastic — view of analytic performance modelling.

**Operational laws**  Operational analysis is based on a number of operational laws originally identified by Buzen [49, 50] and later extended by Denning and Buzen [73]. The operational view of performance analysis contrasts the queueing theoretic view presented in the previous sec. 4.4 because:

- the variable quantities involved are exact measurements rather than stochastic variables;
- the treatment of the variable quantities is exact rather than probabilistic; and
- the quantities involved are directly observable on the operational system (if it exists).

Observing some device $i$ for a finite period of time $T$, the numbers of arrivals $A_i$ and completions $C_i$ may be measured directly, as well as the busy time $B_i$ of the device during the time period. From these, the following operational quantities can be derived:

- **arrival rate:** $\lambda_i = \frac{\text{number of arrivals}}{\text{observation time}} = \frac{A_i}{T}$;
- **throughput:** $X_i = \frac{\text{number of completions}}{\text{observation time}} = \frac{C_i}{T}$;
- **utilisation:** $U_i = \frac{\text{busy time}}{\text{observation time}} = \frac{B_i}{T}$, and
- **mean service time:** $S_i = \frac{\text{busy time}}{\text{number of completions}} = \frac{B_i}{C_i}$.

In addition,

- **visit ratio:** $V_i = \frac{\text{number of device } i \text{ completions}}{\text{number of external completions}} = \frac{C_i}{C_E}$ and
- **system throughput:** $X = \frac{\text{number of external completions}}{\text{observation time}} = \frac{C_E}{T}$.
are defined, where \( C_E \) is the number of completions made from the system of devices as a whole. Although these operational quantities are variables that may change between one observation period and the next, there are certain operational laws that must hold between operational quantities of the same observation period.

**The utilisation law** From the above definitions of device utilisation, throughput, and mean service time,

\[
U_i = \frac{B_i}{T} = \frac{C_i B_i}{T C_i} = X_i S_i, \tag{4.4}
\]

The utilisation law therefore is

\[
U_i = X_i S_i. \tag{4.5}
\]

**The forced flow law** If the operational quantities have been measured over an observation period so that

\[
A_i = C_i \tag{4.6}
\]

for device \( i \), the device satisfies the requirement of flow balance. This means that the number of jobs arriving at the device has equaled the number of jobs completing service at the device during the time period. For practical purposes, it is sufficient that the ratio

\[
\frac{A_i - C_i}{T} \tag{4.7}
\]

is low. This is usually the case if the observation period \( T \) is sufficiently long. From the above definitions of device throughput, system throughput, and visit ratio,

\[
X_i = \frac{C_i}{T} = \frac{C_E C_i}{T C_E} = X V_i. \tag{4.8}
\]

The forced flow law therefore is

\[
X_i = X V_i. \tag{4.9}
\]

**Little’s law** Little’s law [125] is one of the most commonly used results from queueing theory. If \( Q_i \) is the average number of customers present at device \( i \), and \( R_i \) is the average amount of time spent by each customer at the device per visit, the operational formulation of Little’s law is

\[
Q_i = \lambda_i R_i, \tag{4.10}
\]

If the device is flow balanced,

\[
Q_i = X_i R_i. \tag{4.11}
\]

This will become a fundamental result in the sequel. In fact, the utilisation law is just a special case of Little’s law, where the device is considered in isolation from its input queue. This is possible since Little’s law makes no assumption about the nature of the device considered.
4.4. QUEUEING NETWORK MODELLING

General response time law In fact, Little’s law can be applied to a (sub-)system of devices as well as to any part of a device. The only requirement is that the system considered is flow balanced. Little’s law for a system of devices becomes

\[ Q = XR, \]  
\[ (4.12) \]

where \( Q \) is the total number of jobs in the (sub-)system and \( R \) is the (sub-)system response time. Since at all times, these \( Q \) jobs must reside at some of its devices,

\[ Q = \sum_{i=1}^{M} Q_i \]
\[ (4.13) \]

Insertion of eqs. 4.12 and 4.11 into this equation gives

\[ XR = \sum_{i=1}^{M} X_i R_i \]
\[ (4.14) \]

Division of this equation by throughput \( X \) and using eq. 4.9 gives

\[ R = \sum_{i=1}^{M} R_i V_i. \]
\[ (4.15) \]

This is called the general response time law.

Interactive response time law In an interactive system, each job spends on average \( Z \) time units in a terminal subsystem between each visit to the central subsystem, in which \( R \) time units is spent. The total cycle time is therefore \( R + Z \), so that

\[ X = \frac{N}{R + Z} \]
\[ (4.16) \]

giving

\[ R = \frac{N}{X} - Z. \]
\[ (4.17) \]

This is called the interactive response time law. A consequence of this law, is that small changes \( \Delta X \) in throughput induce larger response time changes \( \Delta R \), due to the factor \( N \). Therefore, a common heuristic is that 10% accuracy is “acceptable” for throughputs and utilisation, while inaccuracies of \( \approx 30\% \) must often be accepted for responsivenesses.

Mean-value analysis The operational laws of the previous paragraphs is sufficient for covering open queueing networks, i.e. networks where all customer classes are described in terms of external arrival rates. The reason is that for such networks, under the flow balance assumption applied to the system as a whole, an equilibrium state can be obtained for the system by the above equations.

This, however, is not the case for closed or mixed networks, i.e. queueing networks where all or some of the customer classes are described in terms of a customer number \( N \), and possibly a think time \( Z \). The reason is that for such systems,
• the system throughputs for customer classes are dependent on service centre queue lengths, while

• the queue lengths for service centres are in turn dependent on customer class throughputs (by Little’s law, eq. 4.11).

Due to this circular dependence, exact solution seems intractable. Reiser and Lavenberg [154] solved this problem by demonstrating that the expected queue length \( A_{c,k}(\bar{I}) \) as seen by a customer of type \( c \) on arrival to a service centre \( k \) equals the average queue length for service centre \( k \) with one customer of type \( c \) removed from the system. Let \( \bar{I} \) represent the populations \( N \) and thinks times \( Z \) for all customer classes. Then

\[
A_{c,k}(\bar{I}) = Q_k(I^{(1-c)}), \tag{4.18}
\]

where \( I^{(1-c)} \) is the population obtained by removing one customer of class \( c \) from population \( \bar{I} \). Hence, the residence time for a class \( c \) customer at service centre \( k \) becomes,

\[
R_{c,k}(\bar{I}) = S_k[1 + A_{c,k}(\bar{I})], \tag{4.19}
\]

because the customer must wait for \( A_{c,k}(\bar{I}) \) other customers to receive service before being serviced itself, and because each service reception lasts \( S_k \) time units. Using these equations and the above operational laws, performance measures for the closed system can be obtained recurrently from the zero population, through all possible intermediate populations, and up to the final population.

**Queueing network requirements** The simple, operational analysis technique presented above work only for a restricted family of product-form queueing networks. Since there is no “product-form equation” involved in operational analysis, this term seems slightly misplaced. In this thesis, we will therefore call such queueing networks separable, following [119]. For all practical purposes, the concepts of product-form and separability will refer to the same, analytically tractable class of queueing networks. However, product-form will be used in the context of stochastic queueing network analyses, while separable networks will be analysed operationally.

According to [119], the following requirements must be satisfied for a multiple-class queueing network to be separable:

• *service centre flow balance;*
  As was explained in sec. 4.4.2, the number of arrivals at any service centre must equal the number of departures during the observation period.

• *one-step behaviour;*
  Only a single customer may arrive to or depart from a service centre at a time.

• *routing homogeneity;*
  The visit patterns of customers with respect to service centres are not dependent on the state of the queueing network.
• *device homogeneity*;
  The service rate $\mu_{c,k}(n,n_c)$ of class $c$ customers at centre $k$ is only allowed to vary with the total number of customers $n$ and the number of class $c$ customers $n_c$ at the centre, according to the following formula:
  \[
  \mu_{c,k}(n,n_c) = \frac{n_c}{n} \mu_{c,k}(1,1)a_k(n),
  \]
  for all $c$ and $k$, and where $a_k(n)$ is a positive constant which is fixed for $k$ and $n$.

• *homogeneous external arrivals*;
  The rates of external customer arrivals are not dependent on the state of the queueing network.

In addition, the particular solution algorithm (MVA) used in this thesis is only applicable to networks which exhibit

• *service time homogeneity*;
  The completion rate of class $c$ customers at service centre $k$ times the ratio of the total number of customers at $k$ to the number of class $c$ customers at $k$ is constant for all fixed $c$ and $k$. This is satisfied when $a_k(n) = 1$ for all $k$ and $n$ in the above eq. 4.20.

Although this requirement is a restriction on separability, the remainder of the thesis will implicitly assume that it is being satisfied by all queueing networks referred to as separable.

### 4.4.3 Bounds analysis

Sometimes, exact analysis techniques may be too costly for large closed and mixed models. In addition, the problem of obtaining accurate parameter estimates may not justify such analysis at all. For these problems, bounds analysis is an alternative which will become important in sec. 9.4 of this thesis, which will apply a technique of *asymptotic bounds analysis* developed by [133], and independently by [72].

Tighter (more narrow) bounds are obtainable by the technique of *balanced bounds* first described by Zahorjan et al. [195]. Balanced bounds analysis is based on the assumption that the resource demands $D_1, \ldots, D_k$ at each service centre $1, \ldots, K$ are the same. This is in contrast to the asymptotic bounds which make no assumptions about the queueing network at hand.

The bounds results used in this thesis are presented in appendix C in thesis notation for easy reference. Hence they are not repeated here.

### 4.5 Petri-Net Modelling

Heidelberger & Lavenberg [91] point out that the “main challenge to be faced in computer performance evaluation is that the development of the required performance evaluation methods keep pace with the explosion of new system designs brought on by rapid technological
advances.” In particular, analytic modelling techniques for distributed, parallel, and radically new computer architectures must be developed (e.g. Ajmone Marsan et al. [3] and Gelenbe [82]):

**Distributed systems** call for performance analysis techniques built into operating systems to facilitate dynamic resource allocations and rescheduling. *Optimisation* of analytic models will thus become a major topic. In addition, further work is needed on database modelling, in particular in analysing lock contention and the performance effects of database optimisation.

**Parallel systems and new architectures** call for new approaches to performance modelling. Generalised stochastic Petri-nets (GSPN) [2] seem a promising choice in many respects. Approximative analysis algorithms are sought for to handle the complexity resulting from the large state space of these systems. In particular, computable error bounds is a necessary prerequisite to make these algorithms widely applicable. The *aggregation techniques* outlined in sec. 4.6.1 is another promising alternative [44]. Techniques for validating approximations are needed. In particular, combinations of approximations should have predictable error margins [194].

### 4.5.1 Petri-nets

Basic Petri-nets will now be described in terms of their *structure, execution*, and possible *extensions*. In addition, important Petri-net terminology will be introduced.

**Petri-net structure** Petri-nets were introduced by C. A. Petri in 1962 [147] for description and analysis of concurrency and synchronisation. Hence such nets are well suited for description and analysis of concurrent and synchronised software. Petri-nets in their basic form is a graph comprising a set of *places* *P*, a set of *transitions* *T*, and a set of directed *arcs* *A*. All arcs either go from a place to a transition or from a transition to a place. The set of *input places* to a transition is the set of places from which there are arcs to the transition. Conversely, the set of *output places* from a transition is the set of places to which there are arcs from the transition.

A Petri-net can therefore be formally defined as:

- **PN =** *(P, T, A)*;
- **P =** \{p_1, \ldots, p_n\};
- **T =** \{t_1, \ldots, t_m\};
- **A_i \subseteq (P \times T) ;**
- **A_o \subseteq (T \times P) ;**
- **A = A_i \cup A_o ;**
- **A_i \cap A_o = \emptyset .**
4.5. PETRI-NET MODELLING

In a graphical representation, places are drawn as open circles, transitions as bars, and arcs as arrows. (See e.g. Fig. 11.14 of chapter 11 for an example. However, this model also uses some Petri-net extensions to be introduced later.)

**Petri-net execution** A Petri-net can be *marked* with *tokens* contained in its places. A marking is formally defined as $M = \{m_1, \ldots, m_n\}$, where $m_i$ is the number of tokens contained in place $p_i$. A marked Petri-net can therefore be formally defined as

$$ \text{PN} = (P, T, A, M_0), $$

where $M_0$ represents the *initial marking* of the Petri-net. Graphically, a token residing in a place is depicted as a dot inside the open circle representing that place. Multiple tokens in the same place can also be depicted with a number inside the circle.

Tokens are moved between places by transitions. This corresponds to *executing* the Petri-net, according to the following rules:

1. a transition in *enabled* iff it has tokens for all its input arcs in all its input places;

2. an enabled transition can *fire*, removing one token along each of its input arcs from its input places, and placing one token along each of its output arcs in its output places;

3. the firing of a transition is atomic, i.e. all tokens involved are instantaneously removed and placed, with no other transitions firing “simultaneously.”

Hence more than one transition may be enabled at the same time, but only one of them can fire. If two enabled transitions share input places, the firing of one of them may remove tokens so that the other one is no longer enabled. A set of transitions are therefore said to be in (non-trivial) *conflict* iff they all share at least one input place. For the purposes of this thesis, each transition which does not belong to any non-trivial conflict set will be regarded as forming a *trivial conflict set* on its own. In the remainder of this thesis, a “conflict set” will mean a “non-trivial conflict set” unless specified otherwise.

**Petri-net extensions** Several extensions to this basic formulation exist, such as arc multiplicity, inhibitor arcs, coloured tokens, deterministically or stochastically timed transitions, and random switches. Subsequent secs. 4.5.2 and 4.5.3 will present the extensions which are important in the thesis.

**Petri-net terminology** A marking $M'$ is *immediately reachable* from another marking $M$ iff there is a transition $t_j$ in the net which is enabled by marking $M$ and whose firing in marking $M$ results in $M'$. A marking $M^{(m)}$ is *reachable* from another marking $M^{(1)}$ iff there is an ordered set of markings $\{M^{(1)}, M^{(2)}, \ldots, M^{(m)}\}$ so that marking $M^{(i+1)}$ is immediately reachable from $M^{(i)}$ for all markings in the set.

The *reachability set* $R_{\text{PN}}$ of a Petri-net $\text{PN} = (P, T, A, M_0)$ is the set of markings $M \in R_{\text{PN}}$ that are reachable from its initial marking $M_0$. Markings from which no other markings are
reachable are *dead markings*. A marking is dead iff no transitions are enabled by it. The concepts of reachability sets and dead markings are important in Petri-net verification.

A Petri-net is *safe* iff no marking in its reachability set assigns more than a single token to any place. Accordingly, a Petri-net is *$k$-bounded* iff no marking in its reachability set assigns more than $k$ tokens to any place. Safe and $k$-bounded Petri-nets are important because they have a finite reachability sets, and are therefore isomorphic to finite state automata.

A transition $t_j \in T$ is *live* iff for all markings $M \in \mathcal{R}_P$ there is a marking $M'$ reachable from $M$ which enables $t_j$. A Petri-net is *live* iff all its transitions are live. Liveness is another important aspect of Petri-net verification, and is associated with the absence of deadlock [3].

### 4.5.2 Generalised stochastic Petri-nets

Generalised stochastic Petri-nets (GSPN) were introduced by A. Moreau Marsan, Balbo, and Conte in 1984 [2]. These Petri-nets will become important when considering distributed information systems in chapter 11. Generalised stochastic Petri-nets are based on two Petri-net extensions:

- stochastic timed transitions, and
- random transition switches.

In addition, the generalised stochastic Petri-nets used in this thesis will be coloured. Each of the three extensions will be considered in separate.

**Stochastic Petri-nets** The stochastic Petri-nets upon which this thesis is based were introduced by Molloy in 1981 [131]. Each transition in such a Petri-net has an exponentially distributed random variable associated with it. This variable represents the delay between when the transition becomes enabled and when it fires. A stochastic Petri-net can therefore be formally defined as

- $\mathbf{PN} = (P, T, A, M_0, L)$;
- $L = \{l_1, \ldots, l_m\}$,

where $l_j$ is the *firing rate* for transition $t_j$. This firing rate uniquely determines the exponentially distributed random variable associated with the transition.

Firing rates $l_j$ may be *marking dependent*. Marking dependent firing rates allow the delays associated with transitions to depend on the current marking $M$ of the net, i.e. to be marking-dependent. Hence, marking dependent firing rates are denoted as functions $l_j(M)$ of markings. Marking dependent rates are important for the aggregate analyses of sec. 4.6.1, and for the distributed information system analysis of chapter 11.

According to interpretation, places may represent activities while timed transitions represent activity termination and initiation. In that case, tokens in a place denote that an activity
is ongoing. Alternatively, timed transitions may represent the activities and tokens in places represent the prerequisites for activities to be carried out. This thesis will be based on the latter interpretation.

Just as a basic Petri-net with finite reachability set is isomorphic to a finite state automaton, a $k$-bounded stochastic Petri-net is isomorphic to a finite continuous-time Markov chain. Let $Q$ be the state-transition rate matrix (or just state-transition matrix) of this chain [182]. The state space $i \in S$ of the Markov chain corresponds to the reachability set $M \in R_{\text{Net}}$ of the stochastic Petri-net, so that $i \leftrightarrow M_i$. The transition rate from state $i$ to state $j$ in the Markov chain therefore corresponds to the transition rate from marking $M_i$ to $M_j$ in the stochastic Petri-net. Hence

$$q_{ij} = \sum_{k \in H_{ij}} l_k,$$

(4.21)

where $H_{ij}$ is the set of transitions enabled by marking $M_i$ that produces marking $M_j$ when fired. Accordingly, the marking dependent equation is

$$q_{ij} = \sum_{k \in H_{ij}} l_k(M_i),$$

(4.22)

since $M_i$ is the marking which enables transition $k$.

It is possible to show that the Markov chain produced in this way is ergodic iff the initial marking $M_0$ of the corresponding stochastic Petri-net is reachable from every marking $M \in R_{\text{Net}}$ in its reachability set. For ergodic Markov chains, the steady state probabilities $\pi$ are obtained by

$$\pi Q = \vec{0},$$

(4.23)

where $\vec{0}$ is the null vector, under the usual requirement that

$$\sum_i \pi_i = 1.$$  

(4.24)

From this probability vector, performance measures can be obtained.

Since the steady state solution is independent of initial marking, only the initial number of tokens must be specified. A stochastic Petri-net for which only steady state probabilities are important is therefore formally defined as

- $\text{PN} = (P, T, A, m_0, L),$

where $m_0$ is the initial population of tokens in the net. In this thesis, only steady state probabilities are of interest, and all Petri-nets will be specified in terms of initial populations rather than by initial markings.

**Random transition switches** Generalised stochastic Petri-nets include both timed (i.e. stochastic) and immediate (i.e. not timed) transitions. This raises the question of how to treat (non-trivial) conflict sets:
• if a set of simultaneously enabled transitions contains only timed transitions, their firing rates will determine the likelihood of each firing first: transition \( t_i, i \in H \) fires first with probability \( \frac{l_i}{\sum_{k \in H} l_k} \);

• if the set contains several timed and one immediate transition, only the immediate one can fire;

• but if the set contains more than one immediate transition — regardless of whether timed ones are present or not — the conflict must be resolved by other means.

In generalised stochastic Petri-nets, conflicts between immediate transitions are resolved by associating random switches with each immediate transition which is in a conflict set. In this thesis, all conflict sets will be non-overlapping, i.e. no transition can belong to more than one conflict set. This means that assigning random switches is particularly easy: For each (immediate) transition \( t_j \in H \) in a non-trivial conflict set, assign a switching probability \( \phi_j \) to it. The sum of switching probabilities of all transitions in a conflict set \( H \) must equal one, \( \sum_{k \in H} \phi_k = 1 \). Therefore, every transition which is in a trivial conflict set (i.e. a conflict set with only one member) has the implicit switching probability of one.

For the purposes of this thesis, a Petri-net with random switches can therefore be formally defined as

- \( \text{PN} = (P, T, A, M_0, \Phi) \);
- \( \Phi = \{\phi_1, \ldots, \phi_m\} \),

where \( \phi_j \) is the switching probability of (immediate) transition \( t_j \).

Colours A coloured Petri-net is obtained by assigning colours to its tokens. Token colours in Petri-nets may be used for the same purposes as customer classes in queueing networks.

A coloured marking is formally defined as a set \( M = \{M_1, \ldots, M_r\} \) of markings so that \( M_k = \{m_{k1}, \ldots, m_{kn}\} \), where \( m_{kj} \) is the number of colour \( k \) tokens contained in place \( p_i \). A marked Petri-net is defined accordingly as

- \( \text{PN} = (P, T, A, M_0) \),

where \( M_0 \) is the initial coloured marking of the net.

In a coloured stochastic Petri-net, firing rates \( l_j \) may be dependent on coloured markings, \( l_j(M) \). Accordingly in a coloured Petri-net with random switches, switching probabilities \( \Phi_j \) may be dependent on coloured markings, \( \Phi_j(M) \). However, only the former possibility will be needed in this thesis. I.e. all random switches will be marking independent.

As for stochastic Petri-nets, the initial marking \( M_0 \) is superfluous when only steady-state probabilities are of interest. Instead, the net can be defined as

- \( \text{PN} = (P, T, A, M_0') \);
4.5. **PETRI-NET MODELLING**

- \( M_0^* = \{m_1, \ldots, m_r\} \),

where \( M_0^* \) is the *initial colour population* of the net and \( m_k \) is its initial population of colour \( k \) tokens.

Furthermore, all stochastically timed transitions used in this thesis will have only one input place. Therefore, transition firing rates can depend not only on marking, but also on the *token colour consumed* by the transition. Hence, the colour and marking dependent firing rate of transition \( t_j \) when enabled by a token of colour \( k \) can be written \( l_{jk}(M) \).

**Generalised stochastic Petri-nets** Generalised stochastic Petri-nets comprise both stochastically timed transitions and randomly switched immediate transitions. By convention, transitions \( t_1, \ldots, t_{m'} \) are stochastically timed, while transitions \( t_{m'+1}, \ldots, t_m \) are immediate. In addition, the GSPN-models used in this thesis will be coloured.

For the purposes of this thesis, a generalised stochastic Petri-net can therefore be defined as

- **PN** = \((P, T, A, M_0^*, \mathcal{L}, \Phi)\),
- \( P = \{p_1, \ldots, p_n\} \);
- \( T = \{t_1, \ldots, t_m\} \);
- \( A_i \subset (P \times T) \);
- \( A_o \subset (T \times P) \);
- \( A = A_i \cup A_o \cap A_i \cap A_o = \emptyset \);
- \( M_0^* = \{m_1, \ldots, m_r\} \);
- \( \mathcal{L} = \{L_1, \ldots, L_r\} \);
- \( L_k = \{l_{1k}(M), \ldots, l_{mk}(M)\} \), and
- \( \Phi = \{\phi_{m'+1}, \ldots, \phi_m\} \).

The symbols used in this definition can be summarised as follows:

- **PN** is a generalised stochastic Petri-net;
- \( P \) is a set of \( n \) places indexed by \( i \);
- \( T \) is a set of \( m \) transitions indexed by \( j \);
- \( A \) is a set of directed arcs partitioned into input and output arcs with respect to transitions;
- \( M_0^* \) is an initial population of \( k \) colours;
- \( m_k \) is the initial population of colour \( k \) tokens;
• \( \mathcal{L} \) is a coloured set of firing rates;
• \( L_k \) is a set of firing rates for colour \( k \) tokens;
• \( l_{jk} \) is the firing rate of transition \( t_j \) with respect to colour \( k \) tokens;
• \( l_{jk}(M) \) is the firing rate of transition \( t_j \) with respect to colour \( k \) tokens depending on marking \( M \);
• \( \Phi \) is a set of random switches;
• \( \phi_j \) is the probability of immediate transition \( t_j \) firing when enabled.

4.5.3 Generalised stochastic Petri-net analysis

A \( k \)-bounded generalised stochastic Petri-net can be represented by a finite continuous-time Markov chain. If the initial marking \( M_0 \) is reachable from all markings \( M \in \mathcal{R}_{\mathcal{PN}} \) in its reachability set, this Markov chain is also ergodic.

If for some marking \( M_i \), at least one immediate transition is enabled, that (or one of those) transitions will fire. Hence no time will be spent in marking \( M_i \), and the corresponding Markov-chain state \( i \) becomes a vanishing state. Markings \( M_j \) for which no immediate transitions are enabled, will correspond to tangible states \( j \).

Disregarding the concept of time [3], consider the embedded Markov chain within the stochastic process [182]. Partition the state space \( S \) into sets of tangible \( T \) and vanishing \( V \) states, \( S = T \cup V, T \cap V = \emptyset \). The transition-probability matrix \( U \) can now be defined for the embedded Markov chain as

\[
U = \begin{bmatrix}
U_{tt} & U_{tv} \\
U_{vt} & U_{vv}
\end{bmatrix}
\] (4.25)

where matrices \( U_{tt}, U_{tv}, U_{vt}, \) and \( U_{vv} \) represent transitions from tangible-to-tangible, tangible-to-vanishing, vanishing-to-tangible, and vanishing-to-vanishing states respectively.

Elements of matrices \( U_{tt} \) and \( U_{tt} \) correspond to firings of timed transitions. Elements \( u_{ij} \) of these matrices can be determined as follows: Since \( i \) is a tangible state, only timed transitions are enabled by the corresponding Petri-net marking \( M_i \). Again let \( H_i \) be the set of (timed) transitions enabled by this marking, and let \( H_{ii} \subseteq H_i \) be the subset of (timed) enabled transitions in this marking whose firing produces marking \( M_{ii} \). Furthermore, let \( K_{ij} \) be the set of token colours with which timed transition \( t_j \in H_i \) can fire in marking \( M_i \), and let \( K_{ij} \) be the set of token colours for firing of \( t_j \in H_{ii} \) accordingly. Then

\[
u_{ij} = \frac{\sum_{j \in H_{ii}} \sum_{k \in K_{ij}} \sum_{l \in K_{il}} l_{jk}(M_i)}{\sum_{j \in H_i} \sum_{k \in K_{ij}} l_{jk}(M_i)}. \tag{4.26}
\]

On the other hand, elements of matrices \( U_{vt} \) and \( U_{vv} \) correspond to firings of immediate transitions. They are derived from the switching probabilities of the associated generalised stochastic Petri-net,

\[
u_{ij} = \frac{\sum_{j \in H_{ii}} \phi_j}{\sum_{j \in H_i} \phi_j}, \tag{4.27}
\]
4.6. OTHER PERFORMANCE MODELLING TECHNIQUES

since \( H_i \) is again the set of transitions that are enabled by marking \( M_i \), and \( H_i^\perp \subseteq H_i \) is the subset of those transitions that produce marking \( M_i^\perp \) when fired.

The *stationary probability distribution* \( \bar{\pi} \) can now be obtained from transition probability matrix \( \mathbf{U} \) by

\[
\bar{\pi} = \bar{\pi}\mathbf{U}.
\]  

(4.28)

A more efficient solution method [3] is obtained by removing all vanishing states by pre-analysis before solving the resulting correspondent to eq. 4.28. From probability distribution vector \( \bar{\pi} \), performance measures can now be obtained by reintroducing the concept of time based on the average time spent per visit to each state.

4.6 Other Performance Modelling Techniques

In addition the performance modelling and analysis techniques presented in secs. 4.3, 4.4 and 4.5, several other useful techniques exist for performance models exploitation. Two of them will be surveyed here: Performance model aggregation (sec. 4.6.1) and sensitivity analysis (sec. 4.6.2).

4.6.1 Performance model aggregation

In addition to generalised stochastic Petri-nets, a performance modelling technique that will become important in chapter 11 on distributed information systems, is that of *aggregation*. Aggregation was first introduced by Chandy, Herzog, and Woo [59] as an analogue to “Norton’s Theorem” for electronic circuits. In electronics, Norton’s theorem states that any linear network (of current and voltage sources and resistances) can be replaced by a simpler circuit of just one resistance and one current source in parallel [128]. For queueing networks, the corresponding theorem [59] states that under certain assumptions, a network of service centres can sometimes be replaced by an equivalent, single centre.

**Performance models and submodels** In a more general setting, i.e. covering both queueing networks and generalised stochastic Petri-nets, performance model aggregation can be outlined as follows: A performance model \( M \) somehow represents the dynamic behaviour of a system. For this purpose, the model must comprise some dynamic components \( c \in M \) for which *jobs* compete.

**Example**

The performance models of this thesis will be queueing networks or generalised stochastic Petri-nets, as described in secs. 4.4 and 4.5. Accordingly, the dynamic components will be service centres and timed transitions, while jobs will be customers and tokens. □

Aggregation is based on associating a component \( c \) — called an *aggregate* — in performance model \( M \) with another performance model \( M' \) representing component \( c \) so that \( M' \) decomposes \( c \). In this thesis, \( M' \) will be called a *performance submodel* of model \( M \). The aggregate
component and its corresponding submodel must have the same external interface, i.e. they must look the same to the rest of performance model $M$.

Jobs entering or exiting component $c$ correspond to external arrivals or departures of jobs in performance submodel $M'$. Hence, certain consistency requirements between component $c$ and submodel $M'$ must be satisfied: For each job type that can enter (and hence exit) component $c$, there must be a similar job type entering (and exiting) submodel $M'$.

There are typically three different way of establishing such a performance model/submodel pair:

1. by isolation: submodel $M'$ is a part of model $M$ that has been isolated in a submodel for aggregation purposes;
2. by decomposition: submodel $M'$ was created as a refinement of component $c$, and
3. by hierarchical modelling: models $M$ and $M'$ are autonomous but inter-dependent models which represent different levels of processing in the same system.

Of course, the resulting system of performance models and submodels will not differ in the three cases. The aggregations of this thesis will be of the third, “hierarchical modelling” type.

**Motivation for aggregation techniques** Several reasons exist for using aggregation in performance modelling and analysis:

**repeated analyses:** If the same performance model $M$ must be repeatedly analysed with changes only to a minor part $M_1 \subset M$ of the model, the unchanged part $M_2 = M \setminus M_1$ can be analysed once and for all by aggregation.

**repeated subsystems:** If a performance model $M$ has repeated subsystems, i.e. is on the form $M = M_1 \cup M_2 \uplus M_2 \uplus \cdots \uplus M_2$, the repeated subsystem $M_2$ need only be analysed once by aggregation.

**structured modelling:** Beilha [17] points out that performance models become difficult to manage as computer and information systems grow large. As a result, performance models should be subject to the same structuring principles that are applied in other branches of computer science, such as vertical layering and horizontal modularisation. The resulting structured performance models can be analysed by aggregation.

**heterogenous modelling:** [17] also points out that structured performance models solved by aggregation allow different solution techniques to be applied to different performance submodels.

The two first cases apply aggregation techniques for more efficient solution of “flat” performance models. Hence performance submodels are typically created “by isolation.” The earliest work on aggregation was of this kind [59].

In the latter two cases, aggregation is used to solve hierarchical models, and performance submodels will usually be established by decomposition or as a result of hierarchical modelling. In this thesis, aggregation will be used to for “structured modelling” of dynamic, composite software and hardware systems.
4.6. OTHER PERFORMANCE MODELLING TECHNIQUES

Solving by aggregation In the above two-level model of performance model \( M \) and sub-model \( M' \) decomposing \( c \), the performance of model \( M \) depends on the performance of its component \( c \). The performance of component \( c \) in turn depends on the performance of sub-model \( M' \). The performance of sub-model \( M' \), however, is workload dependent. This workload is in turn determined by the average number of jobs that reside in component \( c \) of model \( M \).

To overcome this circularity, an aggregate must be constructed for performance submodel \( M' \) by pre-analysis. The resulting aggregate component \( c \) will then be inserted into performance model \( M \) to represent submodel \( M' \). The most popular form of aggregate is the exponential flow-equivalent service centre (FESC). Other slightly more sophisticated aggregate types exist, which sometimes provide more accurate analysis results. (E.g. [45] notes that in the queuing network case, an aggregate may be composed of two service centres rather than one.) Only flow-equivalent service centres will be considered in this thesis.

Pre-analysis usually implies solving submodel \( M' \) for all possible populations \( \bar{N} \) that can arise for component \( c \). Submodel \( M' \) can be solved by a technique of short-circuiting in which any job of type \( j \) which leaves the submodel is immediately reentered into the submodel as another job of type \( j \). The average residence time \( r_j \) between leaving and reentering the submodel are collected for all job types \( j \). These residence times \( r_j(\bar{N}) \) will later be used to represent the performance of component \( c \) for the population \( \bar{N} \) in question. The largest population that must be considered is the one in which all jobs of performance model \( M \) reside in component \( c \) simultaneously. Hence aggregation requires that the maximum population \( \bar{N}^* \) of performance model \( M \) is known. For queueing networks, the maximum population is known whenever the network is closed. Aggregation of open or mixed queueing networks introduces approximation errors because open customer classes must be approximated by closed ones, or because a maximum number of simultaneously present customers must be assumed (i.e. limited load-dependent behaviour [119]). For Petri-nets, the maximum population is always known from its initial marking or population.

Several pre-analysis techniques are available, ranging from simulation, through numerical evaluation of Markov chains, to exact and approximate analysis of queueing networks. Exact analyses of product-form or operational queueing networks is convenient, since the mean-value analysis technique automatically computes residence times for all intermediate populations due to its recurrent nature.

After aggregation of submodel \( M' \), performance model \( M \) can be solved with component \( c \) as a load-dependent flow-equivalent service centre (FESC). In this way, circularity is broken, since component \( c \) represents the performance of submodel \( M' \) for all possible workloads on \( M' \). In queueing networks, there are several ways of representing a FESC. The two most common ones are [119]:

- an FCFS service centre with a separate queue and an associated server for each customer class with exponentially distributed service rates whose mean service times \( r_j(\bar{N}) \cdot n_j \) for job type \( j \) depend on the population \( \bar{N} = \{n_1, \ldots, n_C\} \) at the centre, and

- an infinite-server service centre with exponentially distributed service rates whose mean service times \( r_j(\bar{N}) \) for job type \( j \) depend on the population \( \bar{N} \) at the centre.

[45] points out that infinite-server centres are slightly more versatile than FCFS ones, since
their distribution functions can be changed without violation of product-form.

In Petri-nets, an aggregate component is represented as a timed transition with marking dependent firing rates. The firing rates are inverse submodel residence times, and the marking dependency accounts for the workload dependency of the submodel. Since the “jobs” present at a timed transition are represented as tokens in its input places, the firing rates need only be marking dependent with respect to these places.

In the generalised stochastic Petri-nets of this thesis, each timed transition only has one input place, as mentioned in sec. 4.5.2. Also, firing rates may be dependent on the colour of the token consumed. Hence the marking dependent firing rate becomes $1/r_j(\bar{N})$ where $j$ is the token colour corresponding to job type $j$, and $\bar{N}$ represents the token population of its single input place.

**Accuracy of performance model aggregation** In some cases, aggregation yields exact results. In general, an aggregate component features “product-form” iff the following equality is satisfied [45]:

$$r_i([n_1, \ldots, n_C]) \cdot r_j([n_1, \ldots, n_i - 1, \ldots, n_C]) = r_j([n_1, \ldots, n_C]) \cdot r_i([n_1, \ldots, n_i - 1, \ldots, n_C]).$$

This is always the case for exact solution of product-form queueing network submodels, but may not hold when another pre-analysis technique is used. If both performance model $M$ and its aggregate component $c$ feature product-form, aggregation will be exact concerning marginal probabilities of the performance model.

[45] notes that performance submodel aggregation is mostly an approximate solution method which involves some approximation error. In general, the following model/submodel features encourage aggregation:

- product-form (i.e. separable) performance models or submodels;
- weak interactions between performance model and submodel, and
- small dependency of performance submodel on model,

while the following feature should discourage aggregation:

- performance submodels with multiple, dependent entry and exit points, and
- dependency of performance submodel behaviour on model.

### 4.6.2 Performance model sensitivity analysis

Let performance model $M$ have input parameters $I_M$ and output parameters (performance measures) $O_M$. In this thesis, a *sensitivity* means the derivative of a performance measure with respect to a performance parameter [153]. *Sensitivity analysis* is a therefore technique
for determining for every parameter \( x_o \in O_M \) how much it depends on any input parameter \( x_i \in I_M \).

Sensitivity analysis is useful for:

**testing robustness:** determining the extent to which questionable input-parameter assumptions cast doubt on the performance measures [119];

*or more precisely:* obtain approximate confidence intervals for the performance measures given confidence limits on the inputs [30, 77];

**optimisation:** optimisation strategies exist which use numerical sensitivities to detect optimal points in the input parameter space with respect to some cost-benefit function [77, 30];

**performance model validation:** if performance measures are extremely sensitive to some parameter, the model's validity is questionable [153];

**performance model complexity:** if performance measures are extremely insensitive to some parameter, the model may be unnecessarily complex [153];

**performance interpolation:** sensitivities would improve results when predicting the performance of a system under light and heavy workload (and possible some additional workloads), and obtaining intermediate performance measures by interpolation [153];

**parameter capture:** input parameters with strong impact of performance measures are the ones that must be estimated with most care [77, 30], and

**bottlenecks:** detect performance bottlenecks in the system [30].

This thesis will mostly apply sensitivity analysis for “parameter capture” and (design) “optimisation” purposes, as will be explained in chapter 10.

In its “brute-force” form sensitivity analysis can take the form of [119]:

- solving the performance model many times with varying input assumptions and comparing the results to check the robustness of the assumption, and
- obtain bounds on the assumption by evaluating the model with extreme values of the assumption.

Also, numerical sensitivity estimates can be derived by interpolation of performance measures obtained with different input parameters. However, more sophisticated — precise and computationally cheaper — approaches exist. In what follows, we will consider techniques for sensitivity analysis of simulation models, Markovian models, generalised stochastic Petri-nets (which are of course based on Markovian models), as well as queueing networks.

**Sensitivity analysis of simulation models** Apart from the obvious approach of running a simulation several times with different input parameters, there are two main approaches to obtaining sensitivities *in a single run*:
The (infinitesimal) perturbation analysis (PA) introduced by Ho et al. [98] works by \textit{actively changing} (perturbing) the input parameters slightly during a simulation run. Sensitivities are statistically obtained from the observed changes in performance measures. An overview of the method is given in [181].

The \textbf{likelihood method (LM)} discovered independently by Glyn [84], Reiman & Weiss [153], and Rubinstein [158] works by \textit{passively observing} changes in performance measures as input parameters drift slightly ("by themselves") during the simulation run. Again, sensitivities are statistically obtained from the observed changes in performance measures.

However, the two methods are fundamentally different in both theory and practice. They are not always applicable to the same types of systems, and when they are, they will sometimes provide different kinds of sensitivities. Hence, the two are to a large extent complementary and difficult to compare. [153] notes that the perturbation method is probably more accurate on theoretical grounds. On the other hand, the likelihood method is simpler and may be more widely applicable.

\textbf{Sensitivity analysis of Markovian models} For Markovian models, i.e. models represented by Markov chains as in sec. 4.5.2, sensitivities can be easily obtained for both stationary and transient performance measures. Only stationary measures will be considered here.

Let transition rate matrix $Q$ and state-probability vector $\pi$ be functions $Q(\lambda)$ and $\pi(\lambda)$ of some performance parameter $\lambda$. From sec. 4.5.2, for an ergodic Markov chain, the \textit{steady-state probabilities} $\bar{\pi}$ was obtained by (eq. 4.23)

$$\bar{\pi}Q = \bar{0},$$

where $\bar{0}$ is the null vector, under the usual requirement that (eq. 4.24)

$$\sum_i \pi_i = 1.$$  \hfill (4.30)

Differentiation of eqs. 4.29 and 4.30 gives [30, 77]

$$\frac{\partial \bar{\pi}}{\partial \lambda} Q + \bar{\pi} \frac{\partial Q}{\partial \lambda} = \bar{0}$$ \hfill (4.31)

and

$$\sum_i \frac{\partial \pi_i}{\partial \lambda} = 0.$$ \hfill (4.32)

The sensitivity \textit{vector} $\bar{\sigma} = \frac{\partial \bar{\pi}}{\partial \lambda}$ therefore becomes

$$\bar{\sigma}Q = -\bar{\pi} \frac{\partial Q}{\partial \lambda}$$ \hfill (4.33)

$$\sum_i \sigma_i = 0,$$ \hfill (4.34)

where $\bar{\pi}$ was determined by eqs. 4.23 and 4.24, and $\bar{\sigma} = (\sigma_1, \ldots, \sigma_n)$. \hfill [77] notes that since the coefficient matrices of eqs. 4.23 and 4.33 are the same, $Q$, the approach chosen to determine $\bar{\pi}$ will also be useful for determining $\bar{\sigma}$. Both Gauss-elimination and iterative methods are applicable. \hfill [30] notes that iterative approaches are safe and more practical when the matrices involved are large and sparse.
Sensitivity analysis of generalised stochastic Petri-nets  The only unknown in eq. 4.33 is $\frac{\partial Q}{\partial \lambda}$. In sec. 4.5.2, state-transition matrix $Q$ was derived from the stochastic Petri-net in an efficient manner. [77] addresses the question of whether a similar, efficient generation method exists for $\frac{\partial Q}{\partial \lambda}$.

Eq. 4.22 of sec. 4.5.2 derived marking dependent transition rates straightforwardly as

$$q_{ij} = \sum_{k \in \mathcal{H}_{ij}} l_k(M_i),$$  \hspace{1cm} (4.35)

since $M_i$ was the marking which enables transition $k$. Hence

$$\frac{\partial q_{ij}}{\partial \lambda} = \sum_{k \in \mathcal{H}_{ij}} \frac{\partial l_k(M_i)}{\partial \lambda}.$$  \hspace{1cm} (4.36)

[77] notes that this corresponds to replacing all rates depending on $\lambda$ with an expression of how they are derived from $\lambda$. Of course, transition rates $q_{ij}$ not depending on $\lambda$ all become zero.

Similar techniques exist for generalised stochastic Petri-nets also (e.g. Ciardo et al. [65]), but they will not be elaborated upon here. The SPNP tool [64] provides sensitivity analysis of GSPN models.

Sensitivity analysis of queueing networks  Liu & Nain [126] have presented sensitivity results for open, closed, and mixed stochastic, product-form queueing networks. Explicit formulas are provided for the derivatives of performance measures of networks featuring product-form. For such networks, Liu & Nain show that derivatives of expectations of any function of the state of the queueing network with respect to any input parameter are expressible in terms of state variables such as queue lengths. These derivatives are in turn used to study the monotonicity/non-monotonicity of performance measures. In particular, it is demonstrated that throughputs are not in general monotonic with input parameters in closed and mixed queueing networks.

Although efficient simulation-based sensitivity analysis techniques exist, the simplicity, accuracy, and ease of derivation of analytic sensitivities indicate the superiority whenever available. Unfortunately, Liu & Nain’s results will not be directly applicable since this thesis will, in general, be based on operational analysis of separable queueing networks. However, secs. 8.3.7 and 10.3.4 point out that the framework presented in part III can readily be tailored to apply product-form performance models also.

4.7 Performance Modelling Tools

Since the advent of powerful, graphics workstations, a number of integrated performance evaluation tools have been developed. These tools all focus on performance modelling, but some provide additional support for performance measurement, workload modelling, and auxiliary functions. Some major components of an integrated environment for computer performance evaluation are depicted in fig. 4.1. This figure is strongly inspired by the IMSE toolset which will be described in sec. 4.8.
The BGS Systems toolset includes the BEST/1 tool for performance prediction [24]; the CAPTURE tool for performance report generation [25]; the Applications Planner for workload characterisation of existing software systems [27], as well as the CRYSTAL tool for software performance engineering [26].

Versions of the tools also exist for the IBM MVA and VM operating system. However, the VM versions of the tools are not interfaced with CRYSTAL. In addition, BEST/1-SNA and CAPTURE/SNA versions are available for network capacity management and planning.

BEST/1 predicts the performance consequences of modified computer systems, increased load from existing applications, as well as introductions of new software. In particular, this means that existing and projected software can be assessed together. Based on the work of Buzen, the BEST/1 tool was one of the first commercially available queueing network packages. Partial models provided by the CAPTURE, Applications Planner, and CRYSTAL tools are combined and analysed by the BEST/1 tool. It is also possible to use BEST/1 with none or only a few of these tools, in which case the missing partial models must be supplied manually.

CAPTURE produces performance reports based on measurements or accounting logs provided by operating system specific software. In particular, the CAPTURE tool provides resource consumption results in terms of “user groups.” It can generate a computer performance model which is used by BEST/1. Also, measurements can be stored in an INFO/BASE database of experience data.

Applications Planner produces performance reports of specific major applications, such as PROFS, FOCUS, SQL/DS. The tool generates performance models of these applications which can be used by BEST/1.

CRYSTAL is a tool for performance modelling of projected applications early in the development. The tool is interfaced with libraries of information about the performance
overhead induced by specific major target platform subsystems, such as the CICS data communication system and the IMS and DB2 database systems. In addition, site specific overhead descriptions can be created. For IBM MVS systems, CRYSTAL models of projected applications can be used by the BEST/1 tool for performance analysis.

INFO/BASE maintains an experience database of performance measurement reports generated by the CAPTURE tool. This database can be used for trend analysis and workload forecasting by time-series analysis.

In general, the aim of the CAPTURE and Applications Planner tools is to minimise the manual effort required to build a performance model. Typically, the performance analyst only intervenes to identify user groups and calibrate the resulting combined BEST/1 model. Thereafter, the combined model is used interactively to answer “what-if?” questions associated with capacity management and planning.

The CRYSTAL tool will be commented on in the following sec. 5.6.

The HIT environment developed at the University of Dortmund by Beilner et al. [16, 15], supports horizontal and vertical structuring of performance models. Different submodels (called model “components” in HIT) can be solved using different solution techniques, ranging from simulation to exact analysis, and are combined using aggregation techniques. Hence heterogeneous modelling is encouraged.

The aim of HIT is to make performance modelling easily available to system developers. Hence HIT shares many of its goals with the work of this thesis. [95] points out that one benefit of performance models created by system developers themselves is elimination of the communication overhead imposed by the performance analyst.

The horizontal and vertical structuring of HIT models [17] is motivated by the rapidly increasing technological progress which make computers and software more complex, while performance requirements at the same time are becoming tougher. As a result, the performance of more and more complex hardware and software systems must be predicted during development, and the corresponding performance models therefore tend to become unmanageable. To solve this problem, the HIT environment supports creation of performance models that are structured according to conventional principles of structured programming and design (e.g. Dijkstra [74]). The two principles focused on are:

- **Hierarchies** (i.e. vertical structuring) of performance submodels at different levels which cooperate so that level i performance submodels (HIT components) will be realised in terms of services provided (the next lower) level $i-1$, and will themselves provide services to (the next higher) level $i+1$, and

- **Modularisation** (i.e. horizontal structuring) in which performance submodels (HIT-components) at the same level i only interact through well-defined interfaces which should be kept as small as possible, and which are defined at the next higher level $i+1$.

The heterogeneous modelling approach of HIT [17] is motivated by the large variety of available model solution techniques. These techniques are often based on different modelling languages,
and application of any of them requires considerable specialised knowledge. Furthermore, different solution techniques are appropriate for different types of systems, and for different stages of development. E.g., exact analysis techniques are usually favourable for small models early in development, while large, detailed models later in development call for simulation. As a result, there is a need to make several different solution techniques available through the same performance modelling language. Furthermore, higher-level and lower-level submodels (components) should not be restricted to the same solution technique when making an overall performance prediction.

HIT models are specified either textually through the HILANG language [95], or through a graphical user-interface [96]. In other words HIT supports:

- textual or graphical specification of dynamic, discrete-event, stochastic system models, as well as
- performance analysis of these models through simulation or exact or approximate analysis techniques.

Also, HIT facilitates:

- specification and execution of experiments to be carried out on models, and
- a modelling base supporting storage and retrieval of models, model components, experiments, and analysis results.

Although originally designed for computer performance evaluation, HIT models may also represent "similar" systems for communication and office systems, transport and logistic systems, as well as other dynamic, discrete-event systems [95]. HIT should also be applicable to application software and organisation performance modelling.

The SIMMER environment was developed as a UK Alvey project [104]. The SIMMER environment provides a common user-interface and a shared object database, as well as tools for discrete-event simulation and system description. Since the SIMMER environment was one of the starting points of the IMSE project, it will not be discussed further.

The IMSE environment incorporates many of the ideas of the SIMMER and HIT environments, as well as the QNAP2 [151], GreatSPN [61], and Workser [55] tools. The IMSE environment will be introduced in the subsequent sec. 4.8.

4.8 The IMSE Environment

This chapter introduces the IMSE environment [99, 149] for performance evaluation, with particular emphasis on its sp tool for hierarchical workload modelling (Hughes [103]). The presentation will again be illustrated through the bank example of sec. 3.3.
4.8. THE IMSE ENVIRONMENT

4.8.1 An IMSE overview

4.8.1.1 IMSE modelling

The IMSE is an integrated modelling support environment for performance evaluation. It focuses on

1. availability of state-of-art performance evaluation methods through easy-to-use graphical interfaces, and

2. integrating and supporting the performance evaluation methods through a common set of utilities.

The IMSE builds on existing performance evaluation tools for workload derivation [55], workload modelling [101], queueing networks [151], Petri-net models [7], and special-purpose simulation [28].

The IMSE comprises three sets of diagrammatic tools:

1. the system description level (SDL) tools support derivation of workload models from accounting logs and performance measurements (the WAT tool) [53, 54] and hierarchical modelling of hardware and software systems (the sp tool) [129];

2. the performance modelling level (PML) tools support performance modelling based on queueing networks (the QNET tool) [75, 76], Petri-net models (the PNT tool) [8, 83], and special-purpose DEMOS simulation (the PIT tool) [11, 12], and

3. the environmental tools (ET) support execution of static and dynamic performance models (the Executor), animation of model executions (the Animator) [66], automated support for planning and performing experiments (the Experimenter) [94, 93], and generation of reports from experiments (the Reporter).

The IMSE tools share a support environment containing a graphical user interface system (the UIS) [187] and a common object management system (the OMS) [185].

Example

The IMSE environment is depicted in fig. 4.2, separating the support environment and environmental tools from system and performance modelling. The open-endedness with respect to SDL and PML tools allows additional system or performance modelling tools to be included. The PrM tool that will be introduced later in this thesis can be regarded as such an additional IMSE SDL tool. □

4.8.1.2 IMSE methodology

The IMSE (and hence the sp) methodology is open-ended, but use of the IMSE for software performance engineering is likely to follow a generic pattern. The performance of the projected
software system is predicted by the Experimenter tool through an IMSE experiment run. Typically, the experiment is taken from a library of predefined experiments stored in the OMS. These experiments will be *parametric* with respect to

1. the models to be analysed by the SDL and PML tools;
2. which performance measures to collect and analyse, and
3. the kind of thesis produced by the Reporter tool.

Secs. 8.2 and 12.3.4 will demonstrate how an *sp* module and corresponding module implementation is automatically derived from an annotated PrM specification. An information system performance study will initially analyse this and other static workload models to produce the workload under which the dynamic performance models are to be run. The basic framework
assumes separable queueing network models, but any IMSE performance model can be used in principle. The Experimenter selects the performance measures to be derived, invokes the Executor tool, and aborts it in case a simulation run goes astray. The performance measures produced are available for statistical interpretation by the Experimenter. The interpreted analysis results are input to the Reporter which produces an experiment report of text and graphics according to a library of predefined, parametric report formats stored in the OMS.

4.8.2 An sp overview

The sp tool of the IMSE is particularly important for software and information system engineering purposes. The sp tool has been developed by Hughes [100, 101].

4.8.2.1 sp modelling

An sp model is a hierarchical performance specification of a software and hardware system. The hierarchy forms an sp module hierarchy which is a DAG (directed acyclic graph) of sp modules. An sp module is a performance abstraction of some software or hardware component.

Example

An sp model for the bank example is shown in fig. 4.3. This and subsequent sp interfaces have been created using the sp tool of IMSE [129]. The bank has three major software systems for customer transaction, accounting, and employee payroll, in addition to a smaller one for system administration.

Top-level PrM models of the Customer transaction and Accounting systems have been given in figs. 3.5 and 3.6. These software systems run on a minicomputer comprising a CPU, a disk, and a number of terminal lines. The former four modules are non-primitive, while the latter three are primitive.

An sp module definition for the Customer transaction application of fig. 3.5 will be shown in fig. 12.18. □

A module provides sp operations to the modules above it in the hierarchy. Non-primitive modules thus use operations provided by the modules beneath them for communication, processing, or memory access. The workload of a non-primitive module is represented as an sp module implementation which can be thought of as a matrix mapping operations provided by the module onto the operations it uses. The sp implementation for the customer transaction software system is shown in fig. 12.19. An sp module and corresponding module implementation becomes part of a static workload specification for the software system they represent.

The performance of the primitive hardware modules will later be represented as an IMSE performance model.
4.8.2.2 \textit{sp} methodology

An \textit{sp} module may have several implementations corresponding to different algorithms applied to provide its operations. In fact, two aspects of module implementations are represented in \textit{sp}: A \textit{compactness} specification represents the implementation of a memory mapping by defining how many data elements of a memory management subnode that maps onto a single data element of the node itself. A \textit{work complexity} specification determines how many subnode operations must be invoked for each invocation of one of the operations a node provides. However, this thesis will only consider work complexities. Based on modules and
implementations, the \textit{sp} tool provides two kinds of performance analysis, \textit{storage analysis} and \textit{workload analysis}. Only workload analysis is needed in this thesis, and sec. 9.3.3 will demonstrate it for a simplified composite modelling approach inspired by \textit{sp}. Storage analysis is based on the concept of compactness specifications for modules, mentioned in the previous footnote.
Chapter 5

Performance Engineering of Software Systems

This chapter will treat the state of the art in the information system performance engineering field, and in the wider field of software performance engineering. Since the thesis is oriented towards performance *modelling* of information systems, the chapter will focus on information system performance modelling, as opposed to tools and methodology.

The last two chapters 3 and 4 have defined information system engineering as “the practice of developing and maintaining computerised information systems” and computer performance evaluation as “the practice of assessing the performance of computers.”

The term “information system performance engineering” was first defined in sec. 1.1 based on the problem statement of the thesis. It can now be alternatively defined as “the practice of assessing the performance of information systems throughout development and maintenance.” This definition is slightly more general than the earlier one, because the term “assessing” implies more than “predicting and improving.” The reason is that parts III and IV of this thesis — although still focused on prediction and improvement — will also wider issues such as managing and planning for performance.

It is emphasised that the term “information system performance engineering” is only introduced for the purpose of this thesis in order to specialise the more general terms of “software performance engineering” and “software performance modelling.” However, the two are already well-established, and the computer science and informatics terminologies are crowded with buzzwords as it is.

5.1 Software Performance Engineering

Smith [170] points out that performance was an important issue in the early days of computing. Computing hardware was expensive compared to manpower cost, and the space and time required by programs had to be carefully managed to fit them on small machines. Although hardware power grew at a rapid rate, it was continuously outgrown by software demand.
In the early days of computers, the space and time required by programs therefore had to be carefully managed in order to fit them on small machines. Faster and better hardware that became available, did not eliminate performance problems. Instead, it made more complex software systems feasible, and programs grew into systems of programs. For software systems with strict performance requirements — i.e., embedded systems or real-time systems — simulation models were sometimes being made in order to control performance during development. Zurcher & Randell [196], Parnas [146], and Graham et al. [87] presented such early approaches to software simulation, although Parnas did not consider performance explicitly. Theses models were detailed, and creating and solving them was time consuming. Thus they were cost-efficient only in large development projects with very strict performance requirements. [196] points out that one major problem with this approach was the delay between design decisions and updated performance models. As a consequence, performance predictions were seldom available when needed.

**Fix-it-later** In summary, performance assessments during design was infeasible for all but a few kinds of projected software systems. Instead an alternative approach, the “fix-it-later” [167, 18], strategy came to dominate in the software development field. It advocated concentrating on program correctness and ease of modification, postponing performance considerations to the integration and testing phase — if needed. The “fix-it-later” approach has already been discussed in detail in sec. 2.3. It is no longer acceptable for reasons explained there.

![Diagram](image)

**Figure 5.1:** Three major breakthroughs in the development of software performance engineering (SPE): Analytic hardware performance models (a), analytic software performance models (b), and analytic software and hardware models combined.

According to Smith, software performance engineering then evolved through three break-
throughs in computer performance modelling:

1. analytic modelling of computer hardware;
2. analytic modelling of computer software, and
3. combining the two.

Each breakthrough will be considered in separate.

The 1st modelling breakthrough was the introduction of fast analytic approaches to computer performance modelling, as described in sec. 4.4. Most notably, efficient algorithms for queueing network analysis were becoming available, e.g. Buzen in 1973 [48] and Reiser & Lavenberg in 1980 [154]. The major benefit from these algorithms was that computer performance models could now be analysed interactively. Later improvements in the direction of less restricted analytic models, as well as effective approximation techniques also contributed in this direction. The corresponding view of performance modelling is illustrated in fig. 5.1a.

Since this approach did not support establishment of workload models for projected software systems, early analytic models had most impact of the related task of capacity planning. Sometimes although, they would be used for feasibility analysis, from coarse estimates of the workload of the projected software. More precise studies were infeasible, since workload characteristics were not available.

The 2nd modelling breakthrough was the introduction of analytic software models:

Software execution graphs were introduced by Smith [167] and will be extensively described in sec. 5.3. Software execution graphs are mainly oriented towards practical performance engineering of software systems. With such graphs, estimates of resource requirements can be made and performance metrics calculated. Thus best, worst, and average resource requirements are obtained. Best-case response time estimates can also be obtained, but since resource contention at the hardware level was not represented, they do not yield precise values for predicted response times.

Computation structures were introduced by Booth et al. “to cover the performance costs appropriate to software modelling.” The associated modelling approach allows both serial and parallel computations, and is open to evaluation of both space and time parameters. Possible uses of computation structures include software performance modelling [162], modelling of distributed systems [36], performance measurement [38], measurement instrumentation [34], and software design [37, 35]. A more detailed account of computation structures will be given in sec. 5.4.

Database transaction design was studied by Oftedahl & Solvberg [135, 177]. The transaction software is modelled as a Markov chain of routing probabilities between software processes. Furthermore, each software process has a certain probability of accessing each realm of the database. As a result, probabilities of switches between database realms are made available to the physical database designer. One major advantage of Oftedahl
and Sölvberg's approach is the provision of analytic sensitivity analysis of transaction specifications.

**J.W. Sanguinetti** introduced a methodology for assessing the performance of operating systems [159] as well as other complex software systems [160] during design. The approach combines the program process modelling language (PPML) introduced by Riddle [155] with message transfer expressions (MTE) annotated with timing and branching parameters.

The approach followed by these early attempts is summarised in fig. 5.1b.

**The 3rd modelling breakthrough** combining the analytic software models with analytic computer performance models. The resulting combined approach which is the basis for the rest of this section is shown in fig. 5.1c.

### 5.2 Software Performance Engineering Methods

By 1980 these techniques were established, and a few combined commercial tools were available (e.g. the CRYSTAL tool mentioned in chapter 4.7). Through practical experience [174, 4, 19], a methodology for software performance engineering was gradually established [173]. The “mainstream” methodology involved capturing performance specifications, mapping them onto a graphical representation, and performing a static analysis to derive the expected workload for the system. This workload was in turn used when analysing a performance model of the computer hardware.

Techniques were presented for improving — in addition to controlling and evaluating — the performance of software systems during design. Smith [171, 172] presented a sets of design principles for high-performance software. From a practical perspective, Smith emphasises that the software performance engineer should be a member of the development team from the start of the project, and that project management should exhibit early concern with software performance to demonstrate its importance for the development team.

### 5.3 Software Execution Graphs

Software execution graphs were introduced by Smith [167]. The associated “basic methodology” involves capturing performance specifications, mapping them onto a graphical representation, and performing a static analysis to derive expected, worst, and best possible response times for the system.

**Graph representation** A software design is described as a graph with functional components as nodes. A functional component is a collection of program statements, subroutines, modules, or programs that perform a single logical function together. Edges of the graphs represent paths between components. One edge of the graph corresponds to some kind of a context switch, like a procedure call. Edges may have probabilities associated with them.
There are four node types and five edge types in Smith’s software execution diagrams, as shown in figs. 5.2 and 5.3 adapted from [167]. The node types are: basic nodes (fig. 5.2a) representing a functional component whose execution characteristics are defined at the detail level of the graph, collapsed nodes (fig. 5.2b) representing a function whose execution characteristics are described in a sub-graph at the next level of detail, repetition nodes (fig. 5.2c) defining the beginning of a loop that will be iterated \( N \) times, and dummy nodes (fig. 5.2d) with no associated processing, occasionally used to represent special software models.

The edge types of software execution graphs are: arcs (fig. 5.3a) showing flow of control or a switch of protection domain, bi-directional arcs (fig. 5.3b) used for function “calls,” showing that control will return to the original node when processing is complete at the destination node, double bi-directional arcs (DBA’s) (fig. 5.3c) used as for bi-directional arcs with the exception that control returns to the driver \( X \) between the calls, dummy arcs (fig. 5.3d) used to illustrate precedence between nodes, e.g. between a repetition node and the last node of a loop, and self loops (fig. 5.3e) which are special dummy arcs used in connection with DBA’s.
As already mentioned, this graphical representation is extended by adding parameters representing execution characteristics to the graph nodes. These parameters are provided both for nodes and links. Node parameters describe the expected resource usages of the software component, such as CPU and disc time, etc. The parameters associated with links, give estimates of how the software nodes are typically sequenced.

**Graph specification** A top-down approach is used in the specification of these graphs. Initially, a graph is specified to describe the *most common* execution paths of the software. The assumption is that these most common paths will have the strongest impact on performance. Underlying this simplification is the 80%–20% principle of software performance engineering [171]. This principle states that only 20% of the software will determine 80% of its performance. The idea is to associate the nodes and edges of the software execution graph with estimates of the parameters that are relevant to its performance. For some nodes, providing parameter estimates may be difficult. Such nodes may be *decomposed*. When decomposed, a node is expanded into a subgraph. Expansion of “difficult” nodes continues until all nodes of the graph have been parameterised. The result is a hierarchic graph specification of the projected software system. Subgraphs may be collapsed into higher-level graphs through analysis.

Software execution graphs provide a graph language in terms of which projected software systems can be represented. The resulting graph is then annotated with attributes representing the expected computer resource requirements for the software components of the graph. This graph can then be analysed to estimate the work that will be produced by the software in development when it is finished. Additional queuing theory extensions of this basic methodology is provided in order to handle environmental factors and concurrency problems.

**Basic analysis** An execution graph is ready for analysis when it has been fully parameterised. Smith [167] provides an algorithm for this analysis. The algorithm is bottom-up, first calculating the best, worst and expected delays associated with the processing for each leaf node. This calculation uses the parameter estimates provided for the node. As soon as every node in a graph has been analysed in this way, a path analysis is employed to find the best, worst and expected processing time for the whole graph. The path analysis will use additional parameters associated with the links of the execution graph, such as maximum, minimum, and expected number of iterations inside a loop. The algorithm halts when the top-level graph has been analysed in this manner. The processing times of the top-level graph will correspond to the best, worst, and expected processing times of the software in question.

The initial performance specifications also include the *performance goals* of the system. These goals will be compared to the analysis results to judge whether the performance of the projected software system is likely to become acceptable.

The graphical representation extended with performance specification parameters constitute the “basic methodology” of extended design specifications, together with the performance analysis and evaluation techniques presented.

However, performance estimates obtained using the basic methodology are only correct for systems without environment-dependent factors between system components. In addition,
there should be no concurrency problems involved. Smith provides extensions to the basic methodology to counter these two cases, which will be treated in separate.

**Handling environmental factors** Smith treats three environmental complexities: data dependency, resource competition, and memory contention.

**Data dependency** Smith notes [167] that it is often impossible to provide precise specifications of execution characteristics when the behaviour of the software depends on the data it processes. Three cases of data-dependent systems are considered. These cases cover varying looping numbers, varying resource demands, and varying branching probabilities, depending on input data. The data-dependency problem is resolved by introducing parameters representing the data that the execution depends on. This means that a data dependent loop, resource usage, or branching is represented by a variable rather than a fixed value parameter. The algorithm is enhanced accordingly. The output of a design study of a system containing data dependencies will be a response time function of possible inputs, rather than a value.

**Resource competition** The software system being described is unlikely to run alone on its computer. Other software that executes on the same computer will influence the performance of the projected system. The resulting resource competition will introduce queueing delays into the response times of the software being described. To meet this problem, Smith introduces queueing network models into her method. A simple, standard queueing network is used. Algorithms are provided to derive the parameters needed for those models from the execution graph.

In addition, parameters must be provided for the competing software systems in the execution environment. This can be done by measurements. The execution environment specifications are collected only once for an environment. After that, they can be used at all stages of software design, and for several software designs within the same environment.

**Memory contention** A specific case of resource competition is memory contention. In case memory contention is anticipated, a more refined queueing network model is used. Again, the execution graph provides the parameter estimates for this model. In this case, more refined parameters for the competing software systems must be provided.

**Handling concurrency problems** Further extensions to the basic methodology are introduced to capture the effects of multiple, simultaneous users. Queueing network models are introduced to represent concurrency models in these cases too. Smith treats internal concurrency, synchronisation, mutual exclusion, and blocking. Queueing theory provides analytic solutions to the internal concurrency problem, while approximation methods are developed for the other three. In all cases, model parameters are derived from the execution specification graph.
5.4 Computation Structures

This section will outline the computation structures introduced by Booth et al. [162].

The basic idea of a computation structure is that [38] “a model is needed to describe the relationship between the deterministic process of the system and the probabilistic properties of the information being processed.” Thus Booth’s model views a processor as a deterministic transducer accepting an input sequence \( x \) and generating an output sequence \( y = f(x) \). The processing of input \( x \) has cost \( c(x) \). The input \( x \) is produced by a probabilistic information source from a set \( L \) of possible inputs, \( x \in L \), with probability \( p(x) \), so that \( \sum_{x \in L} p(x) = 1 \). Knowing the characteristics of the input sequence, the processing cost can be calculated from a computation structure. This situation is depicted in fig. 5.4. In [34] this model is extended with a feedback loop from the processor output to the information source. This however, makes model analysis more complicated, and will not be considered here.

To derive the execution time and spatial demands from a computation given its input characteristics, Booth expresses the software system to be analyzed in terms of two graphs: a data flow graph and an operation precedence graph. These two graphs together constitute a computation structure.

A data flow graph consists of a set of nodes \( N_d = \{ S, P \} \) where \( S \) is a set of storage cells depicted by rectangles and \( P \) is a set of operators and decision elements. Operators are depicted by circles with an operation name inside them. Empty circles represent decision elements. The diamonds which have predicates written inside them are also part of these decision elements. This is shown in fig. 5.5. (Figs. 5.5 and 5.6 are adapted from [162].)

The edges of a data flow diagram denote where an operator takes its inputs from and places its results in. In addition, the edges show what data is used for a decision. These represent “storage cell accesses.”

A precedence graph consists of a set of nodes \( N_p \). Nodes may be either operators as for the data flow graph, \( \text{AND-nodes} \), \( \text{OR-nodes} \), or \( \text{decision-nodes} \). AND-nodes and OR-nodes are depicted by circles with \( \lor \) or \( \land \) symbols written inside them, respectively. Decision-nodes are depicted by diamonds. In addition there are start and stop symbols, depicted by round-cornered ovals as is shown in fig. 5.6.

The edges of a precedence graph denote “flow of control” inside the graph. Operators have

![Diagram](image-url)
single incoming and outgoing arrows depicting control flow. Decision symbols have a single incoming and several outgoing edges. Each outarrow from a decision node points to the operation that is chosen as result of the decision. The precedence of operators are indicated by the directed edges of the graph. Precedences may be direct or indirect, accordingly.

Parallelism in a computation structure graph is indicated when two or more operation nodes appear in the precedence graph with no precedence between them. In these cases, the operations are said to occur in parallel. AND operators perform merging of two or more parallel operator sequences. Thus AND operators effectuate the synchronisation of parallel tasks. OR operators perform merging of two or more alternate operator sequences arising from a decision.

As already mentioned, there are three sets of quantitative attributes of interest in a computation structure. These may be chosen individually or in combination: \( t_i \), execution time is the time required to perform the \( i \)th operation, \( s_i \), spatial cost is the memory space needed to store the instructions associated with an operation, and \( s_{i,X} \), external spatial cost is the data space needed as input and output for an operation. This data space is denoted by \( X_i \). Thus the cost of an operator \( r_i \) may be denoted as a triple \((t_i, s_i, X_i)\).

A computation structure provides a model of computation that allows these attributes to be associated with the primitive operations of the computation. The goal of Booth's computation analysis is to derive the overall cost of the computation from the execution times and spatial demands of its operators. A number of associated solution techniques and methods for deriving and validating parameter estimates exist. A simple probabilistic estimation technique found in [162] will be outlined as an example.

This method uses the technique of stepwise “cost combinings” of serial and parallel pairs of operators and of decision elements. The three basic types of cost combinings are shown in fig. 5.7. Analysis proceeds by reducing sets of serial or parallel operators to single operators,
and shifting decision elements forwards in the graph. The analysis halts when the computation structure has been reduced to a single decision element at the start of the graph. For these final nodes, the expended time and spatial cost will represent the computation cost, in case of the input parameters corresponding to that particular branching.

5.5 Database Transaction Design

Oftedahl & Solvberg [135] presents a mathematical model for logical design of high-performance databases. In addition to integrating different users’ need for data it is argued that the database design should also take into account how the applications use of these data. Hence
performance trade-offs must be made between these applications. The framework provides a sensitivity analysis technique to pin-point important design parameters which can then be reestimated with more care.

The design procedure involves the following steps [135]:

- specification of user transactions and information repositories;
- quantification of system parameters;
- calculation of information repository usage;
- identification of crucial design parameters;
- reestimation of crucial design parameters and recalculation of usage of the information repositories, and
- restructuring the information repositories to obtain a database with satisfactory performance.
Transactions and information repositories Each software application is represented in the form of a database “transaction.” The framework assumes that a flow-chart is developed for each major database transaction as part of the requirements specification phase. These flow-charts consist of decision and processing boxes connected by directed arcs. In addition, each box is linked to a description of the information repositories it uses or modifies the data of. For the purposes of quantitative database optimisation, decision boxes are extended with branching probabilities which must be provided as part of analysis.

Quantification of system parameters A global information structure is characterised by a triple \((A, W, n)\), where

- \(A = \{a_u\}\) is a set of data elements;
- \(W = \{w_u\}\) is a set of weights associated with data elements, e.g. data transfer costs or sizes — depending on the problem at hand, and
- \(n\) is the number of data elements in the structure.

A (database) transaction is represented as a 6-tuple \((\alpha, V, E, P, f, m)\) derived from the extended flow-chart:

- \(\alpha = \{\alpha_i\}\) is a set of processes corresponding to the processing boxes;
- \(V = \{v_i\}\) is a vector of properties for processes \(\alpha_i\), e.g. the cost of activating the process — depending on the optimisation problem at hand;
- \(E = \{e_i\}\) is a vector of entrance probabilities to process \(\alpha + i\) derived from the external input arcs to the flow-chart;
- \(p = \{p_{ij}\}\) is a matrix of transition probabilities between processes derived from the switching probabilities;
- \(f\) is the number of invocations of the transaction over some period of time, and
- \(m\) is the number of processes.

Lowe [127] defines:

\[
\Gamma^T = E^T(I - P)^{-1},
\]

(5.1)

where \(\Gamma = \{\gamma_i\}\) represents the number of times each process will become active during an execution of the transaction. The traffic within a transaction can then be expressed as

\[
\tau_{ij} = \gamma_i p_{ij},
\]

(5.2)

where \(\tau = \{\tau_{uij}\}\) are the expected numbers of control transfers from processes \(i\) to \(j\). There are two types of processing costs involved:

- the activation costs of processes, \(C = \Gamma^T V\), and
5.5. DATABASE TRANSACTION DESIGN

- the costs of control transfer between processes, expressed by $\tau$.

The coupling between a transaction and the data elements $A = \{a_u\}$ is expressed by a matrix $Q = \{q_{i,u}\}$, where $q_{i,u} = 1$ iff process $a_i$ accesses data element $a_u$, and $q_{i,u} = 0$ iff not. The traffic $T = t_{u,v}$ within the global information structure can now be derived as

$$T = Q^T \tau Q, \quad (5.3)$$

and the activity vector $G = \{g_u\}$ becomes

$$G = Q^T \Gamma, \quad (5.4)$$

where $g_u$ represents the number of times each data element will become active during an execution of the transaction. The above measures can be accumulated over several transactions using the invocation numbers $f$.

Information structure usage The usage of the data elements can be determined from the $T$ matrix. This matrix indicates how access paths should be organised in the logical database. Of course, these considerations should be applied to the $T$ matrix after it has been accumulated over transactions:

- $t_{w,v} = 0, t_{v,u} = 0$;
  No access paths between elements $a_u$ and $a_v$ are needed.

- $t_{w,v} > 0, t_{v,u} = 0$;
  A one way access path is needed from element $a_u$ to $a_v$.

- $t_{w,v} > 0, t_{v,u} > 0$;
  Two way access path is needed between elements $a_u$ and $a_v$.

In the two latter cases, the numerical sizes of the non-zero parameters indicate the relative importance of the access paths in question. An access path can later be realised as a set relation in a hierarchical database, by indexing or hashing, or by foreign keys in a relational database. Also, $a_u$ and $a_v$ can be grouped together physically if feasible.

Identification of crucial design parameters The framework also provides a sensitivity analysis technique to pin-point crucial design parameters. Sensitivities are obtained with respect to:

- transition probabilities $P$;
- the property vector $V$;
- transaction frequencies $f$, and
- data element weights $W$.

In either case, sensitivities are obtained by differentiation of the equations given. After having undertaken a sensitivity analysis, the most important design parameters can be reestimated and recalculated, and the data elements can be restructured accordingly.
5.6 Software Performance Engineering Tools

A number of tools for software performance engineering have recently appeared. In addition, the sp and HIT tools of the previous chapter 4 can also be regarded as software performance engineering tools to some extent.

The CRYSTAL tool mentioned in sec. 4.7 was developed by Buzen [47]. It supports performance specification of projected software systems in terms of estimated resource demands and expected execution paths. Modular software specification is also supported.

The projected software system is specified in CRYSTAL in terms of three components:

- a set of module specifications describes the hardware resource requirements of each software module in machine-independent form;
- a workload specification identifies workload components, and for each component specifies its workload intensity, and
- a configuration specification described the characteristics of hardware resources and files used.

From these components, CRYSTAL generates and analyses a queueing network model of the projected system.

The CRYSTAL tool [26] is interfaced with other BGS tools for performance evaluation as described in sec. 4.7.

The Teamwork environment is an integrated CASE tool with software performance engineering facilities [51]. The Teamwork/IM, /SA, /RT, and /OOA tools support information modelling, structured analysis, real-time system specification, and object-oriented analysis respectively. The resulting /SA, /RT, and /OOA specifications can be simulated in the Teamwork/SIM tool for software performance engineering. Teamwork/SIM provides an architectural modelling language to facilitate specification of computing hardware.

Cadre Technologies also provides an Architecture Design & Assessment System (ADAS) for system level simulation of software/hardware co-design [52], which is also interfaced with Teamwork.

The PrM tool of part IV links the PPP (sec. 3.3) and IMSE (sec. 4.8) toolsets. Hence the resulting combined environment can also be regarded as an integrated software performance engineering tool.
Chapter 6

Requirements for Performance Engineering of Information Systems

This chapter will present the major requirements for successful performance engineering of information systems. Most of them will be satisfied by the framework presented in the following part III. Sec. 14.1 of the conclusion will summarise/discuss the achievements of the thesis with respect to the requirements presented here.

These requirements have continuously changed during the thesis work as understanding of the problem increased. However, it was the first drafts of these requirements that shaped the thesis into its initial form. Early discussions were reported in 1989/90 [136, 137, 141].

6.1 Problems with Software Performance Engineering

The discussion of chapter 2 made the need for performance engineering during information system development clear. However, contemporary approaches to performance engineering of information systems [173, 5, 156] cannot yet be widely applied because of unsolved problems, such as

- low (or not demonstratable) cost-efficiency;
- lack of integration with information system engineering;
- lack of integration with capacity planning, and
- lack of parameter capture support.

where the second point is the main motivation behind the work of this thesis. They will be considered in separate in this section, while the next four sections will discuss each one of them in detail.

This set of requirements was first formulated in [137] and has served as the starting point for this thesis, although it has been continuously revised as user needs and technological opportunities became increasingly more well-understood.
Cost-efficiency Of course, the expenses saved through software performance engineering must be greater than the expenses involved. In practice, an accurate assessment is almost impossible to make. First, reduced development cost due to a successful software performance study will seldom be noticed by managers. Second, hardware cost reductions because of well-designed software and the possibility of reusing performance and workload models give software performance studies value beyond that of a particular development project, as was explained in sec. 2.3. The practical requirement therefore becomes that software performance engineering should be as inexpensive as possible. This is a dominant theme in the remainder of the thesis, and is the main point behind the considerations that follow in this chapter.

Integration with information system engineering Existing software performance engineering tools are stand-alone. They are not thoroughly integrated with the toolsets already used for information systems engineering. Since performance evaluation is only a small part of the information system development picture, engineers do not want to “respecify” their design before initiating a software performance study.

It is therefore clear that a successful approach to performance engineering of information systems must be closely integrated with existing information systems engineering methods. This point is of course strongly related to the above issue of cost-efficiency.

Integration with capacity management Existing, specialised performance evaluation approaches for information systems engineering are not practical (or realistic) because of their tendency to consider single applications and single computers in isolation. As a result, they

- ignore the load balancing possibilities that might otherwise improve cost-efficiency;
- do not take into account how the performances of projected and existing software systems interact, and
- fail to take advantage of previous performance modelling efforts made for capacity management purposes (and vice versa).

Parameter capture support Especially in the early design stages, getting good workload estimates is difficult. For projected applications, this is no surprise, as the software system to describe does not yet exist. However, the problem is also there for existing applications, because of

1. the large number of parameters needed: the possibility of load balancing makes it necessary to consider the computers and applications of a large part of the organisation at the same time;
2. measurement problems: the computerised information system may not even be accessible, and
3. inadequate tools: [119] comments that performance monitors that reliably break down resource usage by both workload component and resource are not commonly used due
to prohibitively high overhead. When modelling the performance of computers in terms of the applications that run on them, this dual break down is exactly what is needed.

Of course, making parameter capture easier would greatly reduce the additional cost of software performance engineering.

6.2 Cost-Efficiency

**Require little additional development effort** Software performance is only one of many aspects that should be considered during information system development. In practice, only aspects that are easy to consider, are considered.

One major drawback of contemporary approaches to software performance engineering is the reliance on dedicated software performance modelling languages. This is unacceptable for application developers, because:

1. the projected application may already be specified in another language;
2. application developers cannot give priority to respecifying the application design in several different languages, and
3. application developers may not want to learn several specification languages.

In summary, the language gap between contemporary information system engineering and software performance makes the additional development effort required too high.

**(Re-)Use existing design specifications** A consequence of the last point is that the (functional) design specification established by application developers should be directly usable for information system performance engineering.

**(Re-)Use existing performance and workload models** As a consequence, the performance and workload models of the existing computerised information system does not have to be created more than once. When a new application is to be introduced, these models can be reused. In case of hardware expansions or upgrades, the old model will have to be modified accordingly. In many cases, this will have been done anyway, as the system managers will have kept the performance model up-to-date for capacity planning purposes meanwhile.

Hence performance engineering of information systems within an organisation only requires much effort the first time it is done. After that, old models can be updated and reused. The new workload models needed are derived from annotated design specifications that have to be created anyway. The integrated support environment should therefore support model reuse.

**Require little additional interpretation effort** Just as the design specification established by application developers should be usable for performance engineering, the resulting performance predictions should relate to that design specification to reduce interpretation effort.
6.3 Integration with Information System Engineering

The previous point about (re-)use of software performance models should be seen in a wide perspective. Successful and cost-efficient application of performance evaluation in the design process requires tight coupling with the information system engineering process, both at the modelling, tool, and methodology levels. The following points also state that performance evaluation of information systems should be directly useful at the organisation level, and not only inside the information system development team.

![Diagram](image)

(a)

![Diagram](image)

(b)

Figure 6.1: “Ideal views” of the integrated environment for information system engineering (a) and capacity managers (c).

**Usefulness for novices in performance evaluation** As mentioned in sec. 2.4, performance expertise is a scarce but valuable resource. In addition, the combination of software engineering and performance evaluation skills is uncommon. Therefore, application developers are unlikely to accept a tool which requires deep understanding of performance evaluation.

Therefore, an integrated environment should hide as much of the performance evaluation apparatus as possible from information system developers. The ideal situation is depicted in fig. 6.1a. Application designers access the integrated environment through its information system engineering toolset, with the computer performance evaluation toolset acting as a “grey box” in the background. The complementary “ideal view” of the capacity managers are shown in fig. 6.1b.
6.4. INTEGRATION WITH CAPACITY MANAGEMENT

Meaningful information system level results  As mentioned at the beginning of chapter 2, computer performance is becoming more and more important at the management level of enterprises, as is computer expenditure. Computing devices are no longer looked upon as an unlimited resource inside the organisation. Instead, they have become investment objects in the same manner as staff, production equipment, etc., and they are amenable to the same cost-benefit analyses. Because of this it is important that performance predictions made by the combined toolset can be interpreted by managers.

Responsivenesses are the performance measures that affect users directly, and they are the indices determining efficiency improvements in the organisation. Hence the integrated environment should provide response times for the computerised information system and residence times for any part of it. This creates problems due to the “response time unpredictability” explained in sec. 4.4.2. Therefore, error bounds and/or confidence intervals should be provided for the responsiveness and residence time measures.

Usefulness throughout the information system life-cycle  As mentioned in sec. 2.4, severe performance problems are more likely to be caused by bad designs than by inefficient implementations. Hence an integrated support environment must support performance prediction from the design specification in the early development stages as well as in the late ones. Ideally, a range of evaluation techniques should be provided for different phases. Typically, fast approximate and simple exact analytic models suffice for early, abstract design specifications, while detailed simulation may be needed for detailed specification in the late stages.

Provide rapid feedback to designers  Design feedback [180] is an important aspect of information system design. Performance predictions can be regarded as one such type of feedback. Performance feedback for the projected application should be immediately available to guide important design decisions. This means the the performance analysis cycle time must be short, preferably interactive.

Answer what-if queries  Two important types of design decisions are modifications and selection among alternatives. In either case, design decisions may be guided by performance considerations. Again, this requires short analysis cycle times, preferably interactive.

Support performance optimisation  In case application performance turns out to be unacceptable, the designer needs to know how the performance of the design specification can be optimised.

6.4 Integration with Capacity Management

The need for (re-)use of capacity management models for software performance engineering accordingly leads to a more general requirements for interaction with the capacity management process at the modelling, tool, and methodology levels. The following requirements
point out that successful software performance engineering can only be performed in the context of the existing computerised information system. Therefore, it must be carried out in cooperation with capacity managers.

**Estimate impact of new large applications on system performance** Sec. 2.4 mentioned extrapolation from past trends as a commonly applied approach to capacity planning. However, it was pointed out that introduction of large projected applications could not be handled in that way. Therefore, performance measures produced as in a software performance study, may be very useful for capacity planners also.

The future workload of the existing applications is extrapolated from measurements. The future workload of the projected application is derived from its annotated design specification. When superimposing these two workloads on the performance model, system performance under the new conditions can be studied.

The integrated environment should therefore provide support for trend extrapolation, as described in sec. 2.4.

**Estimate impact on performance of existing applications** A third set of performance measures produced in a software performance study are the responsivenesses of existing applications. This is an important issue which is often overlooked. It is not acceptable that the projected application exhibits excellent performance if at the same it degrades the performance of other applications too much.

**Take existing applications into account** As pointed out in sec. 2, the projected application is likely to compete with other, already existing applications when in production. Therefore, performance predictions made during development must take the workload of these existing applications into account. (Early approaches to software performance engineering [196, 87, 162] failed to account for this. The idea of representing existing software systems as additional customer classes in a queueing network analysis was first introduced by Smith [167].)

Load balancing solutions can only be found if a complete model of the information system infrastructure of the organisation is maintained. At least, updated models of all the available computers must exist, as well as models of the workload put on these computers by the application. Note that other performance evaluation tasks within the enterprise, such as capacity planning, update studies etc., would also benefit from this.

**Maintaining a model of the computer system infrastructure** All the above requirements are implicitly met if the integrated environment supports

- *an updated workload model of the projected application* created and maintained by the information system developers;

- *a set of updated workload models of the existing applications* created and maintained by the capacity managers, and
• an updated performance models of the existing computer hardware also created and
maintained by the capacity managers.

To simplify the task of capacity managers it is therefore preferable to make as many of the
information system engineering aspects hidden as possible. This creates a “capacity planners
view” of the integrated environment which is complementary to the “application developers
view” of sec. 6.3, as is depicted in fig. 6.1b. Since information system engineering is a less
specialised — and therefore more generally understood — task than computer performance
engineering, this issue is less important than its counter-example of the previous sec. 6.3.

When creating and maintaining performance and workload models, it is important to remem-
ber that many organisations have more than one computer at their hands. Proper performance
evaluation of information systems cannot consider only one system in isolation. Therefore,
performance models of all the computers of the organisation must be created. Of course, an
updated set of such performance model models and their interconnections is also very useful
for capacity planning and management.

Support for balancing studies With such a library of performance models and their
interconnections kept updated in the integrated environment, balancing studies are supported:

• there will be one — perhaps several — workload models of projected applications;
• there will be several workload models of existing applications, and
• there will be several performance models of several computers.

Experiments may now be carried out to search for an optimal distribution of applications
among the computers. A library of such plans should be part of the integrated environment.
In this way, the combined toolset will provide tool support for “balancing” as addressed in
sec. 2.4. (Full support for balancing study also requires compatibility between applications
and computers into account. This is not a performance issue and is therefore outside the
scope of this thesis.)

6.5 Parameter Capture Support

Performance parameter capture is one of the most difficult parts of software performance
engineering. It is therefore a main factor determining cost-efficiency. Numerous possibilities
exist for simplifying parameter capture. The integrated environment should support as many
of those as possible.

Both coarse and detailed models Because of 1) parameter capture difficulties and 2)
the need for performance evaluation throughout the system development, very coarse models
should be an acceptable alternative to more detailed ones in the early design stages. This
applies both to design specifications and to performance and workload models.
Such models can be analysed using the same techniques as other more detailed ones, but are likely to produce less accurate results. As development proceeds, the design specification will gradually become more refined, and the workload and performance models created from it correspondingly more detailed.

**Workload modelling alternatives** Since workload modelling is the most time consuming part of parameter capture, it must be well supported by the integrated environment. Smaller, less important applications should only be described in simple ways, while larger applications that are crucial to performance should be specified in detail. An application may be represented as

1. a workload model established through measurements;
2. a composite workload model created by hand to describe an existing application, or
3. a performance or workload model derived from a design specification.

The integrated environment should support all these alternatives.

**Performance parameter validation** When employing approximate models in the early stages of design, it is crucial that a means of *validating* these estimates later is provided. As pointed out in sec. 6.1, parameter validation must be supported both during design and implementation.

- During design, parts of the design specification can be solved to predict the workload put on the computer by the application or by parts of it, and compared with earlier estimates.

- During testing and production, the (real-world) resource demands of the application — or of parts of it — can be measured. These measurements can be validated against the predictions made. Such comparisons are particularly useful for the first part of the application to be implemented. It is likely that errors in parameter estimates here will indicate errors in other parts of the design specification. Thus parameters can be corrected for the rest of the design specification.

**Identify performance-critical parts of the design** To focus parameter capture effort on key design parameters, the integrated environment should support identification of critical parts of the design specification with respect to performance. The 80%-20% rule of Smith [171] described in sec. 5.3 can be applied here. In fact, the thesis will apply a slightly more general “(1–x)%–x%” version of this principle, where x typically in the range 10, . . . , 20. Parameters within the x% most performance-critical part of the design specification are the ones that must be estimated with most care. It will turn out in sec. 10.1 that this is also useful design feedback to the application developers, since it points out which parts of the projected application that can most effectively be optimised.
6.5. PARAMETER CAPTURE SUPPORT

**Bounds analysis** Bounds analysis of responsiveness is another useful technique in conjunction with coarse, early models. If the performance models used are stochastic, this analysis will give percent-quantiles rather than absolute bounds.

The integrated environment should therefore support bounds analysis.

**Parametric analysis** Parametric analysis of performance and workload models is useful when all but a few performance parameters are available. Parametric analysis allows fixing such “difficult” parameters as late in the software performance modelling process as possible through a parametric pre-analysis. After pre-analysis, multiple values for the difficult parameters can be analysed quickly.

**Sensitivity analysis** Sensitivity analysis is a somewhat more accurate — and hence more sophisticated — alternative to using $(1 - x)\% - x\%$ rules. Rather than applying heuristics, sensitivity analysis provides quantitative parameter sensitivities of how much the parameter affects some performance measure of interest.

Sec. 4.6.2 has surveyed the available sensitivity analysis techniques for different types of performance models, concluding that analytic sensitivities are preferable for analytic models (i.e. whenever available).
Part III

The Framework
This part will present the framework for performance engineering of information systems. This framework constitutes the major result of the thesis. The basic framework is first introduced in its basic form in chapters 7 and 8. It is then extended with techniques for parameter capture support, sensitivity analysis, and analysis of distributed systems in chapters 9, 10, and 11.

- Chapter 7 “Baseline” presents the views of information systems, design specifications, performance evaluation, and information system performance engineering upon which the framework is based;
- chapter 8 “Performance Prediction” demonstrates how these views support prediction of the performance of projected information systems;
- chapter 9 “Parameter Capture Support” extends the basic framework with support for capturing necessary performance parameters during information system development and for information system design optimisation;
- chapter 10 “Sensitivity Analysis” presents a technique for determining the sensitivity of performance measures with respect to performance parameters with the same aims as chapter 9, and
- chapter 11 “Distributed Information Systems” extends the basic framework with capability of analysing distributed systems.
Chapter 7

Baseline

This chapter will present the basic views upon which the framework for performance engineering of information systems has been built. The most important concepts to be used in chapters 8–10 will be introduced here. Also, these are the concepts which chapter 11 on distributed systems will build upon. Concepts will be introduced in terms of bold-faced “definitions.” However, the aim of these definitions is to point out important concepts and relate them to one another rather than to blur the initial picture with immense detail. Therefore, some of the concepts will not obtain a more precise meaning until being used in the following chapter 8.

First, sec. 7.1 will present simple view of information systems based on the three levels of fig. 1.1 in chapter 1. Secs. 7.2 and 7.3 then introduce a functional and a performance-oriented abstraction on this view, respectively. Finally, sec. 7.4 integrates these two abstractions and becomes the baseline of the framework.

The baseline aims at non-distributed systems only. The extensions needed for performance prediction of distributed information systems will be introduced in chapter 11.

7.1 View of Information Systems

In the terminology of this thesis, an information system as defined in chapter 3 is used (or will be used) by an organisation (fig. 7.1a). Hence the term information system of this thesis corresponds to what is sometimes called a “computerised information system.” The organisation concept of this thesis is close to what is sometimes called a “manual information system.” The information systems considered here are therefore more narrowly defined than “information systems” comprising both a “manual” and a “computerised” subsystem.

An information system in this more narrow sense consists of a set of applications running on a computer (fig. 7.1b). An application may be either projected in which case it is under development or existing in which case it is in production. This thesis will focus on projected applications. Computerised information systems and applications both provide functions to their organisations, while computers provide resources to its set of applications (figs. 7.1a-c). The set of applications that run on a computer and the organisation which uses the
Figure 7.1: Three alternative views of information systems. The design specification and performance evaluation views of secs. 7.2 and 7.3 will be abstractions of this.

information system (or set of applications), devolve the **workload** on that computer. Under some workload, the computer exhibits a certain **performance** (fig. 7.1c).

![Diagram](image)

Figure 7.2: Functional abstraction (a) and performance abstractions (b) of the information system view.

In the following secs. 7.2 and 7.3 **abstractions** of the information system will be introduced. In particular, a **functional abstraction** is an abstraction focusing on what happens in the real-world and how it happens (fig. 7.2a), while a **performance abstraction** is an abstraction focusing on the time things take in the real-world and why they take time (fig. 7.2b). The former is related to the **design specifications** representing the functionality of projected applications during development, while the latter is related to computer performance evaluation.

In what follows, a **language** is a formalism for expressing abstractions. Most languages written about in this thesis will have a diagrammatic representation. A **model** is a statement of someone’s perception of something existing or imaginary, expressed in a language. A model is therefore a formalised representation of an abstraction. A **tool** is (narrowly) a computerized system supporting the creation of models using a language. A **method** is a procedure for using a tool and a language.

A **modelling approach** is a language with its corresponding tools and methods. The set of
tools and paradigms used within an engineering domain is called an environment.

In the remainder of this thesis, information system entities will mostly be depicted as parallelograms (figs. 7.1–7.2), while many abstractions will be depicted as rectangles (e.g., figs. 7.3 and 7.4). The small circles on top of the parallelograms and rectangles (e.g., fig. 7.9) represent either functions, operations, resources, or service centres, according to what kind of object they are attached to. Dotted arrows will often be used to make the figures easier to comprehend, without themselves being part of the figures.

7.2 View of Design Specifications

![Diagram showing views of design specifications](image)

Figure 7.3: A design specification abstracts an information system application (a). Both design specifications (b) and components (c) provide operations.

A design specification $S$ is a functional abstraction of a projected application. No assumption is made about the manner in which this design specification is established, only that it exists in a consistent state every time a performance analysis is to be done. A design specification provides operations, just like the projected application will provide functions to its organisation (fig. 7.3a). The set of operations provided by a design specification must correspond to the set of functions provided by the application it abstracts. Since the projected application is likely to be complex, this design specification is hierarchical, containing a DAG (directed acyclic graph) of components (fig. 7.4a).

An operation $o$ is a functional abstraction of a function provided by an application or by some part of an application. Sets of operations are denoted $O$, and $O_S$ is the set of operations provided by design specification $S$.

A component $\gamma$ is a static representation of part of an application. A component provides a set of operations $O_\gamma$ (fig. 7.3b), just like a design specification. Sets of components are denoted $\Gamma$. $\Gamma_S$ is the set of components in design specification $S$.

A component may be composite since it can contain a set of subcomponents $\Gamma_\gamma$ (fig. 7.4b), whose operations it will use to implement its own operations. Of course, subcomponents may again have subcomponents on their own. A component is a partial view of the application because it represents the application at a very abstract level, or because it represents only part of the application at a detailed level, or because of both in combination.

---

1The greek letter $\gamma$ (gamma) has been chosen to denote components because the roman letter $c$ (and $C$) will later be used to denote software component complexity. However, both $\gamma$ and $c$ are the third letters of their respective alphabets.
Figure 7.4: A design specification has a component hierarchy (a) which is a DAG. A component contains a set of subcomponents (b). A non-primitive subcomponent is at the same time a component at the next lower level (c).

Example
In a DFD-based modelling approach, the components would be dataflow diagrams, and operations would correspond to external flows (or sets of external flows) initiating “execution” of the diagrams in some sense. (The particular realisation of the framework reported in [142, 140] is based on a formalisation of DFD [88]). In an object-oriented methodology, components would of course be objects, with operations representing message types. □

The operations provided by a component are the \textit{external operations} of that component. The operations provided by the subcomponents contained in a component are its \textit{internal operations}. The set of external operations of component $\gamma$ has already been denoted $O_{\gamma}$, while its set of internal operations will be denoted $O_{\gamma}^\ast$.

The DAG in the design specification forms a \textit{component hierarchy}. Components are \textit{directly} or \textit{indirectly superior} or \textit{subordinate} to one another according to this hierarchy in the obvious way, e.g. the set of directly subordinate components of component $\gamma$ is $\Gamma_\gamma$. Since no cycles are allowed in the component hierarchy, there must exist some components which are not subordinate to any other components. These are the \textit{top-level components} of the design specification. The set of components in design specification $S$ has already been denoted $\Gamma_S$, while its set of top-level components will be denoted $\Gamma_S^\ast$. In what follows, top-level components will be identified with a “hat” $\hat{\gamma}$ to distinguish them from other components.

The set of operations provided by a design specification corresponds to the union of the sets of operations provided by its top-level components,

$$O_S = \bigcup_{\gamma \in \Gamma^\ast_S} O_\gamma,$$

(7.1)
7.3. VIEW OF PERFORMANCE MODELLING

There must also exist some components which are not superior to any other components (yet). These are the **primitive** components of the design specification. Other components are **non-primitive**.

**Example**

The design specification for the customer transaction processing system of fig. 3.5 is shown in fig. 7.5. The system provides four operations for initiating and terminating customer accounts, as well as deposit and withdrawal from accounts. The top-level component $\gamma_1$ of the specification has a dedicated subcomponent for each of these operations, and an additional subcomponent writing transaction statements back to customers. The Make withdrawal subcomponent in turn has three subcomponents for balance checking, balance updating in case the balance was sufficient, as well as transaction commitment. Hence fig. 7.5 abstracts the PrM process hierarchy structure and operations of the corresponding PrM model. Of course, fig. 7.5 does not depict the complete design specification as seen by the development team, but only a minimal subset necessary in this thesis.

Figs. 7.6 and 7.7 accordingly shows design specification for the Accounting system of fig. 3.6 **before** (fig. 7.6) and **after** (fig. 7.7) the Get balances submodel of fig. 3.10 has been established. Both points of development will be used as examples in chapter 11. □

![Diagram](image_url)

**Figure 7.5:** Design specification for the Customer transaction system of fig. 3.5. (The v’s of this and later figures are intended to resemble the component symbol $\gamma$. The number represent component indexes.)

### 7.3 View of Performance Modelling

The aim of performance modelling is to obtain a performance prediction through performance analysis of a performance model under a workload model, just like the computer exhibits a certain performance under some workload (fig. 7.8).

A **performance model** $M$ is a performance abstraction of a computer. A performance model provides services to its workload model, just like the computer provides resources to its set of applications (fig. 7.9a).
Figure 7.6: Design specification for the Accounting system of fig. 3.6 before decomposing the Get balances process.

Figure 7.7: Design specification for the Accounting system of fig. 3.6 after decomposing the Get balances process.

A service $\sigma$ is a performance abstraction of a resource provided by a computer.\(^2\)

A workload model $W$ is a performance abstraction of the set of applications that run on a computer and the organisation which uses them (fig. 7.9b). A workload model consists of an organisation workload and a set of workload modules (fig. 7.9c).

An organisation workload $\Omega$ is a performance abstraction of how an organisation uses the functions provided by an information system (or set of applications). An organisation workload consists of a set of workload intensities.

A workload intensity $\omega$ is a performance abstraction of how an organisation uses a specific function provided by an application. A workload intensity is either transaction, terminal, or batch [119]. The set of workload intensities contained in an organisation workload must

\(^2\)The greek letter $\sigma$ (sigma) is used to denote services, as the roman letter $S$ has already been reserved for design specifications. This is unfortunate, since the capital sigma $\Sigma$ which will be used to denote sets of services is also the sum operator $\sum$. However, the difference should be clear from context.
correspond to the set of functions used by the organisation it abstracts.

A transaction workload intensity is a performance abstraction of the case where a large number of users each use a function infrequently. This is a consequence of the “Law of rare events” [182]. A transaction intensity is characterised by an arrival rate $\lambda$. A terminal workload intensity is a performance abstraction of the case where a smaller number of users each use a function more frequently, and thinks for a while in between. A terminal intensity is characterised by a number of terminal users $n$ and a terminal think time $\theta$. A batch workload intensity is a performance abstraction of the case where a function is continuously used in the interval for which the computerised information system is considered. This workload intensity type is useful for representing long batch jobs, hence its name. It is characterised by a number of batch jobs $n$ only. In this thesis, numbers $n$ of terminal users and batch jobs will always be integer. This is a requirement of the queueing network analysis techniques which will be used.
Capturing workload intensity parameters has been treated in literature (e.g. Ferrari [78]). For practical purposes, the transaction workload is the most important workload type since it is the easiest to capture and sufficient for most cases [119]. For existing applications, performance measurements or accounting logs can sometimes be used. For projected applications, the workload intensities must be estimated from experience with similar systems and interviews of management and future users. An additional possibility is to automatically derive workload intensities from annotated “organisation specifications.” Such “organisation specifications” will typically have been created during the requirements analysis phase, prior to design.

![Diagram](image)

**Figure 7.10:** The workload module visit matrix maps workload module operations onto performance model service centres.

A **workload module** \( W \) is a performance abstraction of how the functions provided by an application uses the resources provided by some computer system (fig. 7.9d). A workload module provides operations to the organisation workload and uses **services** that will later be mapped onto **service centres** of the performance model. Hence a workload module specification is independent of performance model. This is important when comparing several alternative computers or computer upgrades which may support the computerised information system. The associated mapping \( \mu \) will be described in sec. 8.3.5 of the next chapter.

The set of operations provided by a workload module must correspond to the set of functions provided by the application it abstracts, just like the case for design specifications. The workload module of a design specification is represented as a **visit matrix** of average service centre uses (fig. 7.10). Hence a workload module visit matrix of this view corresponds to an **sp** module implementation as defined in the previous sec. 4.8.2.

This view of workload models is inspired by the **sp** approach [101, 129]. It deviates from the conventional view, where a “workload model” consists of a set of “customer classes.” However, the two views are compatible, because each conventional “customer class” consists of a “workload intensity” and a set of “service demands.” These “workload intensities” correspond to the workload intensities of the framework. Accordingly, the “service demands” contain the same information as workload module visit matrices.

Capturing “service demands” for existing applications has been treated in literature [78]. One approach is to derive “service demand” vectors by cluster analysis of performance measurements or accounting logs [55, 54]. Since the row vectors of visit matrices are “service demand” vectors, similar techniques are applicable within this framework. For projected applications, however, measurements or logs are not available. In this case, contemporary ap-
proaches are based on deriving “service demands” automatically from annotated “execution specifications” [173, 5, 47].

A performance prediction $P$ is a performance abstraction of the expected performance of a computer under a particular workload. A performance prediction contains a set of performance measures. A performance measure (or measure for short) is a performance abstraction of an aspect of the expected performance of some computer. A performance measure is either a responsiveness, a throughput, or a utilisation. A responsiveness performance measure $r$ is a performance abstraction of the time something takes inside the computerised information system. Examples of responsivenesses are response times, residence times, and sojourn times. A throughput performance measure $x$ is a performance abstraction of the rate at which something flows out of (and hence into — since flow balance is assumed) the computerised information system. A utilisation performance measure $u$ is a performance abstraction of the degree to which some computer resource is used. Responsivenesses and throughputs apply to workload modules of either projected or existing applications, while utilisations apply to performance model service centres. Other performance measures such as queue-lengths and distributions are less important in the framework.

A performance analysis is a mapping from a workload model to a performance prediction for some performance model. Numerous performance analysis techniques are available, ranging from algebraic product-form or separable queueing network analysis, through approximate analysis for “near product-form” queueing networks and numerical analysis of Markovian models, to special- or general-purpose simulation. This framework is currently based on simple MVA performance analysis of separable queueing networks, under the assumption that errors introduced elsewhere do not justify the additional effort of creating and analysing more accurate performance models. This consideration is in line with the general agreement that simple models are sufficient for capacity management purposes.

![Diagram](image)

Figure 7.11: Overall picture of the performance modelling view.

An overall picture of the performance modelling view of this chapter is given in fig. 7.11.

### 7.4 Towards an Integrated View

The design specification view of sec. 7.2 is now extended in order to integrate information system development with performance modelling concepts. The obvious place to start is the
design specifications of figs. 7.5–7.7 and the overall performance modelling view of fig. 7.11. The mapping of a design specification onto a workload module is shown in fig. 7.12. Additional workload modules are abstractions of existing applications. This mapping implies automatically deriving the visit matrix\(^3\) of the workload module from the design specification. If this is possible, the performance of a projected application can be predicted during its development, as well as its impact on the performance of existing applications and the computer.

Deriving the visit matrix of a workload module from a design specification requires extensions to the view of sec. 7.2 which will be introduced in the next chapter 8.

Having automatically derived a workload module for the projected application in this way, a performance analysis of the computerised information system can be made. The performance model used in this analysis must be created by conventional means. In a wider sense this means using queueing networks, Petri-net models, or special- or general-purpose simulation languages. Within this framework, separable queueing network models are assumed. In some cases, the performance model will already have been created for capacity management purposes, or as a result of previous information system performance engineering efforts. Unless the computer will be dedicated to the projected application, additional workload modules must be created for each existing application. All these additional workload modules, as well as the corresponding workload intensities, must be created by conventional means. Performance analyses should be carried out several times throughout application development. The workload module for the projected application must be derived anew every time. However, the workload intensities, workload modules for existing applications, as well as the performance model, can be reused. This considerably reduces the effort involved.

In this manner, performance predictions are obtained for the computerised information system after the projected application is in production. The responsiveness and throughput measures for the projected application indicate whether it will perform clearly acceptable or clearly unacceptable, or if further analyses based on refined design specifications are needed. These measures may also be used to compare the performance of alternative designs. The responsiveness and throughput measures for the existing applications predict to what extent

\(^3\)Visit matrices for workload modules were introduced on page 120 and depicted in fig. 7.10.
their performance will be degraded by the projected one. If the performance of some projected or existing application is unacceptable, precautions can be taken before the projected application is put into production. The utilisation measures of the performance model indicate where bottlenecks will be located. This information is useful if the computer must be upgraded, or in case some application must be optimised. In the latter case, the visit matrix for the application predicts which resources it will use most heavily.

This concludes the summary of the most obvious uses of the basic framework. Numerous other possibilities exist, and will be surveyed in subsequent chapters.
Chapter 8

Performance Prediction

To meet the requirements of chapter 6, this thesis presents concepts, analysis techniques and methods for performance engineering during information system development. This chapter introduces the basics which later chapters will extend. The resulting basic framework focuses particularly on integrating the states of the art in information system development and performance engineering and, with emphasis on simplicity and generality. This has been achieved by making the basic framework as independent as possible with respect to specification language. The basic framework can therefore be tailored to any specification language satisfying a small set of criteria.

The aim of the basic framework for performance engineering of information systems is to predict the performance of projected applications during development. To this means, its main constituents are

- a set of performance annotations which must be added to the design specification by the application developers;
- a workload derivation method which transforms the annotated design specification into a workload model of the projected application, and
- a workload and performance modelling framework which uses the workload model to predict the performance of the projected application during development.

8.1 Performance Parameters

The basic framework aims at predicting the performance of the projected application from its design specification. In accordance with the conventional capacity planning approach, visit counts will be derived to represent the projected software in a queueing network model which can then be analysed. This implies that the software performance model will essentially be static, with a dynamic model of the computer hardware, following Smith [173]. Chapter 11 will extend the approach with dynamic software models also.

To derive visit counts from the design specification, it must somehow be annotated with performance parameters. In defining such estimates, the main goal is of course to make
a selection that is both necessary and sufficient for making a performance prediction. In addition, the approach focuses on

- defining a minimum set of parameters needed;
- selecting parameters that are easy to capture and validate, and
- finding parameters that are natural and easy to express in terms of the design specification.

As a consequence of the latter two points, all annotations are average estimates rather than stochastic distributions. This is in line with the choice of operational queuing network analysis at the hardware level which will be introduced in sec. 8.3.7.

In summary, annotations are needed to answer the following three questions:

- **What happens when an operation on some component $\gamma$ is initiated?**
  An entry description specifies which operations on its subcomponents are initiated as a consequence, and how many times they are initiated on average. In a dataflow realisation, entry descriptions would be entered as average counts on the external input flows of the diagrams. An object-oriented realisation would be similar, depending on the graphical formalism used to represent object decompositions. Inside the framework, entry descriptions are represented as entry matrices $E$, whose elements $e_{o,o'}$ map component onto subcomponent operations. Of course, most elements of these matrices will be zero by default.

- **What goes on inside component $\gamma$ after an operation has been initiated?**
  An activity description specifies which other operations on subcomponents that are activated next when a subcomponent operation has terminated, and how many times they are activated on average. In a dataflow realisation, activity descriptions could be entered as average counts on the flows between the processes of the diagram. Again, activity descriptions are represented as activity matrices $A$, inside the framework, and most of their elements $a_{o,o'}$ are zero by default.

- **How much hardware resources are used by the primitive components $\gamma$ of the design specification?**
  For this purpose, the target platform of the development project must be known. The target mapping concept introduced in sec. 8.3.5 and refined in chapter 11 will strongly relax this restriction.

This chapter defines a target platform (or just platform for short) $P$ as the set of services $\sigma \in \Sigma_P$ representing the resources that constitute the underlying computer. (This is of course in line with the view presented in sec. 7.2.) Since the platform services of this and the next two chapters 9 and 10 will always represent hardware level resources, the term hardware platform will sometimes be used for preciseness. These target and hardware platform definitions have been made as simple as possible. Chapters 9 and 11 will outline a more elaborate approach to target platform specification based on Hughes' $sp$ approach [103]. The extensions cover both platforms built from other software systems (e.g. database management systems) and distributed computer systems.
A set of visit counts specifies which platform services that are used by which primitive component operations, and how many times they are visited on average per operation execution. In a realisation of the framework, visit descriptions could be entered through menus provided by the lowest-level DFD processes or primitive objects of the design specification. Visit descriptions are represented as visit matrices $V_\gamma$ inside the framework. Elements $v_{o'}^o$ of visit matrices map primitive component operations onto services at the hardware level.

![Diagram](image)

Figure 8.1: Design specification for the Customer transaction system of fig. 3.5. (This figure is a duplicate of fig. 7.5 for easy reference.)

The three will be considered in separate: First, sec. 8.1.1 defines the entry matrices. Sec. 8.1.2 then presents the activity matrices, while sec. 8.1.3 finally introduces the visit matrices for primitive components. These sections will use the Customer transaction system of sec. 7.2 as an example. Fig. 8.1 is therefore a duplicate of fig. 7.5 for easy reference.

### 8.1.1 Entry matrices

Entry and activity descriptions together specify how many uses of each internal operation of some component correspond to one use of each of its external operations. First, the entry descriptions specify which internal operations are used initially when each external operation is used. Then, the activity descriptions specify which internal operations are used next after each internal operation has been used.

Entry matrices describe how the external operations of a component initiate internal operations. The entry matrix for component $\gamma$, $E_\gamma$, is a matrix $E_\gamma = [e_{o,d'}]$, $e_{o,d'} \geq 0$, $o \in O_\gamma$, $d' \in O_\gamma$. Element $e_{o,d'}$ of this matrix represents the average number of times operation $d'$ on some subcomponent of $\gamma$ is initiated per execution of operation $o$ on component $\gamma$. An entry matrix must be provided for every component in the design specification.

**Example**

The top-level component of the banking example provides four external operations to the organisation,
as indicated by the four triggering flows of fig. 3.5. The corresponding entry matrix becomes

\[
E_{\gamma_1} = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix},
\]

(8.1)

because each of the four operations of the top-level component \( \gamma_1 \) next activate execution of subcomponents \( \gamma_2 \ldots \gamma_6 \) respectively, as shown by entry matrix \( E_{\gamma_1} \).

The \textit{make withdrawal} subcomponent of fig. 3.8 provides only one external operation, hence

\[
E_{\gamma_6} = [1 \ 0 \ 0].
\]

\( \square \)

\[
\begin{align*}
E_{\gamma_1} &= \begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0
\end{bmatrix}, & A_{\gamma_1} &= \begin{bmatrix}
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0
\end{bmatrix}, \\
E_{\gamma_6} &= [1 \ 0 \ 0], & A_{\gamma_6} &= \begin{bmatrix}
0 & 0.96 & 0.04 \\
0 & 0 & 1 \\
0 & 0 & 0
\end{bmatrix}
\]

Table 8.2: Entry and activity matrices for the customer transaction specification of fig. 8.1.

8.1.2 Activity matrices

Activity matrices describe how the internal operations of a component activate one another. The \textbf{activity matrix for component} \( \gamma \), \( A_{\gamma} \), is a matrix \( A_{\gamma} = [a_{\sigma', \sigma''}] \), \( a_{\sigma', \sigma''} \geq 0 \), \( \sigma', \sigma'' \in O_\gamma^s \). Element \( a_{\sigma', \sigma''} \) of this matrix represents the average number of times operation \( \sigma'' \) on some subcomponent of \( \gamma \) is executed next after operation \( \sigma' \) on some (possibly the same) subcomponent of component \( \gamma \). An activity matrix must be provided for every component in the design specification. In sec. 12.3.2 of part IV however, it will turn out that this need not be for particular \textit{realisations} of the basic framework. In the PrM realisation of this thesis, e.g., branching probabilities are \textit{only entered at the lowest level}, and activity matrices are derived from those probabilities at all levels of the design specification.

Introducing the concept of activity matrices has the advantage that it opens for the use of existing results derived by Ofstedahl & Solvberg [135, 177]. Some of the results in chapter 10 will be inspired by this work.

\textbf{Example}
The top-level component of the banking example of fig. 3.5 has the following activity matrix,

\[
A_{\gamma_1} = \begin{bmatrix}
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 \\
\end{bmatrix},
\]  

(8.3)

because all components \(\gamma_2\) to \(\gamma_5\) next activate component \(\gamma_6\), write statement on termination as shown in fig. 3.5.

It is assumed that the customer’s balance is adequate for 96% of the withdrawals. The other 4% are not successful. Since the Make withdrawal subcomponent of fig. 3.8 therefore contains a branching point, its matrix formulation is a little more complicated:

\[
A_{\gamma_6} = \begin{bmatrix}
0.96 & 0.04 & 0 \\
0 & 0 & 1 \\
0 & 0 & 0 \\
\end{bmatrix}.
\]  

(8.4)

The entry and activity matrices of the Customer transaction specification are summarised in table 8.2. □

\begin{center}

**Table 8.3:** Hardware platform for the Customer transaction model of fig. 3.5.

<table>
<thead>
<tr>
<th>(V_{\gamma_2})</th>
<th>(V_{\gamma_3})</th>
<th>(V_{\gamma_4})</th>
<th>(V_{\gamma_5})</th>
<th>(V_{\gamma_6})</th>
</tr>
</thead>
<tbody>
<tr>
<td>([60000 \ 5.0 \ 60])</td>
<td>([100000 \ 17.5 \ 60])</td>
<td>([20000 \ 2.5 \ 20])</td>
<td>([10000 \ 0.5 \ 20])</td>
<td>([15000 \ 2.0 \ 10])</td>
</tr>
</tbody>
</table>

**Table 8.4:** Visit matrices for the customer transaction specification of fig. 8.1.

\end{center}

### 8.1.3 Visit matrices for primitive components

Visit matrices for primitive components describe how their operations visit services provided by the hardware platform. The visit matrix for primitive component \(\gamma\), \(V_{\gamma}\), is a matrix \(V_{\gamma} = [v_{\sigma}^{o}]\), \(v_{\sigma}^{o} \geq 0\), \(o \in O_{\gamma}\), \(\sigma \in \Sigma_{P}\). Element \(v_{\sigma}^{o}\) of this matrix represents the average number of times service \(\sigma\) is visited per execution of operation \(o\) on primitive component \(\gamma\). A visit matrix must be provided only for the primitive components of the design specification.
Example

The hardware platform for the Customer transaction system consists of a CPU, a Disc, and a (set of) Terminal line(s), as shown in table 8.3. Note that the three correspond exactly to the primitive \( sp \) modules of fig. 4.3 in sec. 4.8.2.

Visit matrix \( \mathbf{V}_{\gamma} \) specifies the visit counts for the Write statement component in terms of this platform. From experience with similar systems, it is assumed that the component requires on average 10000 CPU instructions, as well as 0.5 disc accesses and 20 terminal line transmissions per execution. The visit matrix therefore becomes

\[
\mathbf{V}_{\gamma} = \begin{bmatrix} 10000 & 0.5 & 20 \end{bmatrix} \quad (8.5)
\]

These estimates include the additional workload induced by the software platform and operating system software. Sec. 9.3 will consider explicit software platform and operating system workload models to simplify parameter estimation. The low number of disc accesses is due to a very high cache-hit rate of 80% on this system.

The visit counts for all the primitive components of the customer transaction specification are given in table 8.4. \( \Box \)

\[
\begin{align*}
\mathbf{V}_{\gamma} &= \begin{bmatrix} \mathbf{V}_{\gamma_1} \\ \vdots \\ \mathbf{V}_{\gamma_n} \end{bmatrix} \\
\mathbf{V}^*_{\gamma} &= \begin{bmatrix} \mathbf{V}_{\gamma_1} \\ \vdots \\ \mathbf{V}_{\gamma_n} \end{bmatrix} \\
\mathbf{C}_{\gamma} &= \mathbf{I}_{\gamma}(\mathbf{E} - \mathbf{A}_{\gamma})^{-1} \\
\mathbf{V}_{\gamma} &= \mathbf{C}_{\gamma} \mathbf{V}^*_{\gamma}
\end{align*}
\quad (8.6) \quad (8.7) \quad (8.8)
\]

Table 8.5: Workload derivation technique.

8.2 Workload Derivation

A workload derivation method is supported to transform the performance parameters of the previous section into a workload model for the projected application. As already mentioned, this workload model is on the form of visit counts to be used in a queueing network analysis.

The derivation technique will be presented in detail in secs. 8.2.1, 8.2.2, and 8.2.3. It proceeds through bottom-up collapsing of the annotated design specification, starting with the primitive components. When all the subcomponents of component \( \gamma \) have been collapsed, an internal visit matrix \( \mathbf{V}^*_{\gamma} \) is derived by stacking the visit matrices of its subcomponents on top of one another as in eq. 8.6,

\[
\begin{align*}
\mathbf{V}^*_{\gamma} &= \begin{bmatrix} \mathbf{V}_{\gamma_1} \\ \vdots \\ \mathbf{V}_{\gamma_n} \end{bmatrix} \\
\mathbf{V}^*_{\gamma} &= \begin{bmatrix} \mathbf{V}_{\gamma_1} \\ \vdots \\ \mathbf{V}_{\gamma_n} \end{bmatrix} \\
\mathbf{C}_{\gamma} &= \mathbf{I}_{\gamma}(\mathbf{E} - \mathbf{A}_{\gamma})^{-1} \\
\mathbf{V}_{\gamma} &= \mathbf{C}_{\gamma} \mathbf{V}^*_{\gamma}
\end{align*}
\quad (8.6) \quad (8.7) \quad (8.8)
\]

Table 8.5: Workload derivation technique.
A complexity matrix $C_{\gamma}$ for component $\gamma$ specifies how many times each subcomponent operation is executed when each of component $\gamma$'s own operations are executed. It is derived as follows: The total number of subcomponent operations executed $C_{\gamma}$ equals the sum of operations executed initially $E_{\gamma}$ and subsequently $C_{\gamma}A_{\gamma}$ so that

$$C_{\gamma} = E_{\gamma} + C_{\gamma}A_{\gamma},$$

giving eq. 8.7,

$$C_{\gamma} = L_{\gamma}(E - A_{\gamma})^{-1}.$$  

The visit matrix $V_{\gamma}$ for the component is then calculated by eq. 8.8,

$$V_{\gamma} = C_{\gamma}V_{\gamma}^*,$$

preparing for collapsing at the next higher level of components. Since the component DAG is cycle-free, this algorithm always terminates successfully when the top-level components $\gamma_i$ have been collapsed. The resulting top-level visit matrices define the visit counts needed for performance model analysis.

Unless the computer will be dedicated to the projected application, additional visit counts must be provided for each existing application running on the computer. This will be treated in sec. 8.3.

Three kinds of derived parameters are required. First, sec. 8.2.1 defines internal visit matrices for components. Sec. 8.2.2 the introduces the complexity matrices representing the average behaviour of components. Finally, sec. 8.2.3 introduces visit matrices for non-primitive components as well.

### 8.2.1 Internal visit matrices for components

Internal visit matrices for components describe how their internal operations visit services provided by the hardware platform. The **internal visit matrix for component** $\gamma$, $V_{\gamma}^*$, is a matrix $V_{\gamma}^* = \left[v_{\sigma}^o\right]$, $v_{\sigma}^o \geq 0$, $o \in O_{\sigma}$, $\sigma \in \Sigma_p$. Element $v_{\sigma}^o$ of this matrix represents the average number of times service $\sigma$ is visited per execution of operation $o$ on some subcomponent of $\gamma$.

Each internal operation $o$ of component $\gamma$ is an external operation of one of its subcomponents $\gamma_i$. The visit counts of operation $o$ are therefore represented as a row of the visit matrix $V_{\gamma_i}$ of subcomponent $\gamma_i \in \Gamma_{\gamma}$. As a result, the internal visit matrix of component $\gamma$ is formed by stacking the visit matrices of its subcomponents on top of one another in the appropriate order:

$$V_{\gamma}^* = \begin{bmatrix} V_{\gamma_1}^* \\ \vdots \\ V_{\gamma_n}^* \end{bmatrix}, \quad \gamma_i \in \Gamma_{\gamma} \quad (8.9)$$

The term “appropriate order” is well-defined here, since the rows of matrix $V_{\gamma}^*$ represent internal operations of component $\gamma$, and these internal operations are one-to-one with the external operations of components $\Gamma_{\gamma}$ represented by rows of matrices $V_{\gamma_i}$.

**Example**
Table 8.4 gives

\[
V_{\gamma_7} = \begin{bmatrix} 15000 & 2.0 & 10 \end{bmatrix} \quad (8.10)
\]

\[
V_{\gamma_8} = \begin{bmatrix} 10000 & 2.0 & 0 \end{bmatrix} \quad (8.11)
\]

\[
V_{\gamma_9} = \begin{bmatrix} 5000 & 3.0 & 10 \end{bmatrix}. \quad (8.12)
\]

Stacking these three matrices on top of one another gives

\[
V_{\gamma}^* = \begin{bmatrix} 15000 & 2.0 & 10 \\ 10000 & 2.0 & 0 \\ 5000 & 3.0 & 10 \end{bmatrix}, \quad (8.13)
\]

since \( \Gamma_{\gamma} = \{ \gamma_7, \gamma_8, \gamma_9 \} \) The internal visit matrix of the top-level component \( \gamma_1 \), however, cannot be

created until a visit matrix for its subcomponent \( \gamma_9 \) has been derived. \( \square \)

### 8.2.2 Complexity matrices

Sec. 8.1 introduced entry matrices to describe which internal operations are executed initially for each external operation on some component, and activity matrices to describe which internal operations are executed next for of its internal operations. These annotations are now used to describe the average internal behaviour of a component for each of its external operations.

Complexity matrices describe the number of uses of internal operations that correspond to one use of each external operation of a component. The complexity matrix for component \( \gamma \), \( C_\gamma \), is a matrix \( C_\gamma = [c_{o,d}] \), \( c_{o,d} \geq 0 \), \( o \in O_\gamma, \ \ d \in O_\gamma^* \). Element \( c_{o,d} \) of this matrix represents the average number of times operation \( d \) on some subcomponent of \( \gamma \) is used per execution of operation \( o \) on component \( \gamma \).

Since each execution of operation \( d \) on some subcomponent \( \gamma \) must be either externally or internally induced, the relation

\[
c_{o,d} = e_{o,d} + \sum_{o' \in O_\gamma^*} c_{o,o'd}a_{o',d} \quad (8.14)
\]

must hold for all external operations \( o \in O_\gamma \) and internal operations \( d \in O_\gamma^* \) of component \( \gamma \), or in matrix notation

\[
C_\gamma = E_\gamma + C_\gamma A_\gamma. \quad (8.15)
\]

It follows that (Lowe [127] and eq. 5.1)

\[
C_\gamma = E_\gamma (I - A_\gamma)^{-1}. \quad (8.16)
\]

This equation gives the complexity matrix for component \( \gamma \) as a function of its entry and activity counts. It holds for all components in a design specification, and is a main result in the sequel.

The computational complexity brought into the framework by the matrix inversion of eq. 8.16 is justified by components with “loops,” i.e. components whose activity matrices can not be triangularised by reordering the internal operations. For such more complex models, matrix
inversions are more convenient than graph traversal algorithms. In addition, it is important to have an analytically tractable representation of the projected application. In particular, this will become clear when sensitivity analysis is introduced in chapter 10.

Example
The complexity matrix $C_{\gamma_n}$ of the Make withdrawal subcomponent is derived from its entry and activity matrices $E_{\gamma_n}$ and $A_{\gamma_n}$ by eq. 8.16. In the banking example, the complexity matrices therefore become

$$C_{\gamma_n} = E_{\gamma_n} (I - A_{\gamma_n})^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}.$$  \hspace{1cm} (8.17)

For the single operation provided by the Make withdrawal subcomponent,

$$C_{\gamma_n} = E_{\gamma_n} (I - A_{\gamma_n})^{-1} = \begin{bmatrix} 1 & 0.96 & 1 \end{bmatrix}.$$  \hspace{1cm} (8.18)

The correctness of these matrices can easily be asserted by inspection of figs. 3.5 and 3.8. \Box

For use in chapters 9 and 10, the **complexity matrix from component $\gamma$ to subcomponent $\gamma'$** is also defined as $C'_{\gamma'} = [c_{o',o}]$, $c_{o',o} \geq 0$, $o \in O_{\gamma}$, $o' \in O_{\gamma'}$, so that the corresponding elements of matrices $C_{\gamma}$ and $C'_{\gamma'}$ are equal, and the matrix $C'_{\gamma'}$ represents the average number of times each operation provided by subcomponent $\gamma'$ is used per execution of each component provided by component $\gamma$. Hence, the rows of matrix $C'_{\gamma'}$ form a subset of the rows of $C_{\gamma}$. According to this definition, $\gamma'$ must be directly subordinate to component $\gamma$. (I.e. $\gamma' \in \Gamma_{\gamma}$, directly and indirectly superior and subordinate components were defined in sec. 7.2.) Sec. 9.2 will generalise this definition to indirect subordinates as well.

Example
The component-subcomponent complexity matrices for the Customer transaction specification of fig. 8.1 are easily extracted from the above matrices $C_{\gamma_n}$ and $C_{\gamma_m}$.

$$C_{\gamma_n}^{\gamma} = [1 \ 0 \ 0 \ 0]^T$$  \hspace{1cm} (8.19)

$$C_{\gamma_m}^{\gamma} = [0 \ 1 \ 0 \ 0]^T$$  \hspace{1cm} (8.19)

$$C_{\gamma_n}^{\gamma} = [0 \ 0 \ 1 \ 0]^T$$  \hspace{1cm} (8.19)

$$C_{\gamma_m}^{\gamma} = [0 \ 0 \ 0 \ 1]^T$$  \hspace{1cm} (8.19)

$$C_{\gamma_n}^{\gamma} = [0 \ 1 \ 1 \ 1]^T$$  \hspace{1cm} (8.19)

$$C_{\gamma_m}^{\gamma} = [1 \ 1 \ 1 \ 1]^T$$  \hspace{1cm} (8.19)

$$C_{\gamma_n}^{\gamma} = [1 \ 1 \ 1 \ 1]^T$$  \hspace{1cm} (8.19)

$$C_{\gamma_m}^{\gamma} = [1 \ 1 \ 1 \ 1]^T$$  \hspace{1cm} (8.19)

$$C_{\gamma_n}^{\gamma} = [1 \ 1 \ 1 \ 1]^T$$  \hspace{1cm} (8.19)

$$C_{\gamma_m}^{\gamma} = [1 \ 1 \ 1 \ 1]^T$$  \hspace{1cm} (8.19)

$$C_{\gamma_n}^{\gamma} = [1 \ 1 \ 1 \ 1]^T$$  \hspace{1cm} (8.19)

$$C_{\gamma_m}^{\gamma} = [1 \ 1 \ 1 \ 1]^T$$  \hspace{1cm} (8.19)

8.2.3 Visit matrices for non-primitive components

Visit matrices for non-primitive components describe how their internal operations visit services that are provided by the hardware platform. The **visit matrix for non-primitive component $\gamma$, $V_{\gamma}$**, is a matrix $V_{\gamma} = [v^o_{\sigma}]$, $v^o_{\sigma} \geq 0$, $o \in O_{\gamma}$, $\sigma \in \Sigma_p$. Element $v^o_{\sigma}$ of this matrix represents the average of times service $\sigma$ is visited per execution of operation $o$ on non-primitive component $\gamma$.

Since each external operation of component $\gamma$ is implemented in terms of its internal operations, the following relation must hold:

$$v^o_{\sigma} = \sum_{\sigma' \in O_{\gamma}} c_{o,o'} v^o_{\sigma'},$$  \hspace{1cm} (8.20)
or in matrix notation
\[ V_\gamma = C_\gamma V_\gamma^*. \]  \hspace{1cm} (8.21)

**Example**

The complexity matrix derived in the previous sec. 8.2.2 is now multiplied with the internal visit matrix derived in sec. 8.2.1. The resulting visit matrix \( V_\gamma \) estimates the resource demands of the corresponding software component for each of its operations, preparing for collapsing at the next higher level.

In the customer transaction example, the visit matrix of non-primitive component \( \gamma_5 \) becomes
\[ V_{\gamma_5} = C_{\gamma_5} V_{\gamma_5}^* \begin{bmatrix} 29600 & 6.92 & 20 \end{bmatrix}. \]  \hspace{1cm} (8.22)

As a result, the internal visit matrix for top-level component \( \gamma_1 \) can be created, preparing for collapsing at the highest level of the design specification. The resulting complexity and visit matrices for the customer transaction specification are given in table 8.6. The top-level complexity matrix \( C_{\gamma_1} \) shows that execution of one of its operations results in one execution of the corresponding subcomponent and one execution of the write statement subcomponent. The resulting top-level visit matrix \( V_{\gamma_1} \) contains the visit counts needed for performance queueing network analysis. □

A consequence of eq. 8.21 is that a non-primitive component uses only services that are used by at least one of its subcomponents, and conversely that a subcomponent uses only services that are also used by its superior component. This is true both for directly and indirectly superior and subordinate components. This will be called the subcomponent rule, and it will become important when considering distributed information systems in chapter 11.

\[ C_{\gamma_1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix} ; \quad V_{\gamma_1} = \begin{bmatrix} 70000 & 5.5 & 80 \\ 110000 & 18. & 80 \\ 30000 & 3.0 & 40 \\ 39600 & 7.4 & 40 \end{bmatrix} \]

Table 8.6: Workload derivation results.

### 8.3 Workload and Performance Models

The basic framework uses the workload derivation method of the previous sec. 8.2 to predict the performance of the projected application during development. This method is based on a set of workload scenarios; a projected workload module; a set of existing workload modules; a resulting workload model; a performance model; a set of performance measures produced, and a performance analysis technique.
8.3. WORKLOAD AND PERFORMANCE MODELS

8.3.1 Workload scenarios

A performance prediction depends on how often the organisation will use (or uses) each function provided by the projected and existing applications. The basic framework represents organisation workload as simply as possible. It is assumed that each operation corresponds to a set of similar functions, and that the organisation's use of this set follows a quantifiable pattern. Thus organisation workload is described in terms of the workload intensities for the operations provided by the design specification.

Each operation is described by a workload intensity in accordance with sec. 7.3:

- A terminal intensity is appropriate when an operation \( o \) is frequently used by each of a small number of users. This type of workload is described as a tuple \((n_o, \theta_o)\) where \( n_o \) is the average number of users of the operation and \( \theta_o \) is their average think time between each use of the operation.

- A batch intensity is used when an operation \( o \) is executed as a batch job with long running time. Due to their running time, the batch jobs can be regarded as always executing in the computer system for the time interval for which the system is considered. If a batch type customer leaves the system, it is therefore immediately reentered into the system. This workload type is described by the average number of batch jobs \( n_o \) present in the system. Hence it can be regarded as a special case of the terminal type \((n_o, \theta_o)\) with think time \( \theta_o = 0 \).

- A transaction intensity is applied when an operation \( o \) is infrequently used by each of a large number of users. This type of workload is described by the average arrival rate \( \lambda_o \). This is the most versatile among the workload intensities, since it is sufficient in most cases and since the rates of operation usages are the easiest to handle formally (Lazowska et al. [119]).

For organisation workloads with only terminal and batch workload intensities, the numbers \( n \) of terminal and batch users define the maximum population vector for design specification \( S, \tilde{N}_S \). This maximum population vector will become important in the distributed information system analysis of chapter 11. When considering queueing network analysis in sec. 8.3.7, such organisation workloads will correspond to closed networks.

For organisation workloads with at least one transaction type operation, the maximum population vector is undefined, but two possibilities still exist:

- the transaction workload intensities can be approximated by workload intensities of the terminal type, or

- a maximum number of transaction users, \( \bar{n}_o \), may be provided for each transaction type operation \( o \). In this case, the transaction workload intensity becomes a pair \((\lambda_o, \bar{n}_o)\).

An example of the former will be given in sec. 9.4.4.6.

Example
The organisation workload $\Omega_S$ on the Customer transaction specification of fig. 3.5 comprises a set of workload intensities,

$$\Omega_S = (\omega_{01}, \omega_{02}, \omega_{03}, \omega_{04}).$$  \hspace{1cm} (8.23)

The bank currently initiates 1800 and terminates 540 accounts, and processes 72000 deposits and 900000 withdrawals a day on average. The initial assumption is that this will increase with 8% before the customer transaction system is in production, giving arrival rates of $0.162, 0.054, 2.16$, and $27.0$ operations per second, under the very simplistic assumption that all transactions are processed during the same 10 hour day they are requested and are evenly distributed throughout the day. Flostrand [80] considers the case of unevenly distributed organisation workload. Also, this is the topic of a related thesis [41].

The workload intensities for the four top-level operations therefore become

$$\begin{align*}
\omega_{01}: & \text{ type is transaction, } \lambda_{01} = 0.162 \\
\omega_{02}: & \text{ type is transaction, } \lambda_{02} = 0.054 \\
\omega_{03}: & \text{ type is transaction, } \lambda_{03} = 2.16 \\
\omega_{04}: & \text{ type is transaction, } \lambda_{04} = 27.0
\end{align*}$$

However, this organisation workload is highly variant with respect to time of the day, day of the week, and week of the year. A concept of workload scenarios is therefore needed to represent periodic workload fluctuations.

**Example**

Typically, the organisation's use of the information system will vary throughout a 24-hour period. E.g., interactive users might dominate through the working hours, with large batch jobs running in the night-time. Smaller variations can also be observed between different weekdays and different weeks of the year, but the discussion focuses on day-to-day variations. □

Intervals during which the organisation uses the computerised information system in a “similar manner” are represented as workload scenarios. For such intervals, the organisation workload can be quantified as average workload intensities according to sec. 7.3. If the information system users are too dissimilar in their information system use, such averages are difficult to quantify. This is the reason why the information systems considered in this thesis were assumed to have many, heterogeneous users in chapter 3.

A workload scenario is a set of workload intensities for each of the functions used by the organisation. Therefore, a workload scenario characterises how often the organisation uses that function during the corresponding time period. Only some workload scenarios are performance critical.

**Example**

Examples are the first hour of the day when a lot of interactive users log into the system to retrieve lists of today's clients from the database, as well as the high-time when many large batch jobs must all be processed before the morning. On the other hand, performance may not be critical for the
lunch-hour workload scenario. □

Only performance critical workload scenarios are considered in a software performance study. For each of them, a performance goal must be established. A performance goal is a required performance for some performance critical workload scenario. Just as an organisation workload is described as a set of workload intensities for each of the operations provided, a performance goal is characterised by a **required responsiveness** for each of the functions used.

Descriptions of performance critical workload scenarios and their corresponding performance goals constitute the **objective** of the performance prediction study. Early establishment of such objectives is crucial to guide the modelling process (Jain [109]).

This demonstrates how the above hour-to-hour, day-to-day, and week-to-week fluctuations in workload intensities are handled. The remainder of the thesis will focus on performance prediction and improvement **within one** particular workload scenario.

### 8.3.2 The projected workload module

Applying the workload derivation technique of the previous sec. 8.2, the visit matrices $V_{\gamma}$ for top-level components $\gamma \in \Gamma^*_S$ of design specification $S$ are derived. These top-level visit matrices will be used to establish a **workload module** $W_S$ for the projected application.

Visit matrices for workload modules describe how its operations visit services that are provided by the hardware platform. The **visit matrix for workload module** $W_S$, $V_S$, is a matrix $V_S = [v^o_{\sigma}]$, $v^o_{\sigma} \geq 0$, $o \in O_S$, $\sigma \in \Sigma_P$. Element $v^o_{\sigma}$ of this matrix represents the average number of times service $\sigma$ is visited per execution of operation $o$ on design specification $S$.

Each top-level operation $o$ of design specification $S$ is an external operation of one of its top-level components $\gamma_i \in \Gamma^*_S$. The visit counts of operation $o$ are therefore represented as a row of the visit matrix $V_{\gamma_i}$ of some top-level component $\gamma_i \in \Gamma^*_S$. As a result, the visit matrix of workload module $W_S$ is formed by stacking the visit matrices of its subcomponents on top of one another in the appropriate order:

$$V_S = \begin{bmatrix} V_{\gamma_1} \\ \vdots \\ V_{\gamma_n} \end{bmatrix} \quad \gamma_i \in \Gamma^*_S \quad (8.24)$$

**Example**

Since the design specification for the **Customer transaction** application has only a single top-level component, the corresponding workload module visit matrix becomes trivial.

$$V_S = V_{\gamma_1} = \begin{bmatrix} 70000 & 5.5 & 80 \\ 110000 & 18. & 80 \\ 30000 & 3.0 & 40 \\ 39600 & 7.4 & 40 \end{bmatrix} \quad (8.25)$$

□
8.3.3 The existing workload modules

In case the projected application will run alone on a dedicated computer, the workload module for design specification $S$ suffices for performance prediction. In most cases however, the projected application is only one among several, already existing applications which compete for computing resources. Therefore, additional existing workload modules are needed in addition to the projected one. Such additional workload modules typically have already been established for capacity planning purposes, thus providing a link between the fields of software performance engineering and capacity planning.

Such workload modules $W_{A'}$ for existing applications $A'$ are also represented as visit matrices $V_{A'}$. Hence projected and existing workload modules have the same format. Modelling existing workloads on a computer is the topic of much research, as was pointed out in sec. 4.2. Also, it is studied in a related thesis [190], and will not be elaborated on here.

8.3.4 The workload model

The projected and existing workload modules of the previous secs. 8.3.2 and 8.3.3 together with the organisation workload of sec. 8.3.1 constitute a workload model. Hence a workload model corresponds a particular workload scenario, as was commented on in the closing of sec. 8.3.1.

In accordance with sec. 7.3, a workload model $W$ comprises a set of workload modules $W \in W$ for projected and existing applications, and an organisation workload vector $\Omega_W$. This organisation workload vector is defined from the workload intensities of operations provided by workload modules $W \in W$. This set of operations is $O_W$.

In addition, a workload model visit count matrix for workload model $W$ is formed by stacking the visit matrices of workload modules on top of one another in the appropriate order:

$$V_W = \begin{bmatrix} V_{W_1} \\ \vdots \\ V_{W_n} \end{bmatrix}, \quad W_i \in W \quad (8.26)$$

8.3.5 The performance model

To make a performance prediction, however, a model is also needed of the computer hardware on which the software system will run. Such a computers model has typically also already been established for capacity planning purposes, thus providing a link between the two worlds of software performance engineering and capacity planning. At the hardware level, the basic framework focuses on

- ease of modelling and parameter capture, and
- fast solution techniques for interactive design feedback [180].
8.3. WORKLOAD AND PERFORMANCE MODELS

The former is important to reduce the added cost of software performance engineering, and because early parameter estimates are not sufficiently accurate to justify sophisticated performance models. The latter point is due to the large set of design alternatives. Also, it is mandatory that performance predictions become available to developers while they are still valid.

\[
R_s = \{ 1.21, 1.50, 0.58, 0.66 \} \\
Z_p = \{ 5.49 \cdot 10^{-6}, 5.23 \cdot 10^{-3}, 0.01 \} \\
u_{CPU} = 0.86 \quad u_{disc} = 0.52
\]

Table 8.7: Performance analysis results.

Operational queueing network analysis, introduced by Denning & Buzen [73] and presented in sec. 4.4.2, is a natural choice of performance modelling approach given the above requirements, because:

- the quantities involved are estimates of averages rather than stochastic variables;
- the treatment of variable quantities is exact rather than probabilistic, and
- the quantities involved are directly measurable on the operational system if it exists.

In particular, the two first points make operational analysis consistent with the basic framework presented so far in secs. 8.1 and 8.2. The third point makes performance analysis easy to apply for computers and applications that exist, complementing the basic framework whose intention was to make performance analysis easy for projected applications.

The resulting performance models are simple. The few parameters needed correspond to observable quantities. The associated analysis techniques are fast and intuitive. The summary of this section lends most of its terminology and symbols from Lazowska et al. [119].

The hardware resources constituting the computer are represented in queueing network \( M \) as a set of service centres \( k \in K_M \). Service centres are either queueing or delay. Service centre \( k \) is characterised by the average service time \( \tau_k \), \( k \in K_M \), spent on each visit to the corresponding hardware resource.

The concept of service centres of course closely resemble the hardware level services of secs. 7.3 and 7.4. However, the two are not exactly matched because the latter constitute only an abstract interface to the former. In this way, a single hardware platform is made compatible with several alternative performance models. This is of course important since the basic framework must support comparison of alternative hardware configurations for the projected application. Therefore, a platform mapping \( \mu \) is established, mapping

- hardware platform services \( \sigma \) onto performance model service centres \( k = \mu(\sigma) \), and
(hence)
• workload model visit counts \( v^o_k \) onto performance model visit counts \( v^o_k = \mu(v^o_k) \), so that
\[
\mu(v^o_k) = v^o_k, \tag{8.27}
\]

Note that this mapping is not necessarily one-to-one. Hence, different platform services may be mapped onto the same performance model service centre. This further enhances the generality and applicability of the approach. The advantages of the platform mapping concept will become even more clear when distributed systems are considered in chapter 11.

The software functions provided by the combined hardware and software system are represented as the set of operations \( O_W \) of workload model \( W \). Of course, this is a deviation from the conventional concept of \( C \) customer classes, 1\ldots C. The concept of operations is preferable for the purpose of this thesis, as it is closer to common software development terminology and links performance modelling to the workload modules of secs. 8.3.2 and 8.3.3.

An operation \( o \) is characterised by the average numbers of visits \( v^o_k \) made by the corresponding operation to each of the service centres \( k \in K_M \) each time it is used. The concept of performance model operations is exactly the same as the workload module operations of the previous secs. 8.3.2, 8.3.3 and 8.3.4.

Finally, the organisation using the software functions is represented as the organisation workload vector \( \Omega_W \) for workload model \( W \). This vector contains a workload intensity for each function provided by the projected and existing applications. Again, these workload intensities are exactly those of the previous secs. 8.3.1 and 8.3.4.

![Figure 8.8: Queueing network performance model for the hardware platform of table 8.3.](image)

**Example**

A queueing network for the bank example is shown in fig. 8.8. The network consists of three service centres representing the CPU, the disc, and the set of terminal lines of the computer in the bank. Hence the service centres of the queueing network are one-to-one with the services provided by hardware platform \( P \), and the corresponding platform mapping \( \pi \) therefore becomes trivial.

In the example, the service times for the CPU, disc, and terminal line are \( 7.5 \cdot 10^{-7}, 2.5 \cdot 10^{-3}, \) and \( 0.01 \) seconds respectively:
\[
\tau_1 = 7.5 \cdot 10^{-7}, \quad \tau_2 = 2.5 \cdot 10^{-3}, \quad \tau_3 = 0.01. \tag{8.28}
\]
8.3.6 Performance measures

The analysis outputs are also closely connected to the modelling constructs in a natural way. [119] summarises a large set of average quantities of which this thesis will only depend on a few:

- response times $r_o$ and throughputs $x_o$ for operation $o$, which predict the responsiveness of the corresponding software function, and

- sojourn times $z_k$ for service centre $k$, which estimate the average time spent in the corresponding hardware resource per visit, queueing included. (Note that this definition differs from [119], which uses the concept of residence times $r_k$ here.)

In general, analysis provides additional measures for service centres, such as total and per class queue lengths $q_k$ and $q_k^o$, and total and per class utilisations $u_k$ and $u_k^o$, as well as total and per class numbers in system, $q$ and $q_o$, at the system level.

In what follows, let $R_W$ be the responsiveness vector for workload model $W$. The elements of this vector are response times $r_o$ for transaction operations $o$ and response times $r_o$ or throughputs $x_o$ for terminal and batch operations, $o \in O_W$. For terminal and batch operations, response times $r_o$ and throughputs $x_o$ are interchangeable due to the interactive response time law $r_o + \theta_o = n_o x_o$ of sec. 4.4.2.

Also, let $Z_M$ be the sojourn time vector for performance model $M$. The elements of this vector are service centre sojourn times derived from analysis of performance model $M$. For open queueing networks, this vector is well defined since service centre sojourn times are the same for all operations in open networks. In closed and mixed networks, however, this is not the case as is commented on in sec. B.1 in the appendix. For such networks, service centre sojourn times will vary with operation type. However, for (realistic) organisation workloads with large numbers of terminal or batch users, these variations can be assumed to be negligible. Nevertheless, it must be remembered that the vector $Z_M$ is only an approximation in such cases.

The sojourn time vector for hardware platform $P$, $Z_P$, is derived by mapping these service centre sojourn times back onto hardware platform service sojourn times,

$$Z_P = \mu^{-1}(Z_M) \tag{8.29}$$

so that

$$z_o = z_k, \ k \in \mu^{-1}(\sigma). \tag{8.30}$$

This sojourn time vector is important for the parameter capture support techniques of chapter 9 and 10.

The additional service centre performance measures are not important in this thesis. However, they are sometimes mentioned in subsequent discussions for generality.
Example
Performance analysis in the bank example predicts response times of less than one and a half second for initiation and termination of accounts, and responses around half a second for deposits and withdrawals, as shown in table 8.7. This is clearly acceptable. However, the analysis predicts a CPU utilisation as high as 86%. This indicates that the response time estimates are highly sensitive to variations in CPU utilisation, and that small workload fluctuations may degrade performance significantly. Therefore, continued monitoring of expected performance is required based on performance predictions from refined design specifications and more accurate performance parameters. □

8.3.7 Performance analysis
Appendix B presents the performance analysis techniques in the notation of this thesis. The algorithms presented there take as inputs the parameters defined in sec. 8.3.5 and outputs the parameters of sec. 8.3.6.

Product-form queueing network analysis  The visit ratios used $V_i$ in operational analysis of separable queueing networks replace the transition probabilities $p_{ij}$ of stochastic queueing networks as presented in sec. 4.4. Systems of linear equations exist for deriving visit counts from transition probabilities (e.g. Jain [109]). In case of central-server queueing networks where all operations always visit the CPU service centre between visits to other centres, the inverse transformation is also possible. Since the basic framework only produces visit counts, this inverse transformation is important for the basic framework to be interfaced with stochastic product-form queueing network models (as opposed to operational, separable models). Since the basic framework of this chapter considers non-distributed computers only, central-server models are encouraged. Therefore, non-operational queueing network analysis is feasible as an alternative to the techniques of this chapter. Indeed, the realisation of the framework in part IV will be based on such general queueing networks, since the IMSE QNET tool is not operationally oriented.

In chapter 11, the distributed performance models will be represented as such (weakly) interconnected central-server queueing networks. Hence the above criterion is satisfied for distributed models also.

Synchronisation and blocking  Smith [167] has studied the possibility of making simple extensions to computer queueing network models in order to cope with aspects such as memory competition, concurrency, synchronisation, and blocking. Similar techniques are presented in [119]. Since most of them are approximate, some measure of approximation error is needed. Sec. 14.3 will point out that the framework could be extended for this purpose with statistical distributions and confidence intervals, or with error bounds. However, the above techniques are all based on modifying the hardware performance model to capture aspects of software dynamics. Hence they are low-level, and best suited for the dynamics of basic software, such as the operating system.
Other important synchronisation and blocking phenomena occur in higher-level software, i.e., the target platform and application software systems. For instance, locking in database management systems will sometimes affect overall system performance significantly. Also, interprocess communication waits are important for the responsiveness of multi-process application software. Assessing synchronisation and blocking effects in non-distributed systems as discussed in this chapter and in chapters 9 and 10 can be regarded as a special case of the problem of capturing software parallelism in distributed systems as considered in chapter 11. Therefore, the performance prediction technique presented there may be applicable to non-distributed information systems also. However, this will not be demonstrated in this thesis. Instead, possible solution paths will be outlined:

- in sec. 9.3.6 where representation of software dynamics in (non-distributed) target platform and existing application systems is discussed, and
- in sec. 11.6 where representation of software dynamics in (non-distributed) projected application systems is considered.

### 8.4 Performance Prediction Overview

\[
\mathcal{X} = \begin{cases} 
\{E_\gamma | \gamma \in \Gamma_S\} \\
\{A_\gamma | \gamma \in \Gamma_S\} \\
\{V_\gamma | \gamma \in \Gamma_S\}
\end{cases} \quad \rightarrow \quad \begin{cases} 
\{Z_\gamma | \gamma \in \Gamma_S\} \\
R_W \\\nV_W \\\nZ_P
\end{cases}
\]

**Performance model** \( M \)

\[
\Omega_W \quad \rightarrow \quad \Omega_W \quad \rightarrow \quad Z_M \quad \rightarrow \quad \text{additional measures}
\]

\[
\tau_M
\]

| Table 8.9: Overview of combined workload and performance model sensitivity analysis. |

An overall picture of the performance predictions framework is shown in fig. 8.9. The design specification is annotated with sets of entry \( \{E_\gamma\} \), activation \( \{A_\gamma\} \), and visit \( \{V_\gamma\} \) matrices for components, from which a design specification visit vector \( V_W \) is derived. Hence this vector contains exactly the same information as the visit matrix \( V_W \) derived in the workload derivation of sec. 8.2. The reason a vector representation is needed here will become apparent in chapter 10.

The visit vector is used for queueing network analysis together with an organisation workload vector \( \Omega_W \) and a service time vector \( \tau_M \). Performance analysis in return yields vectors of design specification operation responsiveness \( R_W \) and service centre sojourn times \( Z_M \). The latter vector is mapped back into a vector of platform service sojourn times \( Z_P = \mu^{-1}(Z_M) \).
Chapter 9

Parameter Capture Support

The basic framework requires a large set of parameters. Although emphasis has been put on selecting a minimum set of parameters that are easy to capture and natural and easy to express, parameter capture is likely to be the most difficult and time consuming aspect of performance engineering during information system development. Making parameter capture easier would therefore strengthen both the availability and cost-efficiency of the approach.

This chapter therefore presents a set of parameter estimation support techniques. They are all extensions to the basic framework of the previous chapters 7 and 8. Techniques are provided for:

- *parameter validation* which compares early parameter estimates with later measurements or analysis results;
- *residence time analysis* analysis which tells the user which parts of the design specification to annotate;
- *software platform modelling* which lets the user annotate the design specification in terms of development concepts on the level of programming language sentences, screens, and database accesses, rather than on the level of CPU operations and bus transfers, and which frees the users from the responsibility of estimating operating system overhead when annotating the design specification;
- *bounds analysis* as an alternative to analysis of means early in the development, and
- *parametric analysis* in case one or a few parameters are infeasible to obtain.

The next chapter 10 will also introduce

- *sensitivity analysis* which tells the user which parameters to estimate with most care.

The common goal of all these techniques is to make performance engineering of information systems cost-efficient and easily available for the non-expert in computer performance evaluation. Of course, performance engineering is a difficult field, and some experience will always
be required. However, the extensions presented in this thesis at least make performance engineering more available to developers and more cost-efficient.

Sec. 9.1 first treats parameter validation, while sec. 9.2 then suggests which parts of a design specification to annotate with most care through residence time analysis. Sec. 9.3 makes parameter capture even easier by introducing the concept of software platforms into the basic framework. However, parameters may still be difficult to obtain in the early phases of development. For such cases, sec. 9.4 introduces bounds analysis as an alternative to estimating average values. Finally, when some parameter is impossible to obtain, sec. 9.5 suggests parametric analysis.

9.1 Parameter Validation

This section outlines techniques for validating performance parameters as soon as they have been estimated. There are two fundamental approaches to this:

- **internal parameter validation** compares performance parameter estimates with one another through a technique of redundant annotations, while

- **external parameter validation** compares performance parameter estimates with measurements made on the final production code.

Sec. 9.1.1 considers internal parameter validation, while sec. 9.1.2 discusses external validation.

9.1.1 Internal parameter validation

Let $\gamma$ be a component that is primitive (i.e. has no subcomponents) at some stage of development, and let $V_\gamma$ be its primitive component visit matrix. As design proceeds, component $\gamma$ is decomposed into a set of subcomponents $\Gamma_\gamma$, each subcomponent possibly having subcomponents on its own. When these subcomponents have been annotated with performance parameters, workload derivation can proceed as in sec. 8.2. This will result in a non-primitive component visit matrix $V_\gamma$ for component $\gamma$. The matrix replaces and makes the old annotation $V_\gamma$ redundant. Since the new visit matrix is based on more up-to-date knowledge of the design, it is also a refinement of the earlier estimate. If the two deviate considerably, then the user should be warned. The differences between $V_\gamma$ and $V_\gamma$ gives a hint whether early workload estimates were too high or too low. In this way, biases in the earlier estimates can be accounted for in other parts of the design specification as well.

Although interparameter validation is likely to provide the information system performance engineer with useful insights into the problem of performance parameter estimation, this technique basically provides no more than an advanced navigation instrument for steering by your own tail. Nevertheless, such a limited navigation instrument is certainly better than none at all — as long as its limitations are constantly kept in the performance engineers mind.
9.1.2 External parameter validation

External parameter validation compares performance parameters against real-world measurements. Hence while the previous sec. 9.1.1 validated earlier estimates against later, more refined ones, this section validates the latest and most refined parameters against measurements of (parts of) the projected application that has already been implemented. As a result, this parameter validation technique works particularly well with the depth-first approach to information system development. Therefore, external parameter validation cannot proceed until the part of the application whose performance parameters are to be validated has in fact been implemented (coded).

Although such an early validation will only be possible for a small part of the projected information system, its value will be great because parameter estimation errors are likely to be similar for all parts of the system. This is because most parts of a system usually will be implemented in the same programming style and will run on the same software platform. The performance parameters will be based on the same assumptions of programming style and software platform performance in all parts of the system. Therefore, biased assumptions in one part of the design specification parameters will indicate biased performance parameters in all parts of the design specification. (This is related to the concept of software platform modelling which will be introduced in sec. 9.3.) Furthermore, the value of an early extra-parameter validation will be enhanced by choosing a performance-critical part of the projected system for such early implementation. (This idea will be further elaborated upon in sec. 9.2 on residence time analysis and chapter 11 on sensitivity analysis.)

Extra-parameter validation proceeds in 3 steps:

1. prepare the early-implemented part of the information system for monitoring;
2. run the early-implemented part under controlled circumstances while monitoring its behaviour, and
3. compare monitoring results with design specification and performance model parameters.

Validation possibilities are almost unlimited, especially when performance model parameters are also taken into account. To limit the scope of this discussion, only design specification parameters are considered. This is in line with the general aim of this thesis to take as much as the hardware level performance modelling machinery as possible for granted. Since the above step 2 is also a well-established branch of computer performance evaluation as pointed out in sec. 4.1, it will not be considered here.

The performance annotations of the basic framework relate directly to basic constructs of the design specification. These constructs will in turn (assuming that packaging will not destroy major structural properties of the design) be directly reflected in the application code. Hence performance annotations can be detected in the application code in the following manner:

an entry count will be reflected by a call to a subprogram in the interface part of a software component — typically invoking a procedure provided by a software module, or a message passing to an object instance — and only by calls to that subprogram;
an activity count will be reflected by a call from one subprogram in the implementation part of a software component to another subprogram in that component implementation — typically internal procedure calls within a software module or messages passed inside an object instance — and only by calls from the former subprogram to the latter, and

a visit count will be reflected by a call from a subprogram to some primitive construct upon which the information system implementation is based — typically programming language sentences, database queries, screen handling, or communication or operating system calls.

By insertion of software counters at each point in the application code where parameters are to be validated, exact values for the corresponding design specification counts can be collected.

These software counters will later provide the required counts with little additional implementation cost and negligible execution overhead, as only occurrences and no execution times are counted. Of course, calls to the monitoring routines should be conditionally compiled so that the software counters will not have to be removed from the application code when the final production version is finished. Thus later monitoring sessions require only a flagged recompilation. This will be valuable in controlling the performance of the software system during maintenance and modification.

The simplicity and straightforwardness makes it easy to incorporate this code instrumentation technique into automated code-generators. In such cases, the instrumentation calls can be automatically inserted into the code at the performance engineer’s request.

### 9.2 Residence Time Analysis

The previous sec. 9.1 presented parameter validation techniques based on the assumption that some parameters had already been estimated. This section makes parameter estimation easier in the first place. It is based on the observation that $\approx 20\%$ of the code of an application is often responsible for $\approx 80\%$ of its execution time and resource consumption [171], due to the log-log distribution which related to Zipf’s law (e.g., [163]) and is common in a range of phenomena of different sciences. In the field of computer science, the rule is recognised in such heuristics as

- 80% of development time is spent on 20% of the system, and
- only 20% of the code is responsible for 80% of the software execution time.

Smith [173] has taken the latter rule as a general principle of software performance engineering, coining it “the residence time analysis rule”:

\[
\leq 20\% \text{ of a program's code accounts for } \geq 80\% \text{ of its resource requirements at the hardware level,}
\]

or in a generalised version:
\leq 20\% of the system functions that are provided will be requested by the system users most (\geq 80\%) of the time.

As already mentioned in sec. 6.5, the thesis will apply a slightly more general \( (1-x)\%\times x\% \) version of this principle, where \( x \) typically in the range 10, \ldots, 20. A smaller \( x \) is a better \( x \), as it provides a more narrow focus on the design specification. This section will focus on the execution time aspect, and the corresponding technique is therefore called residence time analysis.

First, sec. 9.2.1 discusses the need for residence time analysis, before sec. 9.2.2 introduces the concepts of component layers and chains. Sec. 9.2.4 then extends the concept of complexity counts relative to specifications. Sec. 9.2.5 presents residence time analysis per operation, and finally sec. 9.2.6 discusses average residence time analysis.

### 9.2.1 The need for residence time analysis

The residence time analysis technique is useful for two related reasons:

- If the performance critical \( x\% \) of the components in the design specification could be identified early in the development, rather inaccurate parameters would suffice for the remaining \( (1-x)\% \). Indeed, some parts of the design specification might not need annotations at all. The software execution graph technique of sec. 5.3 is based on modelling only the most common execution paths. Conversely, residence time analysis also indicates parts of the design specification for which performance parameters must be estimated with great care.

- In addition, the residence time analysis can be used to determine which parts of the application that will be most critical with respect to performance. Hence, these parts can be identified at an early phase of development. The corresponding components of the design specification are natural candidates for optimisation, in case early performance predictions indicate unacceptable responsivenesses.

### 9.2.2 Component layers and chains

Residence time analysis is based on the concept of component layers, which is defined as follows:

A **component layer** \( \mathcal{L} \) is a set of components \( \gamma \in \Gamma_S \) of design specification \( S \) so that

1. no component in the set is directly or indirectly superior or subordinate to another component in the set, and
2. no other component in the specification can be included in the set without violating the above restriction.
Thus a component layer is a horizontal “cut” through a specification. The first of the above two requirements ensures that this cut is not “folded” anywhere, while the second one ensures that there are no “holes” in it.

![Diagram](image)

Figure 9.1: Higher-level (a) and lowest-level (b) layers in the Customer transaction specification of fig. 8.1.

**Example**

A top-level and a lowest-level layer through the Customer transaction design specification of fig. 8.1 are depicted in fig. 9.1. The higher-level layer comprises the higher-level components \( \gamma_2 \) to \( \gamma_6 \), while the lowest-level layer contains all the primitive components \( \gamma_2, \ldots, \gamma_4 \) and \( \gamma_6, \ldots, \gamma_9 \).

\[
L_1 = \{ \gamma_2, \gamma_3, \gamma_4, \gamma_6 \}, \quad L_2 = \{ \gamma_2, \gamma_3, \gamma_4, \gamma_6, \gamma_7, \gamma_8, \gamma_9 \}
\]

Note how the above two rules are not violated by these layers.

The set \( \{ \gamma_1, \ldots, \gamma_6 \} \) is *not* a layer because it violates the first of the above two rules: Each component \( \gamma_2, \ldots, \gamma_6 \) is a subcomponent of \( \gamma_1 \).

On the other hand, the set \( \{ \gamma_2, \gamma_3, \gamma_4, \gamma_6 \} \) of components is also not a layer because it violates the second of the above two rule: Component \( \gamma_6 \) or components \( \gamma_7, \ldots, \gamma_9 \) could all be included in the set without violating the restriction of rule 1. □

For use in chapter 11 on performance engineering of distributed information systems, an additional concept is needed:

A **component flake** \( F \) is a set of components \( \gamma \in \Gamma_S \) of design specification \( S \) so that

- no component in the set is directly or indirectly superior or subordinate to another component in the set.

Thus a component layer is a horizontal “cut” through a specification, which is not “folded” anywhere. However, a flake may have “holes” in it as opposed to a layer. Hence every layer
is a flake, but not vice versa.

Residence time analysis for a component layer, is based on

1. finding the average sojourn times for all operations of components in the layer, and
2. finding the average numbers of time these operations are used per execution of each top-level operation of the design specification.

The sojourn times of operations in the layer multiplied with the number of times each operation is executed, then give the total amount of time spent executing each operation in the layer for each design specification operation. The total amount of time spent in all the operations in the layer of course corresponds to the responsiveness of the design specification.

Example
In the Customer transaction example, further performance predictions are needed based on refined design specifications and more accurate performance estimates. To determine which part of the design specification to refine first and which parameters to replace with more accurate ones, a residence time analysis is initiated for the lowest-level layer of fig. 3.5b. Since the Withdrawal operation is by far the most common, the initial residence time analysis will be carried out for it according to the above steps. □

9.2.3 Sojourn times of layer operations

The first of the above steps corresponds to finding the sojourn times of all operations provided by components in the given layer. The sojourn time for operation \( o, z_0 \), is the average time spent each time the operation is executed. This matches the definition of sojourn times of performance model service centres and hardware platform services given in sec. 8.3.6.

The workload derivation method of sec. 8.2 provided visit matrices \( V_\gamma \) for all components in the design specification as a side result. Therefore, it provides visit matrices for all components \( \gamma \) in the given layer \( \gamma \in \mathcal{L} \). These matrices represent the average numbers of time each platform service is visited when each of the component's operations are used. The performance analysis of sec. 8.3.7, on the other hand, provided a vector \( Z_P = \mu^{-1}(Z_M) \) of sojourn times for all the services in platform \( P \). This vector represents the average amount of time spent when visiting each platform service. The sojourn time vector for component \( \gamma \) is therefore defined as

\[
Z_\gamma = V_\gamma Z_P. \tag{9.2}
\]

Hence component sojourn times are always based on the results of an earlier performance analysis. The vector \( Z_\gamma \) provides the sojourn times for all the operation provided by component \( \gamma \). Sojourn time vectors must be calculated for all the components \( \gamma \in \mathcal{L} \) of the given component layer \( \mathcal{L} \). The resulting sojourn time vector for layer \( \mathcal{L} \), \( Z_\mathcal{L} \), is formed by stacking these component sojourn time vectors on top of one another in the appropriate order,

\[
Z_\mathcal{L} = \begin{bmatrix} Z_{\gamma_1} \\ \vdots \\ Z_{\gamma_n} \end{bmatrix}, \quad \gamma_i \in \mathcal{L} \tag{9.3}
\]
This concludes the first of the above two steps of residence time analysis.

Example
In the Customer transaction example, the component sojourn time vectors become

\[
\begin{align*}
    z_{\gamma_2} &= V_{\gamma_2}Z_P = (0.96) \\
    z_{\gamma_3} &= V_{\gamma_3}Z_P = (1.24) \\
    z_{\gamma_4} &= V_{\gamma_4}Z_P = (0.32) \\
    z_{\gamma_5} &= V_{\gamma_5}Z_P = (0.26) \\
    z_{\gamma_7} &= V_{\gamma_7}Z_P = (0.29) \\
    z_{\gamma_8} &= V_{\gamma_8}Z_P = (0.065) \\
    z_{\gamma_9} &= V_{\gamma_9}Z_P = (0.14),
\end{align*}
\]

and the sojourn time vector for layer \( \mathcal{L}_2 \) is

\[
Z_{\mathcal{L}_2} = (0.96, 1.24, 0.32, 0.26, 0.29, 0.065, 0.14). \tag{9.4}
\]

(Here, the convention has been adopted that the vector elements appear in order of increasing component indexes.) □

9.2.4 Complexity matrices under specifications

The second of the above two steps corresponds to determining the average numbers of times each operation provided by the given layer is used per execution of each design specification operation. The number of times each operation on each component in a layer is executed per execution of an operation on its specification can now be determined.

This is based on the concept of component chains, which is defined as follows: A component chain from component \( \gamma_1 \) to subcomponent \( \gamma_n \), \( C(\gamma_1, \gamma_n) \), is a sequence of pairs of a component and a direct subcomponent \( (\gamma_1, \gamma_2), \ldots, (\gamma_i, \gamma_{i+1}), \ldots, (\gamma_{n-1}, \gamma_n) \) so that component \( \gamma_{i+1} \) is a subcomponent of \( \gamma_i \), \( i = 1, \ldots, n-1 \).

Thus a component chain is a vertical “cut” through a specification. The above requirement ensures that this cut is “unbroken,” not “doubled” anywhere, and with specified end-points.

Since the component hierarchy is a DAG (directed acyclic graph) rather than a tree, there may be more than one component chain from any component \( \gamma_1 \) to another one \( \gamma_n \). Therefore:

A chain set from component \( \gamma_1 \) to subcomponent \( \gamma_2 \), \( CS(\gamma_1, \gamma_2) \), is the set of all possible component chains \( C(\gamma_1, \gamma_2) \).

Example
Examples of chains in the design specification of fig. 8.1 are \( C(\hat{\gamma}_1, \gamma_2) = ( (\hat{\gamma}_1, \gamma_2) ) \) and \( C(\hat{\gamma}_1, \gamma_9) = ( (\hat{\gamma}_1, \gamma_8), (\gamma_8, \gamma_9) ) \). The list \( ( (\hat{\gamma}_1, \gamma_2), (\gamma_8, \gamma_9) ) \), on the other hand, is not a chain.

Since the component hierarchy of fig. 8.1 is a tree (and not a DAG), there are no alternative component chains. Hence, \( CS(\hat{\gamma}_1, \gamma_2) = \{C(\hat{\gamma}_1, \gamma_2)\} \) and \( CS(\hat{\gamma}_1, \gamma_9) = \{C(\hat{\gamma}_1, \gamma_9)\} \), etc. □
The complexity matrix from component $\gamma_1$ to component $\gamma_n$ can now be defined as

$$C_{\gamma_1}^{\gamma_n} = \sum_{(\gamma_1, \gamma_2, ..., (\gamma_{n-1}, \gamma_n)) \in CS(\gamma_1, \gamma_n)} C_{\gamma_1}^{\gamma_2} \times ... \times C_{\gamma_{n-1}}^{\gamma_n}. \quad (9.5)$$

This matrix represents the number of times each operation of component $\gamma_n$ is executed per execution of each operation of component $\gamma_1$. Hence eq. 9.5 generalises the definition given in sec. 8.2.2 to the case where $\gamma_n$ is any (i.e. not necessarily a direct) subcomponent of $\gamma_1$.

Complexity matrices must be found from all top-level components $\hat{\gamma}_i \in \Gamma^*_S$ to all components $\gamma_j \in \mathcal{L}$ in the given layer.

The complexity matrix from design specification $S$ to layer $\mathcal{L}$ can finally be defined as

$$C_{\mathcal{L}}^{S} = \begin{bmatrix} C_{\hat{\gamma}_1}^{\gamma_1} & \cdots & C_{\hat{\gamma}_1}^{\gamma_m} \\ \vdots & \ddots & \vdots \\ C_{\hat{\gamma}_m}^{\gamma_1} & \cdots & C_{\hat{\gamma}_m}^{\gamma_m} \end{bmatrix}, \quad (9.6)$$

where $\hat{\gamma}_i \in \Gamma^*_S$ and $\gamma_j \in \mathcal{L}$. This concludes the second of the above two steps of residence time analysis.

Example

In the Customer transaction example, the complexity matrices from top-level component $\hat{\gamma}_1$ to the components in layer $\mathcal{L}_2$ become

$$C_{\hat{\gamma}_1}^{\gamma_1} = [1 \ 0 \ 0 \ 0]^T, \quad C_{\hat{\gamma}_1}^{\gamma_3} = [0 \ 1 \ 0 \ 0]^T$$

$$C_{\hat{\gamma}_1}^{\gamma_4} = [0 \ 0 \ 1 \ 0]^T, \quad C_{\hat{\gamma}_1}^{\gamma_5} = [1 \ 1 \ 1 \ 1]^T$$

as in sec. 8.2.2. In addition,

$$C_{\hat{\gamma}_1}^{\hat{\gamma}_1} = C_{\hat{\gamma}_2}^{\hat{\gamma}_2} C_{\hat{\gamma}_3}^{\hat{\gamma}_3} = [0 \ 0 \ 0 \ 1]^T$$

$$C_{\hat{\gamma}_1}^{\hat{\gamma}_6} = C_{\hat{\gamma}_2}^{\hat{\gamma}_2} C_{\hat{\gamma}_3}^{\hat{\gamma}_3} = [0 \ 0 \ 0 \ 0.96]^T$$

$$C_{\hat{\gamma}_1}^{\hat{\gamma}_7} = C_{\hat{\gamma}_2}^{\hat{\gamma}_2} C_{\hat{\gamma}_3}^{\hat{\gamma}_3} = [0 \ 0 \ 0 \ 1]^T$$

The complexity matrix from design specification $S_1$ to layer $\mathcal{L}_2$ therefore becomes

$$C_{\mathcal{L}_2}^{S_1} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0.96 & 1 \end{bmatrix}. \quad (9.11)$$

The bottom row of this matrix corresponds to the execution of the Withdrawal operation of interest. ∎

For use in chapter 11 on performance engineering of distributed information systems, an additional concept is again needed: The complexity matrix from design specification $S$ to
flake $F$ can finally be defined as:

$$
C^S_F = \begin{bmatrix}
C^S_{\gamma_1} & \cdots & C^S_{\gamma_i} & \cdots & C^S_{\gamma_m}
\end{bmatrix}
$$

where $\gamma_i \in \Gamma^S$ and $\gamma_j \in F$.

### 9.2.5 Per operation residence time analysis

Per operation residence time analysis provides the residence times for all the operations provided by a given layer per operation $o \in O_S$ provided by design specification $S$. The residence time of operation $d'$ under another operation $o$ is the average total time spent executing operation $d'$ each time operation $o$ is used. This resembles the definition given in sec. 8.3.6 of residence times of performance model service centres and hardware platform services.

The **residence time for operation $d'$ under top-level operation $o$** is therefore defined as

$$
r^o_{d'} = c^o_{d'} z_{d'},
$$

where the complexity count of operation $o$ under operation $d'$ was defined by the matrix $C^S_{\gamma_j}$, $o \in O_{\gamma_i}$, $d' \in O_{\gamma_j}$. The **operation residence time vector for layer $L$ under top-level operation $o$** is then defined as

$$
R^o_L = (r^o_{d'}), \quad o \in O_S, \quad d' \in O_L.
$$

The **residence time for component $\gamma'$ under top-level operation $o$** is also defined as

$$
r^o_{\gamma'} = \sum_{d' \in O_{\gamma'}} r^o_{d'} = \sum_{d' \in O_{\gamma'}} c^o_{d'} z_{d'},
$$

and the **component residence time matrix for layer $L$** is finally defined as

$$
R^S_L = \begin{bmatrix}
r^o_{\gamma'}
\end{bmatrix}, \quad o \in O_S, \quad \gamma' \in L,
$$

These residence times $r^o_{d'}$ determine the components in the layer that make up $(1-x)\%$ of the design specification execution time for operation $o$. This is typically only a few $(\leq x\%)$ of the components in the layer.

**Example**

In the Customer transaction example, the operation and component residence times are now easily calculated from the layer sojourn time vector $Z_{L_2}$ and design specification complexity matrix $C^S_{L_2}$.

For top-level operation $o_4$ Withdrawal,

- $r^o_{o_5} = 0$, $r^o_{o_6} = 0$, $r^o_{o_7} = 0$, $r^o_{o_8} = 0.14$,
- $r^o_{o_9} = 0.26$, $r^o_{o_{10}} = 0.29$, $r^o_{o_{12}} = 0.060$, $r^o_{o_{13}} = 0.003$. 
9.2. RESIDENCE TIME ANALYSIS

The components in layer \( \mathcal{L}_2 \) provide only one operation each, apart from component \( \gamma_9 \) which provides both operations \( o_{12} \) and \( o_{13} \). Derivation of the corresponding component residence times is trivial,

\[
\begin{align*}
    r_{\gamma_2}^{o_4} &= r_{o_6}^{o_4} = 0, & r_{\gamma_3}^{o_4} &= r_{o_8}^{o_4} = 0, & r_{\gamma_4}^{o_4} &= r_{o_7}^{o_4} = 0, & r_{\gamma_5}^{o_4} &= r_{o_0}^{o_4} = 0.14, \\
    r_{\gamma_7}^{o_4} &= r_{o_{10}}^{o_4} = 0.26, & r_{\gamma_8}^{o_4} &= r_{o_{11}}^{o_4} = 0.29, & r_{\gamma_9}^{o_4} &= r_{o_{12}}^{o_4} + r_{o_{13}}^{o_4} = 0.063.
\end{align*}
\]

The component residence time vector for layer \( \mathcal{L}_2 \) under operation \( o_4 \) therefore is

\[
R_{\mathcal{L}_2}^{o_4} = (0, 0, 0, 0.14, 0.26, 0.29, 0.063).
\]

(Again under the convention that vector elements appear in order of increasing component indexes.) According to this, components \( \gamma_7 \) and \( \gamma_8 \) make up 73\% of the overall response time of operation \( o_4 \) Withdrawal. In the Customer transaction example, it is therefore decided to focus refinement and parameter capture effort on these two components. At the same time, components \( \gamma_7 \) and \( \gamma_8 \) correspond to 29\% of the components in the design specification. Hence according to the \((1-x)\%\-x\%\) principle, \( x \approx 27\ldots 29 \) in this case. A larger design specification would have been likely to produce a smaller and hence better \( x \).

From considerations parallel to the above, the complexity matrix for layer \( \mathcal{L}_2 \) under design specification \( S_1 \) can be derived as

\[
R_{\mathcal{L}_2}^{S_1} = \begin{bmatrix}
0.96 & 0 & 0 & 0.26 & 0 & 0 & 0 \\
0 & 1.24 & 0 & 0.26 & 0 & 0 & 0 \\
0 & 0 & 0.32 & 0.26 & 0 & 0 & 0 \\
0 & 0 & 0 & 0.26 & 0.29 & 0.063 & 0.14 \\
\end{bmatrix}.
\]

9.2.6 Average residence time analysis

However, the residence times of the previous sec. 9.2.5 only apply to a single operation of the design specification at a time. Since the projected application is likely to provide many functions, the results may therefore be difficult to comprehend. This section therefore introduces average residence time analysis as an alternative.

Average residence time analysis considers the residence times for components and their operations in the component layer per average execution of some operation of design specification \( S \). Since residence time analysis should take as many aspects of the application into consideration as possible, this is believed to be more useful than the per operation analysis.

The analysis relies on the concept of operation frequency vectors for design specifications. The rate for top-level operation \( o \) is first defined as

- \( \rho_o = \frac{\rho_o}{\theta_o + \rho_o} \) iff operation \( o \) is terminal;
- \( \rho_o = \frac{\rho_o}{\theta_o} \) iff operation \( o \) is batch, and
- \( \rho_o = \lambda_o = x_o \) iff operation \( o \) is transaction,

where the \( \lambda_o \), \( n_o \), and \( \theta_o \)'s all were all defined in sec. 8.3.6. These performance analysis inputs represent the arrival rate of transaction operations, the numbers of terminal and batch
operations, and the think times for terminal operations, respectively. Outputs \( r_o \) and \( x_o \) from the earlier performance analysis of sec. 8.3.7 represent response times for all operations and throughputs for terminal and batch operations, respectively. Since residence time analysis must be based on the results of some earlier performance analysis, it may also be assumed that responsiveness measures \( r_o \) and \( x_o \) are available as in the above equations.

Since the set of operations provided by the application corresponds to the set of operations provided by its top-level components, the \textbf{operation frequency for top-level operation} \( o \) can now be derived as

\[
f_o = \frac{\rho_o}{\sum_{o' \in O_S} \rho_{o'}},
\]

and the \textbf{operation frequency vector for design specification} \( S \) accordingly as

\[
F_S = \langle f_{o_1}, \ldots, f_{o_n} \rangle, \quad o_i \in O_S.
\]

This vector is normalised by definition, since

\[
\sum_{o \in O_S} f_o = \sum_{o \in O_S} f_o
\]

\[
= \sum_{o \in O_S} \frac{\rho_o}{\sum_{o' \in O_S} \rho_{o'}}
\]

\[
= \frac{\sum_{o \in O_S} \rho_o}{\sum_{o' \in O_S} \rho_{o'}}
\]

\[
= 1.
\]

\textbf{Example}

In the \textbf{Customer transaction} example, all top-level operations have transaction type workload intensities as specified in sec. 8.3.2. The rates of these operations therefore become

\[
\rho_{o_1} = 0.162, \quad \rho_{o_2} = 0.054, \quad \rho_{o_3} = 2.16, \quad \rho_{o_4} = 27.0,
\]

and the operation frequencies are

\[
f_{o_1} = 5.51 \cdot 10^{-3}, \quad f_{o_2} = 1.84 \cdot 10^{-3}, \quad f_{o_3} = 73.5 \cdot 10^{-3}, \quad f_{o_4} = 0.919.
\]

since \( 0.162 + 0.054 + 2.16 + 27.0 = 29.376 \). Of course, \( 5.51 \cdot 10^{-3} + 1.84 \cdot 10^{-3} + 73.5 \cdot 10^{-3} + 0.919 = 1 \) as is expected.

The operation frequency vector of the \textbf{Customer transaction} specification becomes

\[
F_{S_1} = \langle 5.51 \cdot 10^{-3}, \ 1.84 \cdot 10^{-3}, \ 73.5 \cdot 10^{-3}, \ 0.919 \rangle.
\]

The operation dependent residence times of the previous sec. 9.2.5 may now be averaged using this operation frequency vector.

\[
R_S^D = F_S R_L^D,
\]

where the matrix \( R_L^D \) was derived in the previous sec. 9.2.5.
9.2. RESIDENCE TIME ANALYSIS

Example
In the Customer transaction example,
\[ R^{S_1} = F_S, R_{\mathcal{L}}^{S} = \langle 5.29 \cdot 10^{-3}, 2.28 \cdot 10^{-3}, 23.5 \cdot 10^{-3}, 0.26, 0.27, 0.058, 0.13 \rangle. \] (9.29)
According to this, the two components \( \gamma_7 \) and \( \gamma_8 \) make up 71% of the overall response time on average. These two components still correspond to 29% of the components in the design specification. Hence according to the \((1 - x)\% - x\% \) principle, \( x = 29 \). Again, a larger design specification would have been likely to produce a smaller and hence better \( x \).

The results are consistent with those of the previous sec. 9.2.5 for the Withdrawal operation, which confirms the initial assumption that this operation was the most important to overall performance due to its high workload intensity. \( \square \)

For the purpose of chapter 10, an additional operation frequency vector for component \( \gamma \) is also needed. It is defined as
\[ F_\gamma = (F_S)^T C_{\mathcal{S}}^{\mathcal{F}_\gamma}, \] (9.30)
where \( \mathcal{F}_\gamma = \{\gamma\} \) is the flake containing only component \( \gamma \), since the complexity matrix \( C_{\mathcal{S}}^{\mathcal{F}_\gamma} \) represents the average number of uses for each operation provided by this component for each component provided by design specification \( S \).

9.2.7 Use of residence time analysis

From the residence times \( r_\gamma^o \) of each component \( \gamma \in \mathcal{L} \) of component \( \gamma \) in layer \( \mathcal{L} \), a residence ratio for component \( \gamma \) under top-level operation \( o \), \( \theta_\gamma^o \), is defined as
\[ \theta_\gamma^o = \frac{r_\gamma^o}{r_o}. \] (9.31)
Sorting the resulting set
\[ \{\theta_\gamma^o | \gamma \in \mathcal{L}\} \] (9.32)
of residence ratios for all components \( \gamma \in \mathcal{L} \) the residence ratio list for layer \( \mathcal{L} \) under top-level operation \( o \), \( \mathcal{R}_\mathcal{L}^o \), is defined as
\[ \mathcal{R}_\mathcal{L}^o = (\theta_1^o, \ldots, \theta_{\gamma_1}^o, \ldots, \theta_{\gamma_{|\mathcal{L}|}}^o), \] (9.33)
where \( \theta_{\gamma_i}^o \leq \theta_{\gamma_{i+1}}^o \). From the residence ratio list, a residence quantile list for layer \( \mathcal{L} \) under top-level operation \( o \), \( \mathcal{Q}^o \), is defined as
\[ \mathcal{Q}^o = \left( (\theta_1^o, q_1^o), \ldots, (\theta_i^o, q_i^o), \ldots, (\theta_{|\mathcal{L}|}^o, q_{|\mathcal{L}|}) \right), \] (9.34)
where
\[ \theta_i^o = \sum_{k=1}^{i} \theta_{\gamma_k}^o, \quad q_i = \frac{i}{P(\mathcal{Q}^o)}. \] (9.35)
In this list, the \( \theta_i^o \)'s describe the total residence ratio of the \( i \) components \( \gamma \in \mathcal{L} \) with longest residence times, while \( q_i^o \) is the residence quantile of layer \( \mathcal{L} \) that these components constitute. (Note the distinction between the residence ratio \( \theta_\gamma^o \) of component \( \gamma_i \) and the residence ratio \( \theta_i^o \) of the first \( i \) components in residence ratio list \( \mathcal{R}_\mathcal{L}^o \).) In other words, the \( i \)'th element of residence quantile list \( \mathcal{Q}^o \) states that
100$q_i^o\%$ of the components of layer $\mathcal{L}$ make up $100\beta_i^o\%$ of the total response time.

The purpose of residence time analysis is to limit the designers attention to a set of $i$ components which is the most crucial ones to performance. Therefore, a number $i$ must be determined so that the residence ratio $\beta_i^o$ is as large as possible compared to the residence quantile $q_i^o$.

**Non-distributed software platform $\mathcal{F}'$:**

$\mathcal{F}' = \text{Software platform}$

$O_{\mathcal{F}'} = \{o_1, o_2, o_3, o_4, o_5, o_6, o_7\}$

- $o_1$ = Execute statement
- $o_2$ = Select tuple
- $o_3$ = Select tuples
- $o_4$ = Update
- $o_5$ = Insert/delete
- $o_6$ = Read screen
- $o_7$ = Write screen

<table>
<thead>
<tr>
<th>Table 9.2: Software platform for the Customer transaction model of fig. 3.5.</th>
</tr>
</thead>
</table>

![Software platform](image)

Figure 9.3: Software platform (a) and composite workload model (b) for the bank example.

### 9.3 Software Platform Modelling

The two last secs. 9.1 and 9.2 have introduced techniques to support validation and estimation of performance annotations. This section makes parameter estimation even easier in the first place by extending the basic framework with the concept of *software platforms*.

First, sec. 9.3.1 discusses the need for software platform modelling, before sec. 9.3.2 introduces *software platform models*. Sec. 9.3.3 describes the software platform model *analysis* method,
9.3. SOFTWARE PLATFORM MODELLING

while sec. 9.3.4 introduces the platform estimate assumption that software platform modelling technique depends on.

9.3.1 The need for software platform modelling

So far, design specifications have been annotated with demand estimates in terms of hardware-level services such as CPU’s, disks, and communication channels provided by the hardware platform. This creates two main problems:

- Today there is a tendency to build software by putting together existing components [178, 189]. Common examples of such software are programming languages database management systems (DBMS’s), screen handlers, as well as already existing applications. This is the target platform software of the projected application. This means that the application is designed and implemented in terms of software platform functions rather than in terms of resources at the hardware level. This creates difficulties in two ways:

  - Application developers are not used to think in terms of computer hardware, and it is difficult for them to annotate design specifications in terms of such low-level concepts.

  - Assessing the performance of the software platform, e.g. an optimised relational database management system, is a highly specialised task.

- Every application running on a computer increases the operating system overhead on that computer. Common examples of such overhead is the additional work induced by virtual memory managers, dispatchers, and interrupt handlers. This is the operating system software of the computerised information system. This means that the visit counts of a design specification must include estimates of the induced operating system workload for the derived workload model to become correct. Again, this creates difficulties in two ways:

  - Application developers are not used to take the behaviour of the operating system software into consideration when designing applications.

  - The operating system overhead of an application is not fixed for that application. Instead, it is a function of the total workload on the computer. This workload is produced both by the projected application and by other, already existing applications.

All the above difficulties are in conflict with the aim of this thesis of making software performance engineering easily available to information systems developers.

9.3.2 The software platform model

This section introduces software platform modelling to avoid the above problems.
A software platform $P'$ is a set $O_{P'}$ of operations $o \in O_{P'}$. These operations represent functions provided by the target platform software upon which the projected and existing applications will be and are implemented. Hence software platform $P'$ replaces the resource platform $P$ of sec. 8.1.3. The only difference between the two is that a software platform is a set of operations representing target platform software functions, while a hardware platform is a set of services representing computer resources.

Example
A software platform for the bank example is shown in table 9.2. □

A software platform model $W_{P'}$ is a representation of the target platform and operating system software in terms of workload modules $W_\zeta \in W_{P'}$. These workload modules are the same as those introduced in sec. 8.3, with one minor relaxation. Each major software component $\zeta$ of the target platform and operating system is represented as such a workload module $W_\zeta$. (As mentioned in sec. 4.2, establishment of workload modules has been the topic of much research. Also, it is studied in a related thesis. Therefore, it will not be elaborated on here.) Furthermore, the software platform model organises these workload modules into a DAG, so that higher-level modules use operations provided by lower-level ones. Primitive workload modules in the software platform model use services provided by the hardware platform as before. Hence the relaxation of the earlier workload module definition is seen: Non-primitive workload modules now use operations provided by other workload modules, while primitive ones still visit hardware platform services.

The set of operations provided by workload module $W_\zeta$ is $O_\zeta$, and the set of lower-level operations used by non-primitive workload module $W_\zeta$ is $O_\zeta'$.

The set of top-level workload modules $W_\zeta'$ of workload model $W_{P'}$ is $W_{P'}^\ast$, $W_\zeta' \in W_{P'}^\ast$. The operations $o' \in O_\zeta'$ provided by these top-level workload modules $W_\zeta'$ must correspond to the set of operations provided by software platform $P'$, $o' \in O_{P'}$. Of course, an additional software platform mapping, $\mu'$ can be provided at some notational inconvenience. This is however not considered here.

Example
The software platform model for the Customer transaction development example is shown in fig. 9.3a. The operations provided by the top-level workload modules COBOL, SQL, and Screen handler of this model corresponds to the platform operations of table 9.2. The Customer transaction specification can now be annotated with visit counts in terms of these top-level operations rather than in terms of hardware level services. □

As in the basic framework, an additional top-level workload module $W_S$ is derived from design specification $S$ to represent the projected application according to sec. 8.2. In case the projected application will run alongside other already existing applications $A$, additional top-level workload modules $W_A$ may be established as in sec. 8.3.3 Some or all of these may be based on the same or partially the same software platform as the projected one.

The resulting composite workload model therefore consists of both application, target plat-
form, and operating system workload modules $W_\zeta$. Hence the notation $W_\zeta$ is also used to denote projected and existing application workload modules $W_S$ and $W_A$ for convenience.

**Example**

The composite workload model for the bank example is shown in fig. 9.3b. The projected Customer transaction system, will share the same software platform as the already existing applications for Accounting, Employee payroll, and System administration. However, chapter 11 will regard the Accounting system as a projected application with yet another target platform. □

\[
\begin{align*}
V^*_\zeta &= \overline{V_{s1}}, \\
\vdots &\qquad \zeta \in W^*_P \\
V^*_{sn} &\qquad (9.36) \\
V_\zeta &= C_\zeta V^*_\zeta \qquad (9.37)
\end{align*}
\]

Table 9.4: Composite workload model analysis technique.

### 9.3.3 Composite workload model analysis

After analysing the composite workload model, performance prediction can proceed as in the basic framework. Composite workload model analysis proceeds through bottom-up collapsing of the composite workload model DAG, starting with the primitive workload modules. Hence composite workload model analysis is very similar to the workload derivation method for design specifications described in sec. 8.2, and is also heavily inspired by the sp approach of sec. 4.8.2.

As defined in sec. 8.3.2, visit matrices for primitive workload modules $W_\zeta$ describe how the workload module visits services provided by the hardware platform. The visit matrix for primitive workload module $W_\zeta$, $V_\zeta$, is a matrix $V_\zeta = [v^\sigma_o]$, $v^\sigma_o \geq 0$, $o \in O_\zeta$, $\sigma \in \Sigma_P$. Element $v^\sigma_o$ of this matrix represents the average number of times service $\sigma$ is visited per execution of operation $o$ on workload module $W_\zeta$.

For non-primitive workload modules, however, this definition is somewhat modified since complexity matrices for non-primitive workload modules $W_\zeta$ describe how the workload module uses operations provided lower-level workload modules. The complexity matrix for non-primitive workload module $W_\zeta$, $C_\zeta$, is a matrix $C_\zeta = [c^o_{\sigma'}]$, $c^o_{\sigma'} \geq 0$, $o \in O_\zeta$, $\sigma' \in O^*_\zeta$. Element $c^o_{\sigma'}$ of this matrix represents the average number of times operation $o$ is used per execution of operation $o$ on workload module $W_\zeta$.

When all the lower-level workload modules of module $\zeta$ have been collapsed, an internal visit matrix $V^*_\zeta$ is derived by stacking the visit matrices of its directly subordinate workload modules on top of one another as in eq. 9.36. The visit matrix $V_\zeta$ for the workload module
is then calculated by eq. 9.37, preparing for collapsing at the next higher level of workload modules. Since the workload model DAG is cycle-free, this algorithm always terminates successfully when the top-level workload module has been collapsed. The resulting top-level visit matrices therefore define the visit counts needed for performance model analysis.

As in sec. 8.3.4, an overall visit matrix for workload model \( W \) is now formed from the collapsed top-level workload module visit matrices for projected and existing applications. In addition, a \textit{visit matrix for software platform} \( P' \) can be derived for the software platform model. This matrix \( V_{P'} \) will become important in sec. 9.3.4.2.

9.3.4 Parameter estimation support

9.3.4.1 Parameter validation

The parameter validation techniques presented in sec. 9.1 does not interfere with software platform modelling. These validation techniques are applicable whether or not software platform modelling is used.

9.3.4.2 Residence time analysis

Apart from the additional software platform analysis prior to performance model analysis, software platform modelling does not interfere with the basic framework. However, to combine the residence time analysis technique of sec. 9.2 (and also the sensitivity analysis technique that will be introduced in chapter 10) with software platform modelling, a new concept is needed. The reason is that residence time analysis depend on the sojourn times for components. These sojourn times were again defined in terms of hardware level services. Since software platform modelling replaces the concept of hardware level services with software platform operations, a new concept of \textit{software platform sojourn times} must be introduced.

The \textbf{sojourn time vector for software platform} \( P' \), \( Z_{P'} \), is defined as

\[
Z_{P'} = V_{P'} Z_P, \tag{9.38}
\]

where \( Z_P \) is the sojourn time vector for services of hardware platform \( P \) as defined in sec. 8.3.6. Hence software platform sojourn times depend on the software platform visit matrix derived in the previous sec. 9.3.3 and the service centre sojourn time vector of sec. 8.3.6. Residence time analysis (or sensitivity analysis) now proceeds with software platform sojourn times \( z_o, o \in O_P \) and \( Z_P \) replacing the hardware platform sojourn times of sec. 8.3.6.

9.3.5 Database optimisation

The critical point in establishing a software platform model, is representing the optimised database management system (DBMS) as a workload module. Sevcik [161] has presented an hierarchical approach to database performance prediction. This work also includes a sequence of database workload descriptions. Other work in the same field include Casas Raposo [56] and Hyslop [105]. Also, [40, 39] discusses alternative parameter capture strategies for an
9.3. SOFTWARE PLATFORM MODELLING

SQL-type DBMS in the context of a practical case study. Wieland [192] has established and validated a workload module specification for simple queries on the INGRES relational database system.

In general, the problem is that contemporary database optimisers are complex and have a large number of input parameters which can be varied, partially on a try-and-see basis. In addition, the optimisation algorithm is usually not public. Hence, although it is possible to conceive the inputs to the optimiser at an early stage of development, the performance of its outputs are still close to unknown.

At the same time, the optimised database is one of the information system components that impacts its overall performance the most. In practice, the following possibilities exist:

- **consult the database expert;**
  Development projects large enough to make software performance engineering mandatory, will usually have an expert specifically assigned to the task of database optimisation. Based on previous experience, the expert will be able to give bounds (to be introduced in the next sec. 9.4) or average estimates of the likely performance of the optimised system.

- **consult vendor;**
  Even if the project has no dedicated and experienced database expert available, bounds or average estimates can sometimes still be obtained from the database vendors.

- **query the optimiser itself;**
  If development has reached the point where a data definition schema is available, the optimiser can be run. In addition to producing an optimised database, most professional optimisers will also provide statistics about the optimisation results. These results can be used to establish a workload module representation of the optimised database.

- **assume direct access paths;**
  For interactive transactions, database queries which are not realised in terms of direct access paths are to be regarded as performance bugs for all but very small relations. Direct access can be implemented through indexing or hashing, for which performance is predictable. If later measurements of the actual system indicate deviations from the predicted “direct-access” performance, the database must be re-optimised until the information system’s performance is consistent with predictions.

- **use experience data;**
  Large companies continuously monitor the performance of their hardware and software systems. Often, performance logs are kept for capacity planning purposes. Using previous performance measurements of “similar,” optimised databases, a workload module can be established for the projected one too.

- **use analytic models;**
  Wieland [192] is an example of an attempt to model the optimised performance of database queries from knowledge of optimisation alternatives. Results indicate that even simple models can be accurate (e.g. within 10% for many query types, and often exact) when considering simple queries, although several odd results were measured. Furthermore, most interactive queries in an information system are of this simple type.
Although much work remain on collecting experience with and improving and evaluating these alternatives, it is obvious that numerous options exist, and that the database performance problem is not out of hand. It is likely that all these techniques will be useful in diverse development projects, either alone or in combination. The issue remains a topic for further research.

9.3.6 Target platform synchronisation and blocking

Sec. 8.3.7 pointed out that synchronisation and blocking effects may have important impact on the performance of target platform software systems, e.g. locking in a database. Also, synchronisation and blocking in application software is an important problem. For such cases, the static approach to target platform modelling taken in this section is insufficient. Instead, dynamic models are needed. As was pointed out in sec. 4.5, Petri-net models (e.g. [3]) are suited for representing software dynamics. The possibility therefore exists of extending the technique of this section with dynamic workload modules, specified in terms of a generalised stochastic Petri-net rather than in terms of a (static) visit or complexity matrix. The operations provided by such a workload modules would be represented as token colours in the Petri-net, while operations or services used by the model would correspond to stochastically timed transitions. The resulting workload model would include both static and dynamic workload modules, and can be analysed by adaptation of the methods for performance prediction of distributed information systems presented in chapter 11. Also, specialised approximate algorithms could be used (e.g. [156]).

The main problem with dynamic specification of existing (target platform or application) software systems is that the system interiors that must be represented in the Petri-net model may not be known. Also, the Petri-net will require a refined set of performance parameters that might be infeasible to capture with contemporary measurement tools.

9.3.7 Use of target platform modelling

Software platform modelling resolves the problems described in sec. 9.3.1:

- The top-level workload modules in the software platform correspond to the set of software systems that the projected application will be implemented upon, typically a programming language, a database management system (DBMS), and a screen handler. Each of the top-level software platform modules provide operations that the application uses. Thus instead of annotating the primitive components of the design specification in terms of hardware-level services, they are supplied with visit count estimates in terms of software platform operations. This resolves the first of the above two main problems.

- The lower-level workload modules in the software platform provide operations that the higher-level use, but they are not used by the application itself. These modules correspond to the operating system software of the computer, such as the virtual memory manager, the dispatcher, and the interrupt handler. Thus instead of having to include estimates of the induced operating system workload in the design specification annotations, it is automatically calculated during workload model analysis. This resolves the
second of the above two main problems.

9.4 Bounds Analysis

Sections 9.2 to 9.3 have presented two techniques to support and simplify parameter estimation. Nevertheless, obtaining accurate performance estimates may be difficult — and sometimes not even worthwhile — in the early phases of design. For such cases, Smith [173] advocates bounds analysis.

First, sec. 9.4.1 discusses the need for bounds analysis, before sec. 9.4.2 extends the performance parameters of the basic framework for bounds analysis. Sec. 9.4.3 accordingly extends the workload derivation method, and sec. 9.4.4 discusses workload and performance models for bounds analysis.

9.4.1 The need for bounds analysis

The purpose of bounds analysis is to enable application developers to distinguish between the following three situations early in the development:

1. The current design (of an application or of a part of it) is likely to have clearly acceptable performance.
   In this case, no immediate actions are needed. Nevertheless, it is vital that subsequent performance analyses are undertaken. It has been noted that performance problems have a tendency to creep into seemingly trouble-free design as more functions are added to the requirements [196].

2. The current design (of an application or of a part of it) is clearly unlikely to have acceptable performance.
   In this case, precautions that may be taken are:

   (a) Reorganising the organisation workload.
       The workload module derived from the design specification (possibly together with a platform model) points out the bottleneck resource and thus suggests which of the operations on the application that are most expensive in terms of computing resources.

   (b) Redesigning the projected application or part of it in question.
       In this case, residence time analysis or sensitivity analysis of the design specification or component in question provides hints of where the design can be optimised.

   (c) Upgrading the hardware on which the projected application will run.
       As in the first case, the workload module derived from the design specification (possibly together with a platform model) points out the bottleneck resource and thus suggests which part of the computer hardware to upgrade.
3. It is not clear whether the current design (of an application or of a part of it) is likely to have acceptable performance or not.

In this case, subsequent performance analyses at frequent intervals are mandatory.

**Example**
In the Customer transaction example, it is difficult to estimate the withdrawal success rate at an early stage of development, i.e., the 96% rate that has been assumed so far is a very coarse estimate. However, it is clear that the success rate is above 90% and must be below 100% by definition. Therefore, a bounds analysis study is initiated, rather than an analysis of means at this early design state. All parameters other than \( a_{v_{0,0}}, a_{v_{0,1}} \) and \( a_{v_{0,0},v_{0,2}} \) of table 8.2 are kept as averages. □

### 9.4.2 Performance parameters

Bounds analysis is based on annotating the design specification with minimum and maximum bounds in the early phases of design, instead of average estimates. This is preferable in the early design phases, because

1. minimum and maximum parameters are easier to obtain;
2. the most interesting performance question is whether performance is clearly acceptable or clearly unacceptable, and
3. the obtainable average estimates are probably to inaccurate to be trustworthy.

In sec. 8.1 design specifications were annotated with entry, activity, and visit matrices to support performance predictions. The same types of annotations must be provided for bounds analysis. However, a choice is now available for each parameter, in that either

- an average estimate, or
- one minimum and one maximum bound

... can be provided. From this point on, analysis proceeds in parallel to the basic framework, along one minimum and one maximum path. These two paths provide minimum and maximum bounds on the responsiveness of the projected application. Of course, the minimum and maximum bounds will be equal when an average estimate has been given for a parameter.

**Minimum and maximum parameters** provided must follow the following inequalities:

- for entry, activity, and visit counts, as well as for numbers of terminal users and batch jobs, arrival rates and service centre service times, the minimum bound \( x^- \) must be less than or equal the maximum bound \( x^+ \),

\[
x^- \leq x^+, \tag{9.39}
\]
because these parameters all affect performance measures \textit{negatively}, i.e. they increase response times, residence times, sojourn times, utilisations, and queue lengths, while decreasing throughput.

- for terminal user think times, the minimum bound \( x^- \) must be greater than or equal the maximum bound \( x^+ \),

\[
x^- \geq x^+,
\]

(9.40)

because think times affect performance measures \textit{positively}, i.e. decrease response times, residence times, sojourn times, utilisations, and queue lengths, while increasing throughput.

\textbf{Entry matrices} accordingly still describe how the external operations of a component initiate internal operations. The \textbf{minimum} and \textbf{maximum entry matrices for component} \( \gamma \), \( E^-_\gamma \) and \( E^+_\gamma \) are matrices \( E^-_\gamma = \left[ e^-_{o,d} \right] \) and \( E^+_\gamma = \left[ e^+_{o,d} \right] \), \( e^-_{o,d}, e^+_{o,d} \geq 0 \). Elements \( e^-_{o,d} \leq e^+_{o,d} \) of these matrices bound the number of times operation \( d' \) on some subcomponent of \( \gamma \) is initiated per execution of operation \( o \) on component \( \gamma \). Minimum and maximum entry matrices must be provided for every component in the design specification for bounds analysis to be applicable.
Example

The entry matrices in the Customer transaction example are kept as averages. Thus,

\[
E_{\gamma_1}^- = E_{\gamma_1}^+ = E_{\gamma_1}^0 = \begin{bmatrix}
1 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 1
\end{bmatrix}
\]  \hspace{1cm} (9.41)

and

\[
E_{\gamma_2}^- = E_{\gamma_2}^+ = E_{\gamma_2}^0 = [1 \ 0 \ 0].
\]  \hspace{1cm} (9.42)

Activity matrices still describe how the internal operations of a component activate one another. The minimum and maximum activity matrices for component \( \gamma \), \( A^-_\gamma \) and \( A^+_\gamma \), are matrices \( A^-_\gamma = \begin{bmatrix} a^-_{\sigma, \sigma'} \end{bmatrix} \) and \( A^+_\gamma = \begin{bmatrix} a^+_{\sigma, \sigma'} \end{bmatrix} \), \( a^-_{\sigma, \sigma'}, a^+_{\sigma, \sigma'} \geq 0 \). Elements \( a^-_{\sigma, \sigma'}, a^+_{\sigma, \sigma'} \) of these matrices bound the number of times operation \( \sigma'' \) on some subcomponent of \( \gamma \) is executed next after operation \( \sigma' \) on some (possibly the same) subcomponent of component \( \gamma \). Minimum and maximum activity matrices must also be provided for every component in the design specification for bounds analysis to be applicable.

Example

The activity matrix of top-level component \( \gamma_1 \) remains unchanged,

\[
A^-_{\gamma_1} = A^+_{\gamma_1} = A_{\gamma_1}^0 = \begin{bmatrix}
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0
\end{bmatrix}
\]  \hspace{1cm} (9.43)

as in sec. 8.1.2.

The bounds matrices for the Make withdrawal subcomponent \( \\gamma_5 \) become

\[
A^-_{\gamma_5} = \begin{bmatrix} 0.9 & 0.0 & 0 \\
0 & 0 & 1 \\
0 & 0 & 0
\end{bmatrix}
\]  \hspace{1cm} (9.44)

and

\[
A^+_{\gamma_5} = \begin{bmatrix} 1.0 & 0.1 & 0 \\
0 & 0 & 1 \\
0 & 0 & 0
\end{bmatrix}
\]  \hspace{1cm} (9.45)

since the success rate is now bounded between 0.9 and 1.0 and the failure rate accordingly between 0.0 and 0.1. \( \square \)

Visit matrices for primitive components also still describe how a primitive component visits hardware platform services. The minimum and maximum visit matrices for primitive component \( \gamma \), \( V^-_\gamma \) and \( V^+_\gamma \), are matrices \( V^-_\gamma = \begin{bmatrix} v^-_\sigma \end{bmatrix} \) and \( V^+_\gamma = \begin{bmatrix} v^+_\sigma \end{bmatrix} \), \( v^-_\sigma, v^+_\sigma \geq 0 \).
Elements \( v^\sigma_\sigma^- \leq v^\sigma_\sigma^+ \) of these matrices bound the numbers of time service \( \sigma \) is visited per execution of operation \( \sigma \) on primitive component \( \gamma \). Minimum and maximum visit matrices only have to be provided for the primitive components of the design specification for bounds analysis to be applicable.

**Example**
Visit matrices for primitive components also retain their representation as average estimates, e.g.

\[
V^-_{\gamma_\alpha} = V^+_{\gamma_\alpha} = V_{\gamma_\alpha} = \begin{bmatrix}
10000 & 0.5 & 20
\end{bmatrix}.
\] (9.46)

\( \square \)

### 9.4.3 Workload derivation

**Internal visit matrices for components** still describe how components visit hardware platform services. The minimum and maximum internal visit matrices for component \( \gamma \), \( V^-_{\gamma} \) and \( V^+_{\gamma} \), are matrices \( V^-_{\gamma} = [v^\sigma_\sigma^-] \) and \( V^+_{\gamma} = [v^\sigma_\sigma^+] \). Elements \( v^\sigma_\sigma^- \leq v^\sigma_\sigma^+ \) of these matrices bound the numbers of time service \( \sigma \) is visited per execution of operation \( \sigma \) on some subcomponent of \( \gamma \).

\[
V^-_{\gamma} = \begin{bmatrix} V^-_{\gamma_1} & \vdots & \vdots \\ V^-_{\gamma_n} & \vdots & \vdots \\ \vdots & \ddots & \vdots \\ \vdots & \vdots & \vdots \end{bmatrix}, \quad V^+_{\gamma} = \begin{bmatrix} V^+_{\gamma_1} & \vdots & \vdots \\ \vdots & \ddots & \vdots \\ \vdots & \vdots & \vdots \end{bmatrix}, \quad \gamma_i \in \Gamma^S \tag{9.47}
\]

In the two next paragraphs, the inequality \( v^\sigma_\sigma^- \leq v^\sigma_\sigma^+ \) will be demonstrated by induction over the number of component levels in the design specification. Obviously, the inequality will hold for the first non-primitive level of components since the inequality must hold (by definition of sec. 9.4.2) for each pair of minimum and maximum matrices \( V^-_{\gamma_1} \) and \( V^+_{\gamma_1} \), \ldots, \( V^-_{\gamma_n} \) and \( V^+_{\gamma_n} \) for primitive components \( \gamma_1, \ldots, \gamma_n \). Let the induction hypothesis be that the inequality holds for any component level.

**Example**
The minimum and maximum internal visit matrices for component \( \gamma_5 \) still represent average estimates,

\[
V^-_{\gamma_5} = V^+_{\gamma_5} = V_{\gamma_5} = \begin{bmatrix}
15000 & 2.0 & 10 \\
10000 & 2.0 & 0 \\
5000 & 3.0 & 10
\end{bmatrix}.
\] (9.48)

\( \square \)

**Complexity matrices** accordingly still describe the numbers of uses of internal operations that correspond to one use of each external operation of a component. The minimum and maximum complexity matrices for component \( \gamma \), \( C^-_{\gamma} \) and \( C^+_{\gamma} \), are matrices \( C^-_{\gamma} = [c^-_{\sigma, \sigma}] \) and
\( C_{\gamma}^+ = [\gamma_{o,d}^+] \). Elements \( e_{o,d}^- \leq e_{o,d}^+ \) of these matrices bound the number of times operation \( d \) on some subcomponent of \( \gamma \) is used per execution of operation \( o \) on component \( \gamma \).

The minimum and maximum complexity matrices for component \( \gamma \) are derived from its entry and activity matrices and matrices as follows

\[
C_{\gamma}^- = E_{\gamma}^- (I - A_{\gamma}^-)^{-1},
\]
\[
C_{\gamma}^+ = E_{\gamma}^+ (I - A_{\gamma}^+)^{-1}.
\]

The inequality \( e_{o,d}^- \leq e_{o,d}^+ \) is demonstrated as follows: Let \( X_{i,\gamma} \) be the \( i \)th column vector and \( \overline{X}_{i,\gamma} \) be the \( i \)th row vector of any matrix \( X_{\gamma} \), and let

\[
Q_{\gamma}^- = (I - A_{\gamma}^-)^{-1},
\]
\[
Q_{\gamma}^+ = (I - A_{\gamma}^+)^{-1}.
\]

Note also that matrices \( Q_{\gamma}^- \) and \( Q_{\gamma}^+ \) can be approximated by the infinite sums

\[
Q_{\gamma}^- = (I - A_{\gamma}^-)^{-1} = \sum_{i=0}^{\infty} (A_{\gamma}^-)^i,
\]
\[
Q_{\gamma}^+ = (I - A_{\gamma}^+)^{-1} = \sum_{i=0}^{\infty} (A_{\gamma}^+)^i,
\]

Since all elements of matrices \( A_{\gamma}^- \) and \( A_{\gamma}^+ \) are non-negative by definition of sec. 9.4.2, all elements \( q_{d',o'} \) and \( q_{d,o'}^+ \) of matrices \( Q_{\gamma}^- \) and \( Q_{\gamma}^+ \) must be non-negative.

Assume for some \((o,d)\) that \( c_{o,d}^- > c_{o,d}^+ \). Since

\[
c_{o,d}^- = \overline{E}_{o,\gamma} Q_{d,\gamma}^-,
\]
\[
c_{o,d}^+ = \overline{E}_{o,\gamma} Q_{d,\gamma}^+,
\]

and

1. all elements of matrices \( E_{\gamma}^- \) and \( E_{\gamma}^+ \) are non-negative by definition of sec. 9.4.2;
2. all elements of matrices \( Q_{\gamma}^- \) and \( Q_{\gamma}^+ \) are non-negative by eqs. 9.53 and 9.54;
3. and all elements of vectors \( \overline{E}_{o,\gamma} \) and \( \overline{E}_{o,\gamma}^+ \) satisfy the inequality \( c_{o,d}^- \leq c_{o,d}^+ \) by definition of sec. 9.4.2,

at least one pair of elements of matrices \( Q_{\gamma}^- \) and \( Q_{\gamma}^+ \) must satisfy the inequality

\[
q_{d',o'}^- > q_{d',o'}^+.
\]

Because of eqs. 9.53 and 9.54 and all elements of matrices \( A_{\gamma}^- \) and \( A_{\gamma}^+ \) are non-negative by definition of sec. 9.4.2, this means that for at least one pair of elements, \( a_{d',o'}^- > a_{d',o'}^+ \), violating the definition of sec. 9.4.2. Hence the inequality \( c_{o,d}^- \leq c_{o,d}^+ \) must hold.
9.4. BOUNDS ANALYSIS

Example
In the banking example, the top-level complexity matrices still represent average estimates

\[
C_{\gamma_1}^- = C_{\gamma_1}^+ = C_{\gamma_1} = \begin{bmatrix}
1 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 1 \\
\end{bmatrix}.
\] (9.58)

The complexity matrices of the \textit{Make withdrawal} subcomponent \(\gamma_0\) however, now become

\[
C_{\gamma_0}^- = E_{\gamma_0}^{-}(I - A_{\gamma_0}^-)^{-1} = [1 0.9 0.9]
\] (9.59)

and

\[
C_{\gamma_0}^+ = E_{\gamma_0}^{+}(I - A_{\gamma_0}^+)^{-1} = [1 1.0 1.1].
\] (9.60)

\(\square\)

Visit matrices for non-primitive components also still describe how non-primitive components visit hardware platform services. The \textit{minimum} and \textit{maximum visit matrices for non-primitive component} \(\gamma\), \(V_{\gamma}^-\) and \(V_{\gamma}^+\), are matrices \(V_{\gamma}^- = [v_{\sigma}^{\sigma-}]\) and \(V_{\gamma}^+ = [v_{\sigma}^{\sigma+}]\). Elements \(v_{\sigma}^{\sigma-} \leq v_{\sigma}^{\sigma+}\) of these matrices bound the numbers of times service \(\sigma\) is visited per execution of operation \(o\) on non-primitive component \(\gamma\).

The \textit{minimum} and \textit{maximum visit matrices for component} \(\gamma\) are derived as follows

\[
V_{\gamma}^- = C_{\gamma}^- V_{\gamma}^- ,
\] (9.61)

\[
V_{\gamma}^+ = C_{\gamma}^+ V_{\gamma}^+ .
\] (9.62)

The inequality \(v_{\sigma}^{\sigma-} \leq v_{\sigma}^{\sigma+}\) is demonstrated as follows: Again, let \(X_{i, \gamma}\) be the \(i\)th column vector and \(\overline{X}_{i, \gamma}\) be the \(i\)th row vector of any matrix \(X_{\gamma}\).

The inequality \(v_{\sigma}^{\sigma-} \leq v_{\sigma}^{\sigma+}\) must hold since

\[
v_{\sigma}^{\sigma-} = \overline{C}_{\sigma, \gamma} V_{\sigma, \gamma}^{-}
\] (9.63)

\[
v_{\sigma}^{\sigma+} = \overline{C}_{\sigma, \gamma} V_{\sigma, \gamma}^{+},
\] (9.64)

and the previous two paragraphs have demonstrated that

1. all elements of matrices \(C_{\gamma}^-\) and \(C_{\gamma}^+\) are non-negative;

2. all elements of matrices \(V_{\gamma}^-\) and \(V_{\gamma}^+\) are non-negative;

3. all elements of vectors \(\overline{C}_{\sigma, \gamma}^-\) and \(\overline{C}_{\sigma, \gamma}^+\) satisfy the inequality \(c_{\sigma, \sigma'}^- \leq c_{\sigma, \sigma'}^+\);

and the induction hypothesis on page 169 states that

4. all elements of vectors \(V_{\sigma, \gamma}^-\) and \(V_{\sigma, \gamma}^+\) satisfy the inequality \(v_{\sigma, \sigma'}^- \leq v_{\sigma, \sigma'}^+.\)
This completes the induction step.

Example
In the Customer transaction example, the minimum and maximum visit matrices for non-primitive component \( \gamma_5 \) become
\[
V_\gamma^{-} = C_\gamma^{-} V_\gamma^{+} = \begin{bmatrix} 28500 & 6.5 & 19 \end{bmatrix} \tag{9.65}
\]
and
\[
V_\gamma^{+} = C_\gamma^{+} V_\gamma^{+} = \begin{bmatrix} 30500 & 7.3 & 21 \end{bmatrix}. \tag{9.66}
\]
\( \square \)

9.4.4 Workload and performance models

9.4.4.1 The projected workload module

The minimum and maximum visit matrices for the top-level components in the design specification bound the workload of the projected application. They are the minimum and maximum visit matrices for workload module \( W_5 \).

Example
In the Customer transaction example, the minimum and maximum workload module visit matrices become
\[
V_S^{-} = V_{S_1}^{-} = \begin{bmatrix} 70000 & 5.5 & 80 \\ 110000 & 18.8 & 80 \\ 30000 & 3.0 & 40 \\ 38500 & 7.0 & 39 \end{bmatrix} \tag{9.67}
\]
and
\[
V_S^{+} = V_{S_1}^{+} = \begin{bmatrix} 70000 & 5.5 & 80 \\ 110000 & 18.8 & 80 \\ 30000 & 3.0 & 40 \\ 40500 & 7.8 & 41 \end{bmatrix}. \tag{9.68}
\]
The organisation workload remains unchanged. However, it will turn out in the following sec. 9.4.4.6 that the transaction workload intensities of the basic framework could advantageously have been approximated by terminal type operations. \( \square \)

9.4.4.2 The existing workload modules

Again the projected application is likely to compete with other, already existing applications for computing resources, unless it will run alone on a dedicated computer. Therefore, additional existing workload modules are needed in addition to the projected one. This existing workload model may be

- represented as average visit matrices as in the basic framework, or
• represented as pairs of minimum and maximum visit matrices — as is the projected workload module

While the projected workload model can be derived automatically from annotated design specifications according to the workload derivation method of sec. 8.2, the existing workload modules must be derived by conventional means [119, 78], which are mostly oriented towards obtaining average parameters. Therefore, the latter alternative is likely to be less useful.

Example
In the Customer transaction example, the existing workload modules retain their average representations. □

9.4.4.3 The workload model

As in the average analysis case of secs. 8.3.4 (for the basic framework) and 9.3.2 (for software platform modelling), the projected and existing workload models must be combined to form an overall workload model for input to performance model analysis. If software platform modelling is not used, this is trivial. However, if software platform modelling is used, the resulting workload model must be analysed according to sec. 9.3.3, once for minimum and once for maximum workload modules. The result is a minimum and a maximum visit matrix for workload model \( W \). Since the workload analysis technique presented there was derived from the workload derivation technique of sec. 8.2, it can be shown that all elements of the minimum visit matrix for the workload model are indeed smaller than the corresponding maximum elements along the lines of sec. 9.4.3.

9.4.4.4 The performance model

As in the basic framework, the performance model represents the hardware that the existing applications run on and that the projected application will run on, and the hardware platform services must be mapped onto performance model service centres as in sec. 8.3.5. For bounds analysis, there are two alternative performance analysis methods:

• average performance analysis, e.g. as introduced in sec. 8.3.7, and

• bounds analysis, e.g. as described in [119].

The former alternative would require two average analyses, one using minimum bounds and the other using maxima. The output of the analyses would be two sets of performance measures corresponding to the minimum and maximum performance bounds of sec. 9.4.4.5. This may be convenient, as the bounds provided are tighter than those provide by "proper" bounds analysis. However, the combination of bounded parameters and analysis of means is in principle not sound. The following sec. 9.4.4.6 will therefore outline a technique for bounds analysis at the hardware level.
Example

As is usual, the queueing network model of Fig. 8.8 is applicable to bounds analysis without changes.

Both the minimum and maximum service times for the CPU, disc, and terminal line are still $7.5 \times 10^{-7}$, $2.5 \times 10^{-3}$, and 0.01 seconds respectively:

$$\tau_1^- = 7.5 \times 10^{-7}, \quad \tau_2^- = 2.5 \times 10^{-3}, \quad \tau_3^- = 0.01,$$

$$\tau_1^+ = 7.5 \times 10^{-7}, \quad \tau_2^+ = 2.5 \times 10^{-3}, \quad \tau_3^+ = 0.01.$$  \hspace{1cm} (9.69) \hspace{1cm} (9.70)

\Box

9.4.4.5 Performance bounds

The available performance bounds again reflect the performance measures of Sec. 8.3.6 exactly. They are

- **minimum and maximum response times** $r_o^-$ and $r_o^+$ and **throughputs** $x_o^-$ and $x_o^+$ for operation $o$, which bound the **responsiveness** of the corresponding software function, and

- **minimum and maximum sojourn times** $z_k^-$ and $z_k^+$ for service centre $k$, which bound the amount of time spent in the corresponding hardware resource *per visit*, queueing included. (Again note that this definition differs from [119].)

In general, analysis provides additional bounds for service centres, such as minimum and maximum total and per class queue lengths $q_k^-, q_k^+, q_k^{o-}, q_k^{o+}$, and minimum and maximum total and per class utilisations $u_k^-, u_k^+, u_k^{o-}, u_k^{o+}$, as well as minimum and maximum total and per class numbers in system, $q^-, q^+, q_o^-$, and $q_o^+$, at the system level. In what follows, let $Z^-_P$ and $Z^+_P$ be the minimum and maximum vectors of service sojourn times $z_o^-$ and $z_o^+$ for platform $P$, and let $R^-_W$ and $R^+_W$ be the minimum and maximum vectors of responsivenesses $r_o^-, r_o^+, x_o^-$ and $x_o^+$, $o \in O_W$ for workload model $W$.

9.4.4.6 Bounds analysis

Sec. 4.4.3 outlined a simple bounds analysis technique from the literature [119], App. C will describe this technique in detail using the terminology and notation of this thesis.
Example
In the Customer transaction example, the minimum and maximum total demands of the Withdrawal operation \( o_4 \) become:

\[
\begin{align*}
\sigma_{o_2}^- &= 28.9 \cdot 10^{-3}, & \sigma_{o_2}^+ &= 17.5 \cdot 10^{-3}, & d_{o_3}^- &= 0.39, \\
\sigma_{o_1}^- &= 30.4 \cdot 10^{-3}, & \sigma_{o_1}^+ &= 19.5 \cdot 10^{-3}, & d_{o_3}^+ &= 0.41.
\end{align*}
\]

This is the only operation of fig. 8.1 that is affected by the now bounded parameters \( a_{o_1, o_1} \) and \( a_{o_1, o_2} \). The corresponding minimum and maximum total demands are

\[
d_{o_4}^- = 0.44, \quad d_{o_4}^+ = 0.46, \tag{9.71}
\]

and

\[
d_{o_4}^* = 0.39, \quad d_{o_4}^{**} = 19.5 \cdot 10^{-3}. \tag{9.72}
\]

The maximum throughput of operation \( o_4 \) Withdrawal is therefore

\[
\lambda_{o_4}^+ = 2.56 \tag{9.73}
\]

The lower bound on response time accordingly is

\[
r_{o_4}^- = 0.44. \tag{9.74}
\]

As commented on above, no maximum response time can be found, as queues may potentially grow unlimited in an open queueing network. \( \square \)

9.5 Parametric Analysis

While secs. 9.2 to 9.3 presented techniques to support and simplify parameter estimation, the previous sec. 9.4 considered cases where obtaining average estimates was difficult — or not even worthwhile — in the early phases of design. This section goes one step even further, considering cases where some parameters are impossible to estimate at the current stage of development.

The technique introduces free performance parameters, for which numerical estimates have not to be provided immediately. In principle, these parameters can be bound at any intermediate stage of analysis, or not until the analysis is finished. However, it will be pointed out that the parametric performance analysis of closed and mixed queueing networks is unfeasible for all but trivial performance models.

First, sec. 9.5.1 discusses the need for parametric analysis, before sec. 9.5.2 extends the performance parameters of the basic framework for parametric analysis. Sec. 9.5.3 accordingly extends the workload derivation method, and sec. 9.5.4 discusses parametric software platform modelling. Finally, sec. 9.5.5 discusses parametric workload and performance models.

9.5.1 The need for parametric analysis

Parametric analysis is applicable to all phases of design, and may be useful when
• some parameter or set of parameters is currently impossible to estimate, but performance predictions are nevertheless needed, or

• some parameter or set of parameters may vary within a range and performance predictions are needed for several points within that range.

Example
The previous sec. 9.4 pointed out that it is difficult to estimate the withdrawal success rate in the Customer transaction example, i.e. that the 96% rate that had been assumed so far was a very coarse estimate. An alternative to bounds analysis early in the development is a parametric analysis. In this analysis, parameters \( a_{o_{10}, o_{11}} \) and \( a_{o_{10}, o_{12}} \) are not fixed prior to initiation of the performance study. Instead, they are regarded as free parameters \( a_{o_{10}, o_{11}} = x \) and \( a_{o_{10}, o_{12}} = 1 - x \) of the (currently unknown) withdrawal acceptance rate \( x \). □

9.5.2 Performance parameters

Parametric analysis is based on leaving some performance parameters free when annotating the design specification with average estimates and/or minimum and maximum bounds. This is useful, since

1. estimating or bounding some parameter or set of parameters can be delayed, and

2. analysis may proceed with alternative values for the parameter or set of parameters in question.

Otherwise, the performance parameters of the basic framework are represented as before and derived as before so that

1. entry matrices,

2. activity matrices, and

3. visit matrices for primitive components,

retain their symbolic representations as average estimates. Parametric minimum and maximum annotations are of course also possible, but seem less useful. Therefore, they are not considered here.

If any parameter of a design specification is free, it becomes a parametric design specification.

9.5.3 Workload derivation

From performance parameters now in the form of either bound numeric values or free linear expressions, workload derivation proceeds as before:
9.5. PARAMETRIC ANALYSIS

Internal visit matrices for components in the design specification are derived using eq. 8.9 (or eqs. 9.47 in the bounds analysis case);

complexity matrices for components in the design specification are derived using eq. 8.16 (or eqs. 9.49 and 9.50 in the bounds analysis case), and

visit matrices for non-primitive components in the design specification are derived using eq. 8.21 (or eqs. 9.61 and 9.62 in the bounds analysis case).

The difference when parametric analysis is applied, is again that any performance parameter may be free, rather than bound to a specific numeric value. The consequence is that the corresponding derived internal performance parameters inside the basic framework are free symbolic expressions rather than bound numeric values.

Note that it can no longer be assumed that the free internal performance parameters are linear, because of

1. the matrix inversion of eq. 8.16 (or eqs. 9.49 and 9.50 in the bounds analysis case) in which all free activity counts are mingled with one another, and

2. the matrix multiplications of eq. 8.21 (or eqs. 9.61 and 9.62 in the bounds analysis case), in which all free complexity counts (which may no longer be linear) and visit matrices are mingled with one another.

The result of workload analysis is a set of workload module visit matrices for the projected and existing applications where some of the elements are free symbolic expressions, rather than bound numeric values.

A parametric workload module is a workload module whose visit matrix contains elements which are symbolic expressions of free variables. A non-parametric workload module is a workload module whose complexity matrix contains only elements which are bound numeric values.

Example
Applying the workload derivation method of sec. 8.2 with free algebraic expressions \( a_{0|0,0|1} = x \) and \( a_{0|1,0|1} = 1 - x \) rather than fixed values, the workload derivation gives

\[
V_{\gamma_1} = \begin{bmatrix}
70000 & 5.5 & 80 \\
110000 & 18 & 80 \\
30000 & 3 & 40 \\
30000 + 10000x & 5.5 + 20x & 40
\end{bmatrix}.
\]  

(9.75)

Of course, insertion of \( x = 0.096 \) into this parametric matrix gives the non-parametric matrix \( V_{\gamma_1} \) of table 8.6 derived in sec. 8.2.3. □

9.5.4 Software platform modelling

Parametric analysis may be applied with or without the software platform modelling technique of sec. 9.3. In either case, the following considerations must be taken. In fact, parametric
analysis can be applied at the software platform level whether or not the design specification itself is parametric.

**Software platform** considerations are not changed when parametric analysis is applied at the application or software platform specification levels.

**The software platform model** may be parametric or not, just as the design specification may be. A design specification may be parametric with its software platform being non-parametric, a software platform may be parametric with its design specification being non-parametric, and both the design specification and its software platform may be parametric.

A **parametric software platform model** is a software platform model with at least one parametric workload module. A non-parametric software platform model is a software platform model with only non-parametric workload modules.

Establishing a parametric software platform specification proceeds as for non-parametric specifications. The only difference is that any parameter or set of parameters of a parametric workload module must be left open.

**Composite workload model analysis** If parametric analysis is applied either at the design specification level, at the software platform specification level, or both, the corresponding projected workload model will be parametric.

Analysis may or may not proceed without binding the free variables first. Software platform analysis with free parameters again calls for symbolic evaluation. The result is a set of collapsed parametric visit matrices of the workload modules representing the projected and existing applications. The difference when parametric analysis has been applied in the overall workload derivation is as usual that the matrices contain symbolic expressions of free variables.

**Example**

In the Customer transaction example, the parametric design specification lead to a parametric top-level visit matrix \( V_{T_1} \). The free variable \( x \) of this top-level visit matrix may or may not be bound before a projected workload module is established. If it is not, the visit matrix of the projected workload module is

\[
V_{S_1} = V_{T_1} = \begin{bmatrix}
70000 & 5.5 & 80 \\
110000 & 18 & 80 \\
30000 & 3.0 & 40 \\
30000 + 10000x & 5.5 + 2.0x & 40
\end{bmatrix},
\]

(9.76)

as derived in eq. 9.75. In the bank example, none of the organisation workloads or visit matrices for existing applications are parametric.

\(\square\)

**9.5.5 Workload and performance models**

Parametric design specifications hence may lead to parametric workload modules. Since parametric specifications of existing applications is equally applicable, such analysis may
lead to parametric existing workload models as well.

Performance analysis proceeds according to secs. 8.3.7 and 9.4.4.6 (apps. B and C) in the average and bounded cases. However, the techniques of bounds and parametric analysis are to some extent complementary, and are not likely to be used together.

In the average case, parametric performance analysis of open queueing networks is feasible. This leads to a set of parametric performance measures. For closed and mixed performance models, however, (average) parametric analysis is infeasible for all but trivial performance models. This is because of the recurrence algorithm used, which quickly leads to complex symbolic expressions of the free parameters. For closed and mixed performance models, free parameters should therefore be bound prior to performance analysis.

In the bounded case, parametric bounds analysis is feasible for all types of networks because the associated analysis techniques are simple.

Example
Since the workload intensities in the bank example are all of the transaction type, parametric performance analysis is feasible. The parametric utilisation at the two queueing centres become

\[ u_{k_1}^o = 8.5 \cdot 10^{-3}, \quad u_{k_1}^a = 4.4 \cdot 10^{-3}, \quad u_{k_1}^m = 4.86 \cdot 10^{-2}, \quad u_{k_1}^c = 60.75 \cdot 10^{-2} + 20.25 \cdot 10^{-2} x, \]
\[ u_{k_2}^o = 2.2 \cdot 10^{-3}, \quad u_{k_2}^a = 2.4 \cdot 10^{-3}, \quad u_{k_2}^m = 1.62 \cdot 10^{-2}, \quad u_{k_2}^c = 37.12 \cdot 10^{-2} + 13.5 \cdot 10^{-2} x. \]

The service centre sojourn times become

\[ z_{k_1} = \frac{7.5 \cdot 10^{-7}}{0.331 - 0.203x}, \quad z_{k_2} = \frac{2.5 \cdot 10^{-3}}{0.608 - 0.135x}, \quad z_{k_3} = 0.01, \quad (9.77) \]

giving e.g. a response time for the important withdrawal operation \( o_4 \) of

\[ r_{o_4} = \frac{7.5 \cdot 10^{-7} \cdot 30000 + 10000 x}{0.331 - 0.203x} + 2.5 \cdot 10^{-3} \cdot \frac{5.5 + 2.0 x}{0.608 - 0.135x} + 0.4 \]
\[ = \frac{0.0225 + 0.0075x}{0.331 - 0.203x} + \frac{0.0138 + 0.005x}{0.608 - 0.135x} + 0.4 \quad (9.79) \]

Insertion of \( x = 0.96 \) into this parametric response time gives

\[ r_{o_4} = \frac{0.0297}{0.137} + \frac{0.0178}{0.479} + 0.4 \]
\[ = 0.22 + 0.4 + 0.4 = 0.66, \]

where the result \( r_{o_4} = 0.66 \) was also derived in sec. 8.3.7. ☐
Chapter 10

Sensitivity Analysis

Sec. 9.2 of the previous chapter 9 presented residence time analysis based on the concept of *sojourn times* for hardware platform services and design specification components. This chapter presents an alternative technique based on *differentiating* the sojourn times of components as defined in eq. 9.2. It is therefore called *sensitivity analysis*.

First, sec. 10.1 discusses the need for sensitivity analysis, while sec. 10.2 presents an overview of the technique. Sec. 10.3 then considers performance model sensitivity analysis, while sec. 10.4 treats design specification sensitivity analysis. Sec. 10.7 finally considers the use of sensitivity analysis.

10.1 The Need for Sensitivity Analysis

As already mentioned in sec. 4.6.2, in this thesis a *sensitivity* will mean the derivative of a performance measure with respect to a performance parameter.

Sensitivity analysis is useful for determining which performance parameters are most important to the per operation and average sojourn times for component operations. If this component is at the top-level of the design specification, sensitivity analysis determines which parameters are most important to the overall responsiveness of the application. However, this chapter provides sensitivities for all operations in the design specification. As for residence time analysis, this is particularly useful for two reasons:

- Sensitivity analysis determines the degree to which each parameter is important for the operation. Thus parameter capture effort can be focused on those parameters.

- Sensitivity analysis determines which parameters that are most crucial to performance. Thus optimising effort can be focused accordingly. Hence rapid and precise *design feedback* is provided [180].

In addition, sec. 4.6.2 summarised several other ways in which the performance evaluator may benefit from sensitivity results.
10.2 Sensitivity Analysis Overview

Since the basic framework views the computerised information system as an application evolving workload on a performance model, sensitivity analysis can be carried out at two distinct levels. Two sensitivity analysis techniques are therefore presented. These are called performance model sensitivity analysis and design specification sensitivity analysis. It will turn out that the former analysis can be carried out in isolation, while the latter analysis technique depends on it. The two will be considered in separate.

An earlier approach to sensitivity analysis of annotated design specifications considered the design specification in isolation [140]. Residence times of software component operations were expressed as functions of service sojourn times and design specification annotations. Differentiating these functions with respect to annotations, sensitivities were obtained. Although important design parameters were successfully suggested in this way, the attempt was inaccurate. The reason is that service centre residence times were regarded as constant with respect to design parameters. This is of course untrue, as is shown in figure 8.9.

An overall picture of the situation was given in fig. 8.9 in the summary of chapter 8. The design specification was annotated with sets of entry \( \{ E_\gamma \} \), activation \( \{ A_\gamma \} \), and visit \( \{ V_\gamma \} \) matrices for components, from which a design specification visit vector \( V_W \) was derived. This vector contained exactly the same information as the visit matrix \( V_W \) derived in the workload derivation of sec. 8.2. The reason a vector representation was needed here will become apparent below.

The visit vector was used for queueing network analysis together with an organisation workload vector \( \Omega_W \) and a service time vector \( \bar{\tau}_M \). Performance analysis in return yielded vectors of design specification operation responsivensneses \( R_W \) and service centre sojourn times \( \bar{Z}_M \). The latter vector was mapped back into a vector of platform service sojourn times \( Z_P = \mu^{-1}(Z_M) \).

This platform sojourn time vector again was used to derive vectors of \( Z_\gamma \) of sojourn times for each component \( \gamma \in \Gamma_S \) in the design specification. The dependency of component operation sojourn times on design specification annotations is therefore not straightforward. In what follows, let \( \mathcal{X} \) be the set of design specification annotations, \( \mathcal{X} = (\{ E_\gamma \}, \{ A_\gamma \}, \{ V_\gamma \}) \).

The vector \( Z_\gamma \) is a function of design specification annotations and platform service sojourn times, \( Z_\gamma(\mathcal{X}, \bar{Z}_P) \). The vector \( Z_P \) is in turn a function \( Z_P(\Omega_W, V_W, \bar{\tau}_M) \) of organisation workload, visit count, and service centre service time vectors, where the vector \( V_W \) is again a function \( V_W(\mathcal{X}) \) of design specification annotations. The resulting composite function therefore has the form \( Z_\gamma(\mathcal{X}, Z_P(\Omega_W, V_W(\mathcal{X}), \bar{\tau}_M)) \). Differentiating it with respect to some design parameter \( x \in \mathcal{X} \) gives

\[
\Xi(\gamma, x) = \frac{\partial Z_\gamma(x)}{\partial x} + \frac{\partial Z_\gamma(Z_P)}{\partial Z_P} \frac{\partial Z_P(V_W)}{\partial V_W} \frac{\partial V_W(x)}{\partial x},
\]

(10.1)

where \( \Xi(\gamma, x) \) is the sensitivity vector of component \( \gamma \) with respect to parameter \( x \), and \( x \) is a design parameter \( x \in \mathcal{X} \). Element \( \xi_\gamma(\gamma, x) \) of this vector is the sensitivity of operation \( o \) with respect to parameter \( x \), accordingly. The remainder of this chapter will derive the right-hand-side terms of eq. 10.1. First, however, some general comments about this equation
will be made.

The five terms of eq. 10.1 have the following forms:

\[ \Xi(\gamma, x) \text{ and } \partial Z_{\gamma}(x) / \partial x \text{ are both vectors of size } |O_\gamma|, \]
\[ \partial Z_{\gamma}(Z_P) / \partial Z_P \text{ is an } |O_\gamma| \times |\Sigma_P| \text{ matrix,} \]
\[ \partial Z_P(V_W) / \partial V_W \text{ is a } |\Sigma_P| \times (|\Sigma_P| \cdot |O_\gamma|) \text{ matrix, and} \]
\[ \partial V_W(x) / \partial x \text{ is a vector of size } |\Sigma_P| \cdot |O_\gamma|, \]

where \(|O_\gamma|\) is the number of operations provided by component \(\gamma\) and \(|\Sigma_P|\) is the number of services provided by hardware platform \(P\).

Although the above result looks complicated, it will turn out that several of the terms are trivial, and that efficient algorithms exist for calculating the more complicated ones. Furthermore, note that only two of the terms of eq. 10.1 are dependent on \(x\). In particular, the term \(\partial Z_P(V_W) / \partial V_W\) representing the change in service centre residence times with changes in visit counts, is the same for all design parameters. This means that a single derivation at the performance model level is sufficient for determining all design specification parameter sensitivities. Although derivation of open queuing networks is simple, the savings are large for closed and mixed queuing networks.

This leads to considering the generality of eq. 10.1. No assumptions about design specification and performance model interiors were made in the above derivation. This implies that the above result holds for any pair of software and hardware modelling approaches, whether static or dynamic and regardless of analysis techniques applied. Hence eq. 10.1 is applicable outside the scope of this thesis.

The derivative \(\partial Z_P(V_W) / \partial V_W\) will be derived in sec. 10.3, while derivates \(\partial Z_{\gamma}(x) / \partial x\), \(\partial Z_P(Z_P) / \partial Z_P\), and \(\partial V_W(x) / \partial x\) are derived in sec. 10.4.

**Example**

In the bank example of chapter 8, the analysis predicted a CPU utilisation as high as 86%. This indicated that the response time estimates were highly sensitive to variations in CPU utilisation, and that small workload fluctuations might degrade performance significantly. It was therefore decided to carry out further performance analyses based on refined design specifications and parameter estimates.

To focus refinement effort, a sensitivity study is now initiated to determine important design parameters. In particular, the effect of rejected withdrawals is to be studied, since the withdrawal operation has a very high arrival rate, and the 96% acceptance estimate \(a_{o_{10}, o_{11}} = 0.96\) of table 8.2 is by no means exact. In other words, a measure of the sensitivity of operation residence times on design parameter \(a_{o_{10}, o_{11}}\) is needed. \(\Box\)

### 10.3 Performance Model Sensitivity Analysis

According to the previous sec. 10.2, the performance analysis of sec. 8.3.7 can be regarded as a pair \((R_W, Z_P)\) of functions

\[ R_W(\Omega_W, V_W, \tau^*_M), \quad Z_P(\Omega_W, V_W, \tau^*_M), \] (10.2)
Figure 10.1: The matrix of possible performance model sensitivity analyses.

where each function can be differentiated with respect to a parameter of any of its three parameter types. This gives rise to a matrix of potential sensitivity analyses, as depicted in fig. 10.1. However, it will turn out that sensitivity analysis is only applicable to think times and not operation population vectors of organisation workloads of the terminal or batch workload types when the basic mean-value-analysis algorithm is applied.

This section strongly focuses on sensitivities of the sojourn time vector $Z_P$ with respect to visit counts $V_M$. The reason is that this is the derivate needed for the overall analysis outlined in sec. 10.2. The other eight possibilities are only outlined as useful alternatives. They are derivable in much the same manner as the main one.

In sec. 8.1.2 it was mentioned that some results of this section would be inspired by the work of Oftedal & Solberg [135]. However, the two formulations differ in that [135] is based on Markov chains. Because of the parallelism inherent in the basic framework, Markov chains are not applicable in this thesis. As a consequence, Oftedal & Solberg’s results have been rederived from first principles.

Secs. 10.3.1, 10.3.2, and 10.3.3 will consider open, closed, and mixed queueing networks in separate.

10.3.1 Open queueing networks

$$
\frac{\partial u_k(\lambda)}{\partial v_{\ell}^{\sigma}} = \begin{cases} 
  x_k \tau_k & (k' = k) \\
  0 & (k' \neq k)
\end{cases} \quad (10.3)
$$

$$
\frac{\partial v_k(\lambda)}{\partial v_{\ell}^{\sigma}} = \begin{cases} 
  0 & (delay \ centres) \\
  \frac{\tau_k}{[1 - u_k(\lambda)]^2} \frac{\partial u_k(\lambda)}{\partial v_{\ell}^{\sigma}} & (queueing \ centres)
\end{cases} \quad (10.4)
$$

Table 10.2: Open queueing network sensitivity analysis technique.
For open queueing networks, the derivatives become particularly simple. Sensitivity analysis proceeds through differentiation of table B.1 of sec. B.1 in the appendix with respect to visit counts \( v'_{k'} \). The resulting algorithm for open networks is shown in table 10.2. Of course, it can easily be extended to provide sensitivities of utilisations, queue lengths, and responsiveness too.

**Example**

Applying the equations of table 10.2 gives

\[
\frac{\partial z_{c'}(v_{w})}{\partial v'_{k'}} = \begin{bmatrix}
4.89 \cdot 10^{-12} & 0.0 & 1.63 \cdot 10^{-12} & 0.0 & 0.0 \\
0.0 & 4.43 \cdot 10^{-6} & 0.0 & 0.0 & 1.48 \cdot 10^{-6} & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
65.2 \cdot 10^{-12} & 0.0 & 81.5 \cdot 10^{-12} & 0.0 & 0.0 & 0.0 \\
0.0 & 59.0 \cdot 10^{-6} & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0
\end{bmatrix} \tag{10.5}
\]

For a closed or mixed queueing network, the derivative would also have a form resembling this matrix. □

### 10.3.2 Closed queueing networks

For closed queueing networks, the algorithm proceeds through differentiation of the equations of table B.2 of sec. B.2 in the appendix with respect to visit counts \( v'_{k'} \). From differentiation of eq. B.33,

\[
\frac{\partial z_{c'}(N+v'_{k'})}{\partial v'_{k'}} = \begin{cases}
\frac{\partial z_{c'}}{\partial v'_{k'}} & \text{(delay centres)} \\
\partial \left[ \tau_k [1 + q_k(N,v'_{k'})] / \partial v'_{k'} \right] & \text{(queueing centres)} \\
0 & \text{(delay centres)} \\
\tau_k \frac{\partial q_k(N,v'_{k'})}{\partial v'_{k'}} & \text{(queueing centres)}
\end{cases} \tag{10.6}
\]

Differentiating eq. B.36 gives

\[
\frac{\partial q_k(N,v'_{k'})}{\partial v'_{k'}} = \partial \left[ \sum_{o \in O_s} x_o(N,v'_{k'}) z_{c'}(N,v'_{k'}) \right] / \partial v'_{k'} \tag{10.8}
\]

\[
= \sum_{o \in O_s} \frac{\partial x_o(N,v'_{k'})}{\partial v'_{k'}} z_{c'}(N,v'_{k'}) + \sum_{o \in O_s} x_o(N,v'_{k'}) \frac{\partial z_{c'}(N,v'_{k'})}{\partial v'_{k'}}, \tag{10.9}
\]

where \( \frac{\partial q_k(N,v'_{k'})}{\partial v'_{k'}} \) is derived by eq. B.33 with operation population \( N \) replacing \( N_{+o} \).

Differentiating eq. B.29 gives

\[
\frac{\partial z_{c'}(N,v'_{k'})}{\partial v'_{k'}} = \partial \left[ x_o(N,v'_{k'}) v'_{k'} \right] / \partial v'_{k'} \tag{10.11}
\]
\[
\frac{\partial x_o(N, v_{k'}^{d'})}{\partial v_{k'}^{d'}} = \left\{
\begin{array}{ll}
\frac{\partial x_o(N, v_{k'}^{d'})}{\partial v_{k'}^{d'}} v_{k'}^{q} & (d' \neq o \lor k' \neq k) \\
\frac{\partial x_o(N, v_{k'}^{d'})}{\partial v_{k'}^{d'}} v_{k'}^{p} + x_o(N, v_{k'}^{d'}) & (d' = o \land k' = k)
\end{array}
\right.,
\]

while differentiating eq. B.28 in turn gives

\[
\frac{\partial x_o(N, v_{k'}^{d'})}{\partial v_{k'}^{d'}} = \partial \left[ \frac{n_o(N)}{\theta_o + r_o(N, v_{k'}^{d'})} \right] / \partial v_{k'}^{d'},
\]

\[
= - \frac{n_o(N)}{[\theta_o + r_o(N, v_{k'}^{d'})]^2} \frac{\partial r_o(N, v_{k'}^{d'})}{\partial v_{k'}^{d'}},
\]

the negation sign being consistent with the intuition that throughput decreases when number of service centre visits increases. From eq. B.37 the final derivate is obtained,

\[
\frac{\partial r_o(N, v_{k'}^{d'})}{\partial v_{k'}^{d'}} = \partial \left[ \sum_{k \in K_M} v_k^{o} x_k^o(N, v_{k'}^{d'}) \right] / \partial v_{k'}^{d'},
\]

\[
= \left\{
\begin{array}{ll}
\sum_{k \in K_M} v_k^{o} \frac{\partial x_k^o(N, v_{k'}^{d'})}{\partial v_{k'}^{d'}} & (d' \neq o) \\
\sum_{k \in K_M} v_k^{o} \frac{\partial x_k^o(N, v_{k'}^{d'})}{\partial v_{k'}^{d'}} + x_k^o(N, v_{k'}^{d'}) & (d' = o)
\end{array}
\right.,
\]

Recurrent application of eqs. 10.6 to 10.16 provides sensitivities for visit count on all operation responsivnesses, as well as service centre sojourn times and utilisations. Of course, when there is only one operation \( o'' \) in the network, \( N = 0_{o''} \),

\[
\frac{\partial x_k^o(0_{o''}, v_{k'}^{d''})}{\partial v_{k'}^{d''}} = \left\{
\begin{array}{ll}
\frac{\partial x_k^o}{\partial v_{k'}^{d''}} & (\text{delay centres}) \\
\partial \left[ \tau_k[1 + 0] \right] / \partial v_{k'}^{d''} & (\text{queueing centres})
\end{array}
\right.,
\]

\[
= \left\{
\begin{array}{ll}
0 & (\text{delay centres}) \\
0 & (\text{queueing centres})
\end{array}
\right.,
\]

since all queue lengths \( q_k(0, v_{k'}^{d''}) \) are zero in an empty network. Again, a forward algorithm would be more computationally efficient. Table 10.3 summarises the resulting algorithm.

Sensitivities with respect to service and think times are derived along the same lines. The corresponding equations are provided in secs. D.1 and D.2 in the appendix.

### 10.3.3 Mixed queueing networks

Again, sensitivity analysis of mixed queueing network is carried out using the equations of the two previous secs. 10.3.1 and 10.3.2 in combination. It will not be outlined in detail in this thesis.

### 10.3.4 Product-form queueing network sensitivity analysis

Sec. 8.3.7 pointed out that under certain conditions, the basic framework could be interfaced with stochastic product-form queueing network models. Under the same set of conditions, the
10.4. Design Specification Sensitivity Analysis

From the previous sec. 10.2 it is known that design specification analysis can be regarded as a triple \((R_Y, \{Z_\gamma\}, V_Y)\) of functions

\[
R_Y(\mathcal{X}, Z_P), \{Z_\gamma(\mathcal{X}, Z_P)\}, \ V_Y(\mathcal{X}),
\]

(10.25)

where each function can be differentiated with respect to a parameter of any of its parameter types. This gives rise to a matrix of potential sensitivity analyses, as depicted in fig. 10.4. However, since design specification sensitivity analysis aims at determining the terms \(\partial Z_\gamma(x)/\partial x, \partial Z_\gamma(Z_P)/\partial Z_P,\) and \(\partial V_Y(x)/\partial x\) of eq. 10.1, these derivatives will be focused on.

Since visit matrix \(V_\gamma\) specifies the number of visits made by each operation of component \(\gamma\) to each platform service, and \(Z_P = \mu^{-1}(Z_M)\) is the vector of platform service sojourn times,
Differentiation with respect to:

<table>
<thead>
<tr>
<th>Entry matrices</th>
<th>Activity matrices</th>
<th>Visit matrices</th>
<th>Sojourn times</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main result</td>
<td>Main result</td>
<td>Main result</td>
<td>(side result)</td>
</tr>
<tr>
<td>(side result)</td>
<td>(side result)</td>
<td>(side result)</td>
<td>(side result)</td>
</tr>
<tr>
<td>Main result</td>
<td>Main result</td>
<td>Main result</td>
<td>Main result</td>
</tr>
</tbody>
</table>

Figure 10.4: The matrix of possible design specification sensitivity analyses.

the sojourn time vector becomes

\[ Z_\gamma = V_\gamma Z_P, \]

as defined in eq. 9.2 of sec. 9.2.3. This gives the trivial first derive

\[ \frac{\partial Z_\gamma(Z_P)}{\partial Z_P} = V_\gamma, \]

where \( V_\gamma \) was calculated for all components \( \gamma \) during workload derivation. \( \partial Z_\gamma(x)/\partial x \) and \( \partial V_\gamma(x)/\partial x \) must now be determined when \( Z_P \) is kept constant, and

\[ \frac{\partial Z_\gamma(x)}{\partial x} = \frac{\partial V_\gamma(x)}{\partial x} Z_P. \]

Furthermore, the term \( \partial V_\gamma(x)/\partial x \) is just the matrix \( \partial V_\gamma(x)/\partial x \) written in vector form, and

\[ V_W = \begin{bmatrix} V_{W_1} \\ \vdots \\ V_{W_n} \end{bmatrix}, \quad W_i \in W \]

as in sec. 8.3.4, where for the projected workload module \( W_i = W_S \),

\[ V_{W_S} = \begin{bmatrix} \bar{V}_{\gamma_1} \\ \vdots \\ \bar{V}_{\gamma_\gamma} \end{bmatrix}, \quad \gamma_i \in \Gamma^*_S \]

as in sec. 8.3.2. Hence, determining the derivate \( \partial V_\gamma(x)/\partial x \) in general provides both terms \( \partial Z_\gamma(x)/\partial x \) and \( \partial V_\gamma(x)/\partial x \) of eq. 10.1.

First, consider the case where \( x \) is a parameter of component \( \gamma \). Non-primitive components only have entry and activity count parameters. Insertion of eq. 8.16 into 8.21 gives

\[ V_\gamma = E_\gamma(I - A_\gamma)^{-1} V_\gamma^*, \]

for non-primitive components \( \gamma \). The derivates are most efficiently computed many at a time as in table 10.5. The (non-trivial) derivations of these equations are given in sec. D.3 in the appendix.
10.4. DESIGN SPECIFICATION SENSITIVITY ANALYSIS

Primitive components only have visit count parameters, and \( \frac{\partial V_\gamma(v_o^o)}{\partial v_o^o} \) is a matrix of zeros, apart from element \((o, \sigma)\) whose value is 1.

Next, consider the case where \( x \) is a parameter of some (possibly indirect) subcomponent \( \gamma_n \) of \( \gamma \). In this case,

\[
\frac{\partial V_\gamma(x)}{\partial x} = C_\gamma \frac{\partial V^*_\gamma(x)}{\partial x},
\]

where \( \frac{\partial V^*_\gamma(x)}{\partial x} \) is calculated by recurrent derivation of visit matrices \( V_{\gamma_1} \ldots V_{\gamma_n} \) of the subcomponents \( \Gamma_{\gamma} \) of \( \gamma \).

Finally, when \( x \) is not a parameter of component \( \gamma \) or any of its (possibly indirect) subcomponents, \( \frac{\partial V_\gamma(x)}{\partial x} \) is a matrix of zeros.

These equations provide all software sensitivities with a minimum of computation. Nevertheless, the cost of performing an exhaustive sensitivity analysis for all design parameters may become prohibitive due to the possibly large number of parameters. Methods to reduce the search space should be sought for. Of course, the scope of a sensitivity analysis can also be restricted to

- certain kinds of parameters, or to
- certain levels of subcomponents.

Let \( X_{i, \gamma} \) be the \( i \)th column vector and \( X_{i, \gamma} \) be the \( i \)th row vector of any matrix \( X_{\gamma} \), then

\[
\frac{\partial v_{o, \sigma}(\overline{E}_{o, \gamma})}{\partial \overline{E}_{o', \gamma}} = \begin{cases} \begin{pmatrix} (I - A_{\gamma})^{-1}V^*_{\sigma, \gamma} & (o = o') \\ 0, \ldots, 0 \end{pmatrix} & (o \neq o') \end{cases} \quad (10.33)
\]

\[
\frac{\partial v_{o, \sigma}(A_{\gamma})}{\partial A_{\gamma}} = [(I - A_{\gamma})^{-1}]^T \overline{E}_{o, \gamma}(V^*_{\sigma, \gamma})^T [(I - A_{\gamma})^{-1}]^T \quad (10.34)
\]

Table 10.5: Non-primitive component sensitivities.

**Example**

In the example, the software component derivates with respect to activity count \( a_{o_0, o_1} \) are shown in table 10.6. The visit count derivate for component \( \gamma_5 \) becomes

\[
\frac{\partial V_{\gamma_5}(a_{o_0, o_1})}{\partial a_{o_0, o_1}} = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} = V_{\gamma_6}, \quad (10.35)
\]

which is intuitively justified since parameter \( a_{o_0, o_1} \) only affects the number of executions of subcomponent \( \gamma_8 \). From this result, derivates \( \frac{\partial V_W(a_{o_0, o_1})}{\partial a_{o_0, o_1}} \) and \( \frac{\partial Z_7(a_{o_0, o_1})}{\partial a_{o_0, o_1}} \) are calculated recurrently as described.

Visit count derivate \( \frac{\partial V_W(a_{o_0, o_1})}{\partial a_{o_0, o_1}} \) shows that only the visit counts of the Withdrawal operation are affected by changes in \( a_{o_0, o_1} \). This is no surprise, since \( a_{o_0, o_1} \) is a parameter of the
Withdrawal subcomponent. Furthermore, visit count derivatives for this operation equals the visit counts of subcomponent \( \gamma_8 \). Again this is expected, since \( a_{o_{10}, o_{11}} \) determines the number of executions of this component.

Residence time derivative \( \partial Z_{\gamma_1}(a_{o_{10}, o_{11}})/\partial a_{o_{10}, o_{11}} \) shows that only the residence time of the Withdrawal operation is affected when \( a_{o_{10}, o_{11}} \) changes, as long as service centre residence times \( R_M \) are kept constant. The change in residence time for this operation with respect to \( a_{o_{10}, o_{11}} \) equals \( V_{\gamma_8} R_M \), as is expected.

Overall sensitivities for the combined software and hardware performance models of the customer transaction system can now be found by combining the software and hardware derivatives of this and the previous section. Eq. 10.1 gives

\[
\Xi(\gamma_1, a_{o_{10}, o_{11}}) = \frac{\partial R_1(a_{o_{10}, o_{11}})}{\partial a_{o_{10}, o_{11}}} + \frac{\partial R_1(Z_P)}{\partial Z_P} \frac{\partial R_M(Z_P)}{\partial V_{\gamma_8}} \frac{\partial V_{\gamma_8}(a_{o_{10}, o_{11}})}{\partial a_{o_{10}, o_{11}}} \quad (10.36)
\]

\[
= \langle 0.0, 0.0, 0.0, 63.7 \cdot 10^{-3} \rangle + \langle 0.58, 0.92, 0.25, 0.33 \rangle \quad (10.37)
\]

\[
= \langle 0.58, 0.92, 0.25, 0.39 \rangle . \quad (10.38)
\]

Thus the sensitivities of top-level operations with respect to the rate of successful withdrawals are 0.58, 0.92, 0.25, and 0.39 respectively. It is interesting to note that these are mostly due to increased resource utilizations at the hardware level when workload increases. The parameter \( a_{o_{10}, o_{11}} \) has little direct impact on operation residence times. This is because the hardware system in question is already close to saturation, and because the parameter considered belongs to a lower-level component of the design specification. \( \Box \)

\[
\frac{\partial Z_{\gamma_1}(a_{o_{10}, o_{11}})}{\partial a_{o_{10}, o_{11}}} = \langle 0, 0, 0, 65.4 \cdot 10^{-3} \rangle
\]

\[
\frac{\partial Z_{\gamma_1}(Z_P)}{\partial Z_P} = \begin{bmatrix}
70000 & 5.5 & 80
110000 & 18 & 80
30000 & 3.0 & 40
39600 & 7.4 & 40
\end{bmatrix}
\]

\[
\frac{\partial V_{\gamma_8}(a_{o_{10}, o_{11}})}{\partial a_{o_{10}, o_{11}}} = \langle 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 10000, 2.0, 0 \rangle
\]

| Table 10.6: Software sensitivity analysis results. |

### 10.5 Average Sensitivity Analysis

However, the sensitivities of the previous sec. 9.2.5 only apply to a single operation of the design specification at a time. Since the projected application is likely to provide many functions, the results may therefore be difficult to comprehend. This section therefore introduces average sojourn time sensitivity analysis as an alternative to the per operation analysis of the previous sec. 10.4.

Average sensitivity analysis considers the sojourn times for components and their operations
in the component layer per *average* execution of an operation of design specification $S$. Since sensitivity analysis should capture the *overall* behaviour of the projected application, this is believed to be more useful than per operation analysis. The analysis relies on the concept of *operation frequency vectors* of components defined in sec. 9.2.6.

The analysis of averaged sensitivities proceeds like the per operation analysis of the previous section. However, the *average sojourn time of component* $\gamma$,

$$z_\gamma = F_\gamma Z_\gamma,$$

(10.39)

is used instead of the vector $Z_\gamma$ of per operation sojourn times. The average sojourn time $z_\gamma$ of component $\gamma$ may vary with operation frequencies $f_o$, $\omega_in O_\gamma$, in addition to the design parameters $x \in \mathcal{X}$ of the previous sec. 10.4.

Both derivatives are easily calculated,

$$\frac{\partial z_\gamma(f_o)}{\partial f_o} = z_o, \quad \text{and}$$

(10.40)

$$\frac{\partial z_\gamma(x)}{\partial x} = F_\gamma \frac{\partial Z_\gamma(x)}{\partial x},$$

(10.41)

where the non-trivial derivate $\frac{\partial Z_\gamma(x)}{\partial x}$ was derived in the previous sec. 10.4.

![Graph showing the residence time of withdrawal operation](image)

**Figure 10.7**: Validation of the sensitivity analysis technique.

### 10.6 Validation

To validate the sensitivity analysis technique of this chapter, the combined software (workload) and hardware performance model can be reevaluated with different values for $a_{o_{10},o_{11}}$. Fig. 10.7 plots the residence time of the “withdrawal” operation $o_4$ for success rates $a_{o_{10},o_{11}} = 0.94, 0.95, 0.97,$ and $0.98$, as well as for the initial estimate $a_{o_{10},o_{11}} = 0.96$ (solid curve). The slope of the tangent to this curve at the point $a_{o_{10},o_{11}} = 0.96$ (dashed curve) represents the
derivate of operation residence time \( r_{o_1} \) with respect to design parameter \( a_{o_{10} \cdot o_{11}} \). In fig. 10.7, this tangent has the form \( r_{o_1} \approx 0.39 \cdot a_{o_{10} \cdot o_{11}} + 0.29 \), and the slope 0.39 equals the analytically predicted sensitivity of the previous sec. 10.4. Similar plots can be made for the other operation residence times of the design specification, and for any other design parameter differentiated with respect to.

Of course, the residence times of the “withdrawal” operation for these success rates could also have been calculated using the parametric expression of eq. 9.79 in sec. 9.5.5.

### 10.7 Use of Sensitivity Analysis

An exhaustive global sensitivity analysis for a component \( \gamma \) implies calculating the sensitivity for every parameter in it, as well as every parameter in its directly and indirectly subordinate components, and sorting the results. Top parameters in this list would again be the most important to the component sojourn time, while middle and lower rank parameters could be regarded as unimportant. In this way, sensitivity analysis not only suggests which parameters to estimate with most care, but also points out where in the application design optimisations have the most impact, as already mentioned.

A global sensitivity analysis restricted to a certain layer of components provides results similar to the residence time analysis of sec. 9.2. Hence the two are alternative techniques which need not be integrated in the framework.

In general, choosing between the residence time analysis and sensitivity analysis is therefore a matter of convenience. Sensitivity analysis is more advanced, precise, and comprehensive. However, its functional superiority comes at a computational cost. For small to medium sized design specifications, sensitivity analysis should probably be used throughout. For larger specifications, a hybrid using global residence time analysis to point out hot-spots for local sensitivity analysis may be preferable.

The need for sensitivity analysis was explained and an overview of the technique was given. Sensitivity analysis depended on separate analyses of software and hardware performance model sensitivity. The sensitivity found was validated against plots of repeated performance analyses.

The technique is promising in making software performance engineering easier and more efficient. The sensitivities provided are useful for

- pointing out where performance modelling and parameter capture effort should be focussed, and

- suggesting performance optimisations in the design specification.

Results similar to those of the example was derived for closed queueing networks also, and sensitivity analysis in connection with other performance analysis techniques was briefly discussed. Sensitivity analysis of dynamic software performance models and specifications of distributed software and hardware systems are other topics for further study.
Chapter 11

Distributed Information Systems

This chapter will extend the basic framework with the capability of predicting the performance of distributed software systems during development. The chapter is strongly inspired by the HIT environment (Beilner [16, 15]), in addition to the other sources [103, 135, 173] already mentioned in the thesis.

The basic framework of chapters 7 and 8 presented results which have been implemented (as will be presented in part IV) and used in a practical case-study [40, 39]. Also, chapters 9 and 10 made extensions to the framework which have been published [139, 143] and thoroughly exemplified. This chapter, on the other hand, considers a problem domain which is — as of today — not fully understood in either the information system engineering or the computer performance evaluation fields. Hence this chapter aims at outlining in detail and justifying a possible solution path, rather than at presenting a final and directly applicable solution.

First, sec. 11.1 discusses the need for performance engineering of distributed information systems, before sec. 11.2 presents an overview of the approach. Sec. 11.3 discusses aggregation of the distributed performance model at the hardware level. Sec. 11.4 uses these aggregates for overall performance analysis of generalised stochastic Petri-net (GSPN) models derived from the top-level distributed PrM specification. Sec. 11.5 supports aggregation of GSPN models derived from intermediate-level PrM specifications for cases with more than one level of distribution. Finally, sec. 11.6 outlines how the results of this chapter can be tailored to support representation of software synchronisation and blocking in non-distributed systems.

While the framework has so far focused on both performance prediction and improvement while touching related issues, this chapter will focus on performance prediction only.

11.1 The Need for Performance Engineering of Distributed Information Systems

Chapter 2 explained why performance engineering is becoming mandatory when developing (non-distributed) information systems. Design of distributed software systems calls for performance engineering even more, because:
Figure 11.1: Extended design specification for the accounting software and hardware system. Allocation of components among computers is also indicated.

1. the complexity of distributed systems makes intuitive performance assessment unfeasible,

2. the number of design alternatives grow when software components can be allocated to several computers, and

3. performance is likely to be important in the first place when a distributed system has been called for.

Although some contemporary approaches to software performance engineering (e.g. Booth et al. [36] and Smith [173]) also consider distributed software system analysis, they are based on specialised, lower-level, flow-chart type software specifications.

Example
In the bank example, the Accounting application of figs. 3.6, 3.9, and 3.10 is distributed. Fig. 11.1 shows

1. its design specification;
2. the computers it is implemented on, as well as
3. which components in the design specification that are to be implemented on which computers.

The Present menu and Present result components both run on a Screen manager minicomputer, while the Get balances and the Update balances components as well as their subcomponents run on
a Database manager mainframe. The two computers communicate through a Local area network. This distributed information system will be used as an example throughout the chapter. □

The term implemented on will be precisely defined in a moment. As indicated by point 3 in the above example, a requirement of this chapter is that all primitive components in the design specification are to be implemented on one and only one computer each, as will be explained in secs. 11.3.1.

11.2 Overview of Distributed Information System Analysis

The basic framework provides performance predictions from

- a static workload model derived from the design specification to represent the projected application, and

- a dynamic performance model established to represent the computer hardware.

As a result, software responsiveness (and other performance measures) are obtained directly from performance analysis of the software workload. For distributed software systems, however, this is insufficient because software response times and throughputs will also depend on the degree of parallelism inherent in the design specification. This calls for an extended framework based on an additional dynamic model representing software parallelism. Since this implies dynamic performance analysis at at least two levels (i.e. the hardware level and at least one level of software parallelism), the situation calls for aggregate analysis [44].

Aggregation must proceed bottom-up from the distributed performance model through the distributed design specification. Dynamic software analyses are needed at each point in the design specification where parallelism is introduced. Sec. 8.3.5 pointed out why operational analysis of queueing network models was a natural technique for performance analysis at the hardware level for the purposes of this thesis. However, these queueing networks are not appropriate for analysis of parallel software systems, since no synchronisation mechanisms are provided. (However, several analytically intractable queueing network extensions exist, e.g. simulative QNP2 models [151].) Therefore, this chapter will use generalised stochastic Petri-nets [2] for performance analysis at the distributed software level.

Analysis proceeds

1. from bottom-level aggregation of “almost separable” performance submodels at the hardware level;

2. possibly through aggregations of generalised stochastic Petri-net models at intermediate levels of distribution;

3. to analysis of a generalised stochastic Petri-net model of the distributed software system at the highest level.
Each higher-level analysis is based on aggregates derived from the next lower ones. In the best case, the distributed design specification has only two levels of distribution, and the top-level GSPN analysis embeds the almost-separable submodel aggregates directly. Since the submodel aggregates feature almost separability, the top-level GSPN analysis is likely to be accurate, because the difference in abstraction level is usually large. However, it must always be remembered when using the technique that it is approximate. Multiple small aggregation errors may have disastrous effects for the overall accuracy of the performance prediction when combined. Understanding the subtle interactions and errors involved requires a great deal of skill for the performance analyst. Hence, the danger of large aggregation errors is in strong conflict with the aim of the thesis of making performance engineering easily available to software developers. As is pointed out in sec. 14.3 an important path for further work is therefore to extend the framework with confidence intervals and/or error bounds.

Cases where parallelism is introduced at several different abstraction levels in the distributed design specification will be considered in sec. 11.5.

<table>
<thead>
<tr>
<th>Distributed hardware platform $P$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P = ({M_1, M_2}, {C_1})$</td>
</tr>
<tr>
<td>$M_1 = $ Screen manager</td>
</tr>
<tr>
<td>$M_2 = $ Database manager</td>
</tr>
<tr>
<td>$C_1 = $ Local area network</td>
</tr>
</tbody>
</table>

Services provided by $P$:  

$\Sigma_{M_1} = \{\sigma_1, \sigma_2\}$  
$\sigma_1 = $ Execute instruction  
$\sigma_2 = $ Access disk  

$\Sigma_{M_2} = \{\sigma_3, \sigma_4, \sigma_5\}$  
$\sigma_3 = $ Execute instruction  
$\sigma_4 = $ Access disc A  
$\sigma_5 = $ Access disc B  

$\Sigma_{C_1} = \{\sigma_6\}$  
$\sigma_6 = $ Transmit line

Table 11.2: Distributed hardware platform for the Accounting model of fig. 3.6.

11.3 Distributed Performance Submodel Aggregation

11.3.1 Distributed hardware platforms

The key problem in the overview of the previous sec. 11.2 is to determine the points in the design specification where parallelism is introduced. Distribution in an information system means that different parts of the system use disjunct sets of services at the hardware level. In distributed systems, this hardware level comprises a system of computers rather than a single computer. Each computer in the computer system will therefore be called a computer subsystem in this chapter. In addition, the computer system comprises a set of communi-
11.3. DISTRIBUTED PERFORMANCE SUBMODEL AGGREGATION

Figure 11.3: Queueing network performance model for the distributed hardware platform of table 11.2.

cation channels connecting the computer subsystems. Furthermore, a design specification component is said to be implemented on a (set of) computer subsystem(s) iff the part of the distributed application it represents uses resources provided by that (those) computer subsystem(s), and by no other subsystems. In addition, the component may of course use communication channel resources.

However, the hardware platforms introduced in sec. 8.1.3 were just sets of services provided by a single computer, containing no notion of computer subsystems. This view must now be refined.

In this chapter, a hardware platform $P$ may be either distributed or non-distributed. In either case, platform $P$ is defined in terms of a set $\mathcal{M}$ of virtual machines $M \in \mathcal{M}$ and a set $C$ of communication services $\sigma \in C$, so that

$$P = (\mathcal{M}, C). \quad (11.1)$$

A virtual machine $M$ is just a set of services $\sigma \in \Sigma_M$. Hence a (non-distributed) hardware platform of the basic framework corresponds to a single virtual machine, which is in turn just a set of services $\sigma \in M$.

A non-distributed hardware platform $P = (\{M\}, \{\})$ contains a set of exactly one virtual machine $M$ and an empty set of communication services. Although this view has been strongly inspired by the sp language, the differences are again deliberate. Since not all the power of sp is needed, the extensions to the basic framework have been limited as much as possible. A distributed hardware platform $P = (\mathcal{M}, C)$, on the other hand, contains a set $\mathcal{M}$ of more than one virtual machine $M \in \mathcal{M}$ and a set $C$ of at least one communication service $\sigma \in \Sigma_C$.

Again, a design specification component is implemented on a (set of) virtual machine(s) iff it visits services provided by that (those) virtual machine(s), and by no other machines. In addition, the component may of course visit communication services.
A distributed hardware platform for the projected Accounting application is shown in table 11.2. This hardware platform contains two virtual machines \( M_1 \) and \( M_2 \) corresponding to the Screen manager and Database manager computers of fig. 11.1. In addition, there is a single communication service \( \sigma_6 \) representing the local area network. □

A requirement introduced by this in refined hardware platform definition, is that a primitive component in the design specification may not use services provided by more than one virtual machine. Hence primitive components are only allowed to use services provided by a single virtual machine, as well as any communication services. This is the primitive component rule. Since the platforms of the basic framework of chapters 7 and 8 consisted only of a single virtual machine, this requirement has been implicitly satisfied until now. For distributed design specifications at a high level of abstraction early in development, this rule may not be possible to satisfy. In such cases, the mean-value analysis techniques of this chapter are not applicable.

Since the services of the platform were mapped onto performance model service centres in sec. 8.3.5, the extended platform definition has consequences for the view of performance models presented there also. A performance model \( M \) now becomes either distributed or non-distributed. In either case, performance model \( M \) is defined in terms of a set \( \mathcal{M}' \) of performance submodels \( M' \in \mathcal{M}' \) and a set \( K \) of communication service centres \( k \in K \), so that

\[
M' = (\mathcal{M}', K).
\]  

A performance submodel \( M' \) corresponds exactly to the concept of performance model as presented in sec. 8.3.5. A non-distributed performance model \( M = (\{M\}', \{\}) \) contains a set of exactly one performance submodel \( M' \) and an empty set of communication service centres. Hence a (non-distributed) performance model in this chapter corresponds to a single performance submodel, which is in turn just a performance model in the sense of chapters 7 and 8. A distributed performance model \( M = (\mathcal{M}', K) \) contains a set \( \mathcal{M}' \) of more than one performance submodel \( M' \in \mathcal{M}' \) and a set \( K \) of at least one communication service centre \( k \in K \). Furthermore, service centres of a performance submodel can only be connected to other service centres of that performance submodel, and to communication service centres. Performance submodels which are connected to the same communication centres are said to be adjacent.

A design specification component is implemented on a (set of) performance submodel(s) iff the (possibly singular) set of virtual machines it is implemented on is mapped onto that (those) performance submodels, and onto no other submodels.

Hence the extended platform and performance model views correspond to one another. Furthermore, the platforms and performance models of sec. 8.3.5 correspond to the non-distributed platforms and performance models of this chapter.

The platform mapping \( \mu \) does not have to be refined to account for the extended platforms and performance models. It still maps platform services \( \sigma \) onto performance model service centres \( k \). However, three platform mapping rules now come into effect:
1. virtual machine services can only be mapped onto performance submodel service centres;
2. communication services can only be mapped onto communication service centres, and
3. if a service of virtual machine $M$ is mapped onto a service centre of performance submodel $M'$, then all services of virtual machine $M$ must be mapped onto service centres of performance submodel $M'$.

However, there is still no requirement that the mapping should be one-to-one, so that several platform services can be mapped onto the same service centre. In addition, it now becomes possible to map several virtual machines onto the same performance submodel. Thus it becomes possible to annotate the design specification in terms of a highly distributed platform, and then mapping it onto a range of performance models with varying degree of parallelism. This represents a major advantage of the platform mapping framework.

Example
A distributed queueing network performance model for the Accounting application is shown in fig. 11.3. The queueing network contains two performance submodels and a communication service centre, corresponding to the two virtual machines and the communication centre in the distributed platform of table 11.2.

Since both the service centres and the performance submodels of this queueing network are one-to-one with the services and virtual machines of the platform, the platform mapping $\mu$ again becomes trivial.

11.3.2 Primitive subspecifications

The extended framework is based on the concept of subspecifications within the design specification. A subspecification is a subset of components in the design specification which use only the same subset of performance model service centres. Subspecifications are therefore defined in terms of performance model service centres and not platform services. This is very important, since several virtual machines can be mapped onto the same performance submodel, as explained in sec. 11.3.1. Before introducing primitive subspecifications, the notion of “almost separability” is needed.

A performance submodel is almost separable iff it possesses such features as

1. weak interaction and
2. little dependency

on the other performance submodels and communication service centres. This must be considered for every distributed performance model in isolation. In most cases, a performance submodel is almost separable as long as the utilisations of its adjacent communication service centres are low. This can be ensured by inspection of the corresponding columns of the projected workload module visit matrix of sec. 8.3.2. If some performance submodel is not almost separable, it can be combined with some adjacent submodels through the heavily
utilised communication service centre, until the combined submodel becomes almost separable. This approach introduces additional concepts and requires a refined notation, without deviating much from the basic ideas of the extended framework. In this chapter, only almost separable performance submodels will be considered.

A primitive subspecification over performance submodel $M'$ and communication service centres $K$, $S'(\{M'\}, K)$ is a set of components $\gamma \in \Gamma_S$ so that

1. all components $\gamma \in \Gamma_S$ are implemented only on performance submodel $M'$, and
2. no other components $\gamma \notin \Gamma_S$ are implemented only on performance submodel $M'$.

Furthermore, $K$ is the set of communication service centres used by the components $\gamma \in \Gamma_S$.

The first of the above two points implies that design specification components above a certain level of decomposition are excluded from the primitive subspecification since such components will be implemented on more than one performance submodel. Also, the first point implies that only some components below this level will be included, because other components may be implemented on other performance submodel. The latter point 2 states that all components below the level of decomposition and implemented on the right performance submodel are included in the primitive subspecification.

Hence a primitive subspecification $S'$ is a set of components in a design specification so that

1. the performance of all components in the subspecification depends "only" on the performance of its performance submodel, and
2. all such components are included in the set.

Therefore, the primitive subspecification constitutes the largest possible set of components to consider when aggregating performance submodel $M'$ corresponding to one or several virtual machines. This will only yield acceptable results when the performance submodel of the primitive subspecification is almost separable. If not, several primitive subspecifications must be combined until the corresponding combined performance submodels are almost separable, explaining why the word "only" was quoted above.

Example

The primitive Screen manager and Database manager subspecifications of fig. 11.1 is shown in fig. 11.4. \Box

11.3.3 Performance submodel aggregation

The previous sec. 11.3.2 stated that a performance submodel was "almost separable" iff it possessed such features as

1. weak interaction and
2. Little dependency

on the other performance submodels and communication service centres. In most cases, it sufficed that the utilisations of its adjacent communication service centres are low.

Assuming almost separability, each performance submodel corresponds to a primitive subspecification. Since a primitive component is implemented on only one performance submodel, this subspecification comprises at least these primitive components, as well as all their non-distributed superiors.

The performance submodel is aggregated with

1. a set of customer classes corresponding to the operations provided by the top-level components corresponding to primitive subspecification;

2. a subspecification visit matrix derived through workload analysis of the subspecification, and

3. a maximum population vector for subspecification \( S' \), derived top-down from the maximum population vector of design specification \( S \).

Each will be considered in separate.

**Customer classes** A primitive subspecification has a set of top-level components, and a (potentially overlapping) set of primitive components, as has the design specification itself.
Aggregate representation $A_{S'}$ for subspecification $S' = \text{Screen manager}$:

$$A_{S'} = (a_{o_3}, a_{o_4}, a_{o_5}, a_{o_6})$$

- $o_3 = \text{Initiate session}$
- $o_4 = \text{Next transaction}$
- $o_5 = \text{Gotten balances}$
- $o_6 = \text{Updated balances}$

$$a_{o_3} = \{z_{o_3}(\tilde{N})|\tilde{N} \geq \hat{N} \geq \tilde{0}\}$$
$$a_{o_4} = \{z_{o_4}(\tilde{N})|\tilde{N} \geq \hat{N} \geq \tilde{0}\}$$
$$a_{o_5} = \{z_{o_5}(\tilde{N})|\tilde{N} \geq \hat{N} \geq \tilde{0}\}$$
$$a_{o_6} = \{z_{o_6}(\tilde{N})|\tilde{N} \geq \hat{N} \geq \tilde{0}\}$$

Table 11.5: Form of the \text{Screen manager} subspecification aggregate.

Aggregate representation $A_{\gamma_1}$ for component $\gamma_1 = \text{Present menu}$ of subspecification $S' = \text{Screen manager}$:

$$A_{\gamma_1} = (a_{o_3}, a_{o_4})$$

- $o_3 = \text{Initiate session}$
- $o_4 = \text{Next transaction}$

Aggregate representation $A_{\gamma_2}$ for component $\gamma_2 = \text{Present results}$ of subspecification $S' = \text{Screen manager}$:

$$A_{\gamma_2} = (a_{o_5}, a_{o_6})$$

- $o_5 = \text{Gotten balances}$
- $o_6 = \text{Updated balances}$

Table 11.6: Form of the component aggregates of the \text{Screen manager} subspecification.

The set of top-level components of subspecification $S'$ is the \textit{top-level flake for subspecification} $S'$, $\mathcal{F}_{S'}$, while its set of primitive components constitute a \textit{bottom-level flake for subspecification} $S'$, $\mathcal{F}_{S'}$, accordingly. Furthermore, the operations provided by the top-level components of a primitive subspecification are the external operations of that subspecification. The operations provided by the primitive components of a subspecification are its internal operations, accordingly.

The set of customer classes (1) corresponds to the set of operations provided by the top-level components of the primitive subspecification.

**Subspecification visit matrices** The projected workload module visit matrix (2) is derived as in sec. 8.2 with the primitive subspecification replacing the design specification.
11.3. DISTRIBUTED PERFORMANCE SUBMODEL AGGREGATION

Maximum population vectors  As explained in sec. 4.6.1, aggregate analysis is a short-circuit analysis of a performance submodel for all possible populations. Therefore, a maximum population vector \( \mathbf{N}_S \) for primitive subspecification \( S' \) must be derived before its performance submodel can be aggregated.

To determine the maximum population of subspecification \( S' \), the maximum population of the overall design specification \( S \) must also be known. This is in general not the case for open or mixed queuing networks, since any number of transaction operations may enter the system during any interval with non-zero probability. Therefore, the average distributed information system analysis techniques of this chapter are only applicable to closed queuing networks, or to open queuing networks approximated with closed networks as in sec. 8.3.1. For such queuing networks, the population vector \( \mathbf{N} \) denotes the maximum (and also the fixed) number of operations in the system.

\[
\text{Operation list } O_{m_1} \text{ for top-level model } m_1 = \text{Accounting}:
\]

\[
O_{m_1} = (o_1, o_2)
\]

\( o_1 = \text{Initiate session} \quad o_2 = \text{Check balance} \)

\[
\text{Flow list } F_{m_1} \text{ for operation list } O_{m_1}:
\]

\[
F_{m_1} = (f_{o_1}, f_{o_2})
\]

\( f_{o_1} = \text{Initiate session} \quad f_{o_2} = \text{Check balance} \)

Table 11.7: Operation list for the PrM model of fig. 3.6.

\[
\text{Organisation workload } \Omega_{m_1} \text{ on operation list } O_{m_1}:
\]

\[
\Omega_{m_1} = (\omega_{o_1}, \omega_{o_2})
\]

\( \omega_{o_1}: \text{ type is terminal, } n_{o_1} = 540, \quad \theta_{o_1} = 2903.2 \)

\( \omega_{o_2}: \text{ type is terminal, } n_{o_2} = 59, \quad \theta_{o_2} = 1223.6 \)

Table 11.8: Organisation workload for the PrM model of fig. 3.6.

Example
The operations provided by the Accounting design specification of fig. 11.1 are shown in table 11.7.

The organisation workload for the application is specified in table 11.8 in terms of this list. The 500 Clerks currently initiate 6200 sessions and the 55 Accountants check 1618 balances a day. The initial assumption is that this will increase with 8% before the Accounting application is put in production. This leads to terminal workload intensities with numbers of users of approx. 540 and 59 and think times of 2903.2 and 1223.6 seconds, accordingly. (Again under the simplistic assumption that all transactions are be processed during the same 10 hour day they are requested, and are evenly distributed.) The corresponding queuing network of fig. 11.3 is therefore closed.

For both operations, the workload intensities approximate the transaction type, because the number of software system users is large and each user uses the system infrequently. This is the usual case for
The complexity matrix $C_{F_{S'} S''}$ for flake $F_{S'}$ under design specification $S'$ was defined in sec. 9.2.4. This matrix represents the number of times each operation provided by components in flake $F_{S'}$ is used per execution of each design specification operations. Since the population vector $\vec{N}_S$ is maximum number of simultaneous uses of these design specification operations, the maximum population vector for subspecification $S'$ becomes

$$\vec{N}_{S'} = \vec{N}_S C_{F_{S'} S''}. \tag{11.3}$$

**Subspecification aggregates** The performance measures to collect from short-circuit queuing network analysis for each population vector $\vec{N} \leq \vec{N}$ are the throughputs, $\lambda_0$, for each operation $o$ to represent the sojourn time $z_o(\vec{N})$ in the aggregate for population $\vec{N}$. $z_o(\vec{N}) = n_o/\lambda_o$. The sojourn times for every subspecification population bounded by the maximum population vector defines the corresponding aggregate for subspecification $S', A_{S'}$.

When annotating GSPN models at the next-higher level of subspecifications, the concept of aggregate for component $\gamma$, $A_{\gamma}$, will turn out to be convenient. This aggregate is defined for each top-level component $\gamma$ of the subspecification, as the corresponding set of columns of $A_{S'}$.

**Example**

The form of the aggregate for the Screen manager subspecification is shown in table 11.5. Also, the forms of the Present menu and Present result component aggregates for the Screen manager subspecification are shown in table 11.6. □

### 11.4 Top-Level Subspecification Analysis

The aggregates of sec. 11.3 represent the sojourn times of each performance submodel under varying load. The software system response time depends on the degree of parallelism between these submodels. In the best case of maximum parallelism, the response time corresponds to the maximum subspecification sojourn time. In the worst, sequential case, it is the sum of subspecification sojourn times. To obtain an average estimate, a notion is needed of where in the distributed software specification parallelism is introduced.

#### 11.4.1 Non-primitive subspecifications

The concept of primitive subspecifications sufficed for the performance submodel aggregation of the previous sec. 11.3. For the purposes of this section and the following sec. 11.5, however, a generalisation of this definition is required.

A non-primitive subspecification over performance submodels $M'$ and communication service centres $K$, $S'(M', K)$, is a set of components $\gamma \in \Gamma_{S'}$ so that
1. all components $\gamma \in \Gamma_{S'}$ are implemented only on the set $\mathcal{M}'$ of performance submodels, and

2. no components $\gamma \notin \Gamma_{S'}$ are implemented only on this set $\mathcal{M}'$ of performance submodels.

Again, $K$ is the set of communication service centres used by these components $\gamma \in \Gamma_{S'}$.

Again, the first of the above two points implies that design specification components above a certain level of decomposition are excluded from the non-primitive subspecification since such components will be implemented on other performance submodels in addition to those in $\mathcal{M}'$. Also, the first point implies that only some components below this level will be included, because other components may be implemented on performance submodels not in $\mathcal{M}'$. The latter point 2 states that all components below the level of decomposition and implemented on the right performance submodels are included in the non-primitive subspecification.

Hence a non-primitive subspecification $S'$ is a set of components in a design specification so that

1. the performance of all components in the subspecification depends “only” on the performances of their performance submodels, and

2. all such components are included in the set.

Therefore, the non-primitive subspecification constitutes the largest possible set of components to consider when aggregating a set of performance submodels corresponding to a set of virtual machines. Note that this assumption only holds if the set of performance submodels of the non-primitive subspecification are almost separable. If not, several non-primitive subspecifications must again be combined until the corresponding set combined performance submodels are separable.

**Example**

The top-level and primitive subspecifications in the two-level subspecification hierarchy of fig. 11.4 are shown in fig. 11.9. □

### 11.4.2 The hierarchy of subspecifications

A subspecification $S'(\mathcal{M}', K')$ is **superior** to subspecification $S''(\mathcal{M}'', K'')$ iff $\mathcal{M}' \supset \mathcal{M}''$. If so, subspecification $S''$ is also **subordinate** to $S'$. Furthermore, subspecification $S'$ is **directly superior** to subspecification $S''$ iff there is no subspecification $S'''$ which is superior to $S''$ and subordinate to $S'$, $S''' \neq S'$ and $S''' \neq S''$. In that case, $S''$ is also **directly subordinate** to $S'$.

These concepts defined a **subspecification hierarchy** $\mathcal{H}'$, which can be constructed bottom-up from the annotated design specification. This hierarchy contains both **primitive** and **top-level** subspecifications, and possibly **intermediate** subspecifications as well.
If the distributed software specification has only two levels of subspecifications, all subspecifications except the top-level one has been aggregated by the performance submodel analyses. A GSPN model of this top-level subspecification must now be generated. Since the submodel aggregates have been assumed to be almost separable, the top-level GSPN analysis is likely to be accurate, because the difference in abstraction level is large.

The concepts of top-level and bottom-level flakes for non-primitive subspecifications can now be defined. They correspond exactly to the definitions given for primitive subspecifications in sec. 11.3.2, with one important exception. A non-primitive subspecification has a set of top-level components, as has the design specification itself. The set of top-level components of subspecification $S'$ is the **top-level flake for non-primitive subspecification** $S'$, $F^s_{S'}$. Furthermore, the operations provided by the top-level components of a non-primitive subspecification are the external operations of that subspecification.

However, its set of primitive components does no longer constitute its bottom-level flake. Instead, the **bottom-level flake for non-primitive subspecification** $S'$, $F_{S'}$, is defined as the set of components $\gamma \in \Gamma_S$ that are also **top-level components** of its directly subordinate subspecifications. The operations provided by the bottom-level flake components of a subspecification, are its internal operations accordingly.
Connectives $C^p_1$ for process $p_1 =$ Present menu:

$$C^p_1 = (C^p_1 i, C^p_1 o)$$

$$C^p_1 i = (\text{XOR } f_1 (T f_2))$$

$f_1 =$ Initiate session $f_2 =$ Next transaction

$$C^p_1 o = (\text{XOR } (0.074 \text{ (REP } 1.13 (T f_3))) (0.926 \text{ (T f_4)}))$$

$f_3 =$ Get balances $f_4 =$ Update balances

Connectives $C^p_3$ for process $p_3 =$ Update balances:

$$C^p_3 = (C^p_3 i, C^p_3 o)$$

$$C^p_3 i = (\text{AND } f_5 (T f_6))$$

$f_5 =$ Old balances $f_6 =$ Update balances

$$C^p_3 o = (\text{AND } (\text{REP } 1.24 f_7) \text{ (COND } 0.0058 f_8) (T f_9))$$

$f_7 =$ New balances $f_8 =$ Notify accountant $f_9 =$ Updated balances

Table 11.10: Execution characterisations for the Present menu and Update balances processes of fig. 3.6 in textual form.

### 11.4.3 PrM annotations

Of course, the view of design specifications introduced in sec. 7.2 contained no notions of “concurrency” or “synchronisation.” This was in line with the explicit aim of making the resulting basic framework of chapters 7 and 8 as general as possible. When considering distributed systems, however, these issues are the main concern. Hence this chapter will consider PrM specifications directly. This means that the GSPN generation method presented here is only applicable to the PrM specification language introduced in sec. 3.3. However, the aggregation based approach and the concept of subspecifications are general results with applicability beyond the scope of the PrM language.

**Example**

In case of the Accounting system specification of fig. 3.6, it is assumed that the Clerks will get 8 balances for 100 update balances, giving XOR output selection factors of 0.074 and 0.926 for the Present menu process. In the former case, on average 1.13 balances are requested. The Update balances process produces on average 1.24 New balances and notifies the accountant for 0.58% of the executions.

The corresponding “branching probabilities” for the PrM specification of fig. 3.6 are shown in table 11.10. These probabilities are used to parameterise conflict sets in the generated GSPN model of the distributed application.

The PrM annotations of this and other tables of this chapter will be thoroughly introduced and explained in part IV. □
Figure 11.11: GSPN subtemplates for binary AND (a) and XOR (b), as well as REP (c)
input connectives, and for binary AND (d) and XOR (e), as well as COND (f) and REP (g)
output connectives.

Figure 11.12: The “short-hand” output REP connective template of fig. 11.11 (a) and the
same template in expanded form (b).

11.4.4 GSPN generation

The top-level subspecification is analysed with

1. a set of customer classes corresponding to the operations provided by the top-level
subspecification,\(^1\) and

2. a GSPN model automatically derived from the top-level subspecification itself, and
   annotated with the aggregates for its subordinate subspecifications.

\(^1\) Of course, these operations correspond to the operations provided by the PrM specification itself.
The set of customer classes (1) corresponds to the set of operations provided by the top-level processes of the top-level subspecification. The GSPN model (2) is generated as follows:

1. a stochastically timed transition and corresponding input place is generated for each bottom-level flake process of the subspecification to represent process execution time;
2. an input template of immediate transitions with corresponding input places is generated for each primitive process of the subspecification to represent process input connectives;
3. an output template of immediate transitions with corresponding input places is generated for each primitive process of the subspecification to represent process output connectives;
4. a workload template is generated for every operation provided by a top-level process of the subspecification to represent the organisation using the projected distributed software system, and
5. a vector of $n$ token colours is generated for each customer class, where $n$ is the number of operations provided by the primitive processes of the subspecification, to represent how the customer class uses these operations.

The input and output templates for the timed transition are generated from the possibly composite connectives of the corresponding primitive process of the subspecification. The

---

2 Again, these operations correspond to the operations provided by the PrM specification itself.
GSPN subtemplates for binary AND (a) and XOR (b), as well as REP (c) input connectives, and for binary AND (d) and XOR (e), as well as COND (f) and REP (g) output connectives are shown in fig. 11.11. In case of primitive, non-REP and non-COND input or output connectives, these subtemplates become the GSPN templates for the input or output connective. In case of composite, REP-ed or COND-ed connectives, the GSPN template is constructed from the subtemplates outside-in. However, no input or output templates are needed for non-triggering flows.

**Liveness and boundedness** The REP output connective template of fig. 11.11 deserves particular attention. For the generated GSPN model to be analytically tractable, it must be both *live* and *bounded*.

**Liveness** means that the GSPN model must have no reachable absorbing states. Examples of such states are deadlock states, as well as states where all tokens have been consumed.

The GSPN generation algorithm of this chapter does *not* guarantee liveness. Instead, liveness is regarded as a functional requirement that the PrM specification must satisfy for the analysis technique presented here to be applicable. Nevertheless, the GSPN generation algorithm of this chapter may be useful for verifying the liveness of a PrM specification, through verification of the associated Petri-net.

Also, the non-zero probability that all tokens may be consumed is not a problem when the GSPN models are solved analytically, although this represents a problem for simulation.

**Boundedness** means that there must be a maximum number of tokens of each colour that can exist simultaneously in the net. This requirement is necessary for the associated state-space to be finite. If the generated GSPN model is not bounded is must be solved by simulation, which conflicts the analytic approach of the thesis. This explains the “awkward” output REP connective template of fig. 11.11.

**The REP output connective template** A REP output connective of a PrM process specifies that the process produces several items on the corresponding flow per process execution. This number of items is specified in PrM as an average repetition factor. However, no maximum number of items per process execution is provided. This is required for the corresponding GSPN model to become bounded. Therefore, let each PrM output REP connective now instead be annotated with an average repetition factor $rf$, as well as a maximum number of repetitions $n$.

The REP output connective template designed to represent this is depicted in fig. 11.12a. Each time the connective template is activated through a token in its leftmost place, two things may happen:

- with probability $1/(rf + 1)$ the token is consumed by the top transition and repetition is terminated, or
- with probability $rf/(rf + 1)$ the token is consumed by the bottom transition. In this case, a token is produced by the connective template, and the template reactivates itself.
However, the template has also been provided with a “control box” to control that the maximum number of repetitions is not exceeded:

- When the top transition fires and terminates repetition, a “termination” token is sent to the control box.
- Before the bottom transition can fire, it must receive a “permission” token from the control box.
- After the bottom transition has reactivated the REP template, a “new repetition” token is sent to the control box.

The control box must therefore count the number of “new repetitions” received so far in the current template activation. It must only provide “permission” tokens as long as the maximum number of repetitions has not been exceeded, and it must reset the repetition count each time a “termination” token is received.

The GSPN internals of this “control box” are shown in fig. 11.12b.

### 11.4.5 GSPN annotations

Each bottom-level flake process of the subspecification corresponds to a top-level process of one of its directly subordinate subspecifications. The stochastically timed transition representing execution of that process is annotated with the corresponding process aggregate. The service time of the transition will therefore depend on

1. the number of customers in its input place, and
2. the number of customers in the input places of transitions representing execution of other top-level processes of the same directly subordinate subspecification.

The input and output connective templates are annotated with the execution characterisations of the corresponding primitive process of the subspecification as follows:

1. the conflict set of an XOR output connective subtemplate is annotated with the choice factors of the corresponding XOR connective;
2. the conflict set of a REP input or output connective subtemplate is annotated geometrically based on the repetition factor of the corresponding REP connective, and
3. the conflict set of a COND output connective subtemplate (remember that triggering flows cannot be CONDed on input) is annotated based on the selection factor of the corresponding COND connective.

**Example**

This is also shown in fig. 11.11. □
The workload templates are generated from the organisation workloads of the top-level models. GSPN workload templates for terminal (a) and batch (b) workload intensities are shown in fig. 11.13. (The short-circuit templates (c) will be needed for intermediate subspecification analysis.) The timed transitions are annotated with

- the think time, $\theta$, for terminal usage intensities.

The corresponding transition is immediate for batch workload intensities. The initial marking of the GSPN model is defined through the initial number of tokens in the place of each workload template. The number of tokens equals the number of users, $n$, for every workload template representing a terminal of batch workload intensity. This is also shown in fig. 11.13.

All input places of input connective or workload templates correspond to consumption of items in the PrM specification. All output transitions of output connective or workload templates, with the exception of the sink transitions used in REP or COND subtemplates, correspond to production of items in the PrM specification. These input places and output transitions must therefore be connected in accordance with the PrM flows of the subspecification to represent flow of items. Furthermore, tokens must change color to represent the proper operations provided by each of the primitive processes of the subspecification.

Example

The GSPN model generated for the top-level PrM model of fig. 3.6 is shown in fig. 11.14. □

11.4.6 GSPN analysis

The performance measures to collect from top-level GSPN analysis are

1. the interarrival times, $\Delta t$, of the timed transition for each terminal workload template to represent the response time, $r$, of the corresponding operation, $r = n \Delta t - r$, and

2. the interarrival times, $\Delta t$, of the immediate transition for each batch workload template to represent the throughput, $\lambda$, of the corresponding operation, $\lambda = 1/\Delta t$.

The colour vectors representing customer classes must be disjoint for the response times and throughputs of each operation provided by the PrM specification to be derivable. For practical purposes however, the number of token colours may be reduced significantly because operations on different subordinate subspecifications need not be represented by different colors. This reduces the number of colors needed. Furthermore, the number of colours can be reduced because not every external operation provided by the specification uses all its lowest-level operations. The number of immediate transitions can be reduced along the same lines. In particular, strictly sequential immediate transitions may be eliminated, as well as places and transitions corresponding to non-triggering PrM flows.

Although the primitive processes of the top-level specification in the Accounting example are all contained in the same PrM top-level model, GSPN models can also be automatically generated from
1. processes of different PrM models;
2. not all the processes of a model, or
3. a combination of the two since the next-higher level PrM model specifies how external PrM flows cross submodel boundaries.

The only GSPN generation step affected is the above connection of output transitions of output connective and workload templates to input places of input connective and workload templates.

Figure 11.15: Subspecifications for the refined Accounting system design specification.

Figure 11.16: Generalised stochastic Petri-net model generated for the intermediate subspecification of fig. 11.15.
11.5 Intermediate Subspecification Aggregation

In case the distributed software specification has more than two levels of subspecification, two principal alternatives arise:

1. all levels above the primitive one can be ignored, combining all non-primitive subspecifications into a top-level subspecification and generating and analysing a corresponding top-level GSPN model, and

2. additional levels of aggregation can be introduced, generating and aggregating an “intermediate” GSPN model for each non-primitive subspecification not at the top-level.

The former alternative is likely to be more accurate. Unfortunately, the top-level GSPN model may generate an analytically intractable state space. If distribution occurs at a low level of software specification, analysis becomes expensive. Whether this approach is feasible, therefore depends on the level at which top-level GSPN analysis is needed.

The latter alternative is the more efficient because each intermediate GSPN model is likely to have much smaller state spaces than the combined, top-level one. Unfortunately, it may introduce prohibitive aggregation errors if neither the GSPN model being aggregated, nor the GSPN model to include the aggregate feature almost separability. Whether this approach is applicable, therefore depends on the degree of parallelism inherent in the intermediate subspecification. For subspecifications containing only unary triggering AND- and n-ary triggering XOR-connectives, the generated GSPN model is separable, and indeed a queueing network model for the subspecification could have been generated directly. Any deviation from this introduces aggregation errors in addition to those inherited from the short-circuit analysis of an almost separable performance submodel.

Since parallelism is likely to be lowest in lowest-level subspecifications, and the top-level GSPN models generated have acceptable sizes for highest-level subspecifications, a combination of the two may be useful. Lower-level intermediate subspecifications are aggregated in isolation and the higher-level intermediate subspecifications combined to form a top-level GSPN model of manageable size.

Example
Fig. 11.15 shows

1. the process hierarchy of a refined PrM specification for the Accounting software system,

2. the performance submodels constituting an alternative hardware platform for the PrM specification, as well as

3. the corresponding subspecification hierarchy, again assuming that all the performance submodels are really separable.

The Get balances process of fig. 3.6 has been decomposed into a submodel comprising three primitive processes. The Database manager has been replaced by a Database front-end minicomputer and a Database mainframe. The two computers communicate through a High-speed connection
Intermediate subspecification aggregation is carried out as a hybrid of the performance submodel aggregation of sec. 11.3 and the top-level GSPN model analysis of sec. 11.4:

- A GSPN model of the intermediate subspecification is generated and annotated as in sec. 11.4 (with minor exceptions that will be explained), and
- aggregated for all possible operation populations as in sec. 11.3.

The details have already been given in the corresponding sections. The only difference is that short-circuit workload intensity templates (fig. 11.13c) are now used rather than terminal and batch templates (fig. 11.13a&b).

**Example**

*Half* of the GSPN model generated for the intermediate subspecification of fig. 11.15 is shown in fig. 11.16. This half is generated from the Update balances submodel of fig. 3.9. The other half corresponds to the Get balances submodel. Since there are no triggering flows between the two corresponding processes in the next-higher level model of fig. 3.6, the two halves of the GSPN model are not connected. However, they do depend on one another, because their sojourn times depend on the marking of both halves. □

### 11.6 Software Synchronisation and Blocking

Sec. 8.3.7 pointed out the importance of synchronisation and blocking effects in non-distributed systems, while sec. 9.3.6 outlined a GSPN-based approach to dynamic modelling of target platform and existing application software systems. It was suggested that the aggregation technique of this chapter could be used for analysis of the resulting dynamic workload model. In one respect, analysis of a non-distributed system would be simpler, in that the corresponding hardware performance model would consist of a single performance submodel only. On the other hand, this chapter has only considered aggregation of partially dynamic design specifications. For solution of dynamic workload models, the concept of subspecifications would have to be generalised. The dynamic workload model can then be analysed using the aggregation scenario outlined in secs. 11.3, 11.4, and 11.5.

A similar strategy can be chosen for projected applications. Based on an annotated PrM specification of the application, the GSPN generation and annotation technique of secs. 11.4.4 and 11.4.5 can be used to establish a (set of) dynamic workload modules representing the projected, distributed application. This workload module can then be analysed together with other potentially dynamic workload modules according to the above.
Figure 11.17: An integration of the PPP and IMSE environments.
Part IV

A Realisation of the Framework
The previous part III presented a framework for performance engineering during information system development. To demonstrate and validate the practical applicability of the presented ideas, this part also presents a realisation of the framework in connection with the PPP and IMSE environments presented in secs. 3.3 and 4.8. Fig. 11.17 depicts how the PrM tool intends to integrate the PPP and IMSE environments of figs. 3.2 and 4.2. More specifically, the realisation presented is based on the PrM and sp specification languages.

Although this part is a repetition of work done in earlier parts of the thesis, it is nevertheless necessary both by making the framework itself more easily comprehensible, and by serving as a validation of the applicability of the framework. To further strengthen this validation, central parts of the realisation has been implemented in a working CASE tool for performance engineering of information systems. Also, the tool has been equipped with a user-interface supporting most of the techniques. Part IV will be exemplified with descriptions of this user-interface.

The integration of PrM and sp is based on:

1. extending the PrM language with performance annotations;
2. deriving a basic framework representation of the annotated PrM specification;
3. carrying out performance analyses inside the framework, and
4. presenting analysis results to information system developers in terms of the PrM specification.

Since the aim of this part is to demonstrate the applicability of the framework already presented rather than to present new ideas, the following chapters have been made far less formal than those of part III.

Chapter 12 integrates the PrM language with the basic framework of chapters 7 and 8, while chapter 13 outlines integration with the framework extensions of chapters 9, 10, and 11.
Chapter 12

Integrating the Basic Framework with the PrM CASE Tool

This chapter realises the basic framework of chapters 7 and 8 in terms of the PrM language of sec. 3.2 and the IMSE environment of sec. 4.2. Focus is put on the PrM language and the associated PrM tool developed as part of this thesis. All the PrM tool user-interfaces and analysis algorithms described in this chapter have been implemented. The tool has also been used in the case-study reported in [40, 39]. However, the interface with the IMSE QNET tool is not complete.

First, sec. 12.1 presents the PrM tool. Sec. 12.2 then extends the PrM language with the performance annotations necessitated by the basic framework. Finally, sec. 12.3 establishes mappings between the extended PrM language and the basic framework through the PrM tool.

12.1 The PrM Tool

A PrM tool has been developed to realise the basic framework of chapters 7 and 8 in terms of the PrM language of sec. 3.3.2. The tool supports the graphical PrM modelling approach supported by the PPP environment, and is integrated with the IMSE environment. The user-interface of the tool has been developed through the GSS/SDMF graphical support system of the IMSE [187], and it communicates with other IMSE tools through the object management system (OMS) [185]. The user-interface and inner workings of this tool will be used to illustrate the realisation of this part IV.

Example
The Customer transaction specification of fig. 3.5 is drawn in the PrM tool in fig. 12.1. Processes, agents, and stores are selected from a pop-up menu and placed on the screen by clicking the mouse. Flows between processes, agents, and stores are also created using the mouse. Every process, agent, store, and flow has a menu activated by clicking the mouse, into which an entity name may be entered.

The Make withdrawal submodel of fig. 3.8 accordingly is drawn in the PrM tool in fig. 12.2. Submodels are created by clicking the process to be decomposed, and then selecting the Create submodel entry
Figure 12.1: The Customer transaction top-level model of fig. 3.5 drawn in the PrM tool.

of its pop-up menu.

In what follows, the PrM specification of figs. 12.1 and 12.2 will be extended with performance annotations through menus provided by this tool. □

12.2 Annotating PrM Specifications

The PrM paradigm provides a well-defined set of concepts for specifying the dynamic properties of projected applications during development. To derive workload models from PrM specifications, however, some additional, performance related PrM annotations are also needed. Three types of PrM annotations are sufficient:

Demand descriptions for each primitive process in the PrM specification estimate the average amount of computing resources needed by the PrM process per execution.

Execution characterisations for each model in the PrM specification estimate the average number of items consumed and produced by each PrM process per execution.
Figure 12.2: The Make withdraw submodel of fig. 3.8 drawn in the PrM tool.

**Organisation workloads** for the PrM specification estimate how often the surrounding organisation uses each of the operations provided by the PrM specification.

In line with the basic framework, care has been taken to select annotations that

1. are both necessary and sufficient for obtaining a basic framework representation of the PrM specification,
2. are consistent with the PrM language, i.e. that the annotations follow the application developer's view of the projected application,
3. are easy to estimate, i.e. that the annotations are quantities that can be estimated by examining the design specification closely, and
4. are possible to validate, i.e. that the PrM annotations estimated in one phase of development later can be compared with measurements or more accurate estimates.

Each of the three types of annotations are considered in separate. (An additional extension made to the PrM language was the concept of operations introduced in sec. 3.3.2.1.)

### 12.2.1 Organisation specification

As already mentioned, the PrM specification provides operations representing functions that the surrounding organisation will use. The **PrM organisation specification** specifies how often the surrounding organisation will use each of the operations provided. The PrM specification is therefore annotated with
Operation list \( O_{m_1} \) for top-level model \( m_1 = \text{Customer transaction} \):

\[
O_{m_1} = (o_1, o_2, o_3, o_4)
\]

\( o_1 = \text{Initiate} \) \quad \( o_2 = \text{Terminate} \)
\( o_3 = \text{Deposit} \) \quad \( o_4 = \text{Withdrawal} \)

Flow list \( F_{m_1} \) for operation list \( O_{m_1} \):

\[
F_{m_1} = (f_{o_1}, f_{o_2}, f_{o_3}, f_{o_4})
\]

\( f_{o_1} = \text{Initiate} \) \quad \( f_{o_2} = \text{Terminate} \)
\( f_{o_3} = \text{Deposit} \) \quad \( f_{o_4} = \text{Withdrawal} \)

Table 12.3: Operation list for the PrM model of fig. 12.1.

Organisation workload \( \Omega_{m_1} \) on operation list \( O_{m_1} \):

\[
\Omega_{m_1} = (\omega_{o_1}, \omega_{o_2}, \omega_{o_3}, \omega_{o_4})
\]

\( \omega_{o_1} \): type is transaction \quad \( \lambda_{o_1} = 0.162 \)
\( \omega_{o_2} \): type is transaction \quad \( \lambda_{o_2} = 0.054 \)
\( \omega_{o_3} \): type is transaction \quad \( \lambda_{o_3} = 2.16 \)
\( \omega_{o_4} \): type is transaction \quad \( \lambda_{o_4} = 27.0 \)

Table 12.4: Organisation workload for the PrM model of fig. 12.1.

1. a PrM operation list for each top-level model, defining a PrM flow list for each of the external operations it provides, specifying the set of external triggering flows invoking the operation on the model, and

2. a PrM organisation workload also for each top-level PrM model, estimating the PrM workload intensity for each of the operations it provides, which is either transaction, terminal, or batch, with the corresponding parameters \( \lambda, (n, \theta) \) and \( n \) in accordance with sec. 8.3.1. Annotations \( \lambda, n, \) and \( \theta \) still represent transaction operation arrival rates, numbers of terminal and batch operations, and terminal operation think times, respectively.

The organisation workload is specified only once for each top-level PrM model. In this case, all the top-level models must be annotated.

12.2.1.1 The operation lists

The operations of a top-level PrM model are specified in an operation list. Each operation in this list has a PrM operation name and a flow list. The flow list contains the names of the flows that cooperatively trigger the operation. All the flows in this flow list must be triggering and external input flows to the PrM model. For each flow in the list, there must
12.2. ANNOTATING PRM SPECIFICATIONS

Figure 12.5: The operations provided by the PrM model of fig. 12.1.

Figure 12.6: The organisation workload on the PrM model of fig. 12.1.

also be a flow count, specifying the average number of items on each flow that are needed and are consumed per execution of the operation. In most cases, this count equals one.

Operation lists correspond closely to the entry counts of the basic framework, as will be demonstrated in sec. 12.3.1
Figure 12.7: The operations provided by the Make withdrawal submodel of fig. 12.2.

Figure 12.8: The operations provided by a submodel of the Accept withdrawal process of fig. 12.1. (Such a submodel does not exist at the current point of development.)

Example
The operation list for the top-level PrM model of fig. 12.1 is shown in table 12.3. The model provides four operations each corresponding to one of the four external triggering input flows to the model. In each case, the operation corresponds to exactly one occurrence of the corresponding flow. This is the usual case for top-level models. Other operations might correspond to more than one triggering flow, and more or less than one occurrence of each flow. □

12.2.1.2 The organisation workload

The organisation workload estimates capture knowledge of how often the organisation uses each of the operations provided by the PrM specification. In this thesis, this workload is represented as simply as possible. It is assumed that each operation represents a set of similar functions, and that the operation uses each follow a distinct, quantifiable pattern. Hence,
each operation can be described by a workload intensity in accordance with sec. 7.3.

Example
The organisation workload for the Customer transaction specification is shown in table 12.4 in terms of the operation list of table 12.3. □

12.2.1.3 Tool support

In the PrM tool, organisation specifications are entered in two steps, through menus for operation lists and organisation workloads.

Example
The PrM tool menu for specifying operations lists for top-level PrM models is shown in fig. 12.5. The menu for entering organisation workloads for each of these operations accordingly is shown in fig. 12.6. Only workload intensities have to be provided in this menu. The operation names are automatically filled in by the PrM tool. □

Although every PrM model has an operation list menu, operation lists only have to be supplied by the user at the top-level. For all PrM submodels, they are automatically generated by the PrM tool from the input connectives of the processes they decompose. Operation lists for models not at the top-level are nevertheless available to the PrM tool user for inspection.

Example
The (read-only) operation list menu generated for the Make withdrawal submodel of fig. 12.2 is shown in fig. 12.7. This submodel provides only a single operation corresponding to the single operation of its superior process of fig. 12.1.

The (read-only) operation list menu generated for the submodel of the Write statement process of fig. 12.1 accordingly is shown in fig. 12.8. This submodel provides one operation for each of the four XORed triggering inputs to its superior process. □

<table>
<thead>
<tr>
<th>Connectives $C_{p_a}$ for process $p_a =$ Make withdrawal:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_{p_a} = (C_{i_4}^{p_a}, C_{o_4}^{p_a})$</td>
</tr>
<tr>
<td>$C_{i_4}^{p_a} = (AND (T f_4) f_{10} (COND 0.1 (REP 3.0 f_{11})))$</td>
</tr>
<tr>
<td>$f_4 = $ Withdrawal</td>
</tr>
<tr>
<td>$f_{10} = $ Old balance</td>
</tr>
<tr>
<td>$f_{11} = $ Additional balance</td>
</tr>
<tr>
<td>$C_{o_4}^{p_a} = (AND (T f_{12}) (COND 0.96 f_{16}))$</td>
</tr>
<tr>
<td>$f_{16} = $ Write wdrwl stmt</td>
</tr>
<tr>
<td>$f_{12} = $ New balance</td>
</tr>
</tbody>
</table>

Table 12.9: Execution characterisations for the Make withdrawal process of fig. 12.1 in textual form.
Figure 12.10: Execution characterisations for the Make withdrawal process of fig. 12.1 specified in the PrM tool.

### 12.2.2 Execution characterisation

The PrM execution characterisations capture knowledge about the average number of items produced and consumed per execution of a PrM model. Execution characterisations are used to derive the activity counts of the basic framework, as will be demonstrated in sec. 12.3.2.

More specifically, execution characterisations for each primitive process in the PrM specification estimate the average number of items it consumes and produces per execution. Hence execution characterisations need only be entered at the lowest level of the PrM specification as mentioned in sec. 8.1.2. The input and output connectives of the primitive process are annotated with

1. a choice factor for each connective or flow inner to some XOR output connective, estimating the average number of times the inner connective or flow is selected per selection of the connective;\(^1\)

2. a repetition factor for each REP input or output connective, estimating the average number of times its inner connectives and flows are repeated per selection of the connective, and

\(^1\)This means that one choice factor of every XOR output connective is redundant in principle.
3. a selection factor for each COND input or output connective, estimating the average number of times its inner connectives and flows are selected per selection of the connective.

However, no PrM annotations are needed for non-triggering flows.

Each choice factor must be in range \([0 \ldots 1]\). Of course, the sum of choice factors for an XOR connective must be 1. Although PrM execution characterisations are entered in terms of the process connectives, they do not have to be provided at the same time as the connectives.

**Example**

The PrM connectives and execution characterisations for the \texttt{Make withdrawal} process of fig. 12.1 are shown in table 12.9 in textual form. \(\square\)

### 12.2.2.1 Tool support

In the PrM tool, execution characterisations are entered through a pop-up menu provided for each primitive PrM process.
Example

The PrM tool menu specifying the input and output connectives of the Make withdrawal process of fig. 12.1 is shown in fig. 12.10. PrM execution characterisations are entered through the same menu. However, this does not mean that execution characterisations must be provided at the same time as the connectives.

The execution characterisations for the Accept withdrawal process of the PrM submodel of fig. 12.2 are depicted in fig. 12.11, accordingly. The two must necessarily match on the input side. Note the choice factors for the XOR output connective of this process.

The connectives depicted in figs. 12.1 and 12.2 have been automatically generated from similar menus by the PrM tool. □

<table>
<thead>
<tr>
<th>Non-distributed PrM platform $P$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P = ({M}, {(})$</td>
</tr>
<tr>
<td>$M = \text{Mainframe}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resources provided by $P$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_M = {r_1, r_2, r_3}$</td>
</tr>
<tr>
<td>$r_1 = \text{Execute instruction}$</td>
</tr>
<tr>
<td>$r_2 = \text{Access disc}$</td>
</tr>
<tr>
<td>$r_3 = \text{Transmit line}$</td>
</tr>
</tbody>
</table>

Table 12.12: Non-distributed hardware platform for the Customer transaction model of fig. 3.5.

<table>
<thead>
<tr>
<th>Resource demands $D_p$ for primitive process $p_4 = \text{Write statement}$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_p = \langle d_{P_1}^p, d_{P_2}^p, d_{P_3}^p \rangle$</td>
</tr>
<tr>
<td>$d_{P_1}^p = 10000$</td>
</tr>
<tr>
<td>$d_{P_2}^p = 0.5$</td>
</tr>
<tr>
<td>$d_{P_3}^p = 20$</td>
</tr>
</tbody>
</table>

Table 12.13: Visit counts of the Write statement process of fig. 12.1.

12.2.3 Demand description

The PrM resource demand estimates capture knowledge about the average amount of computing resources each primitive PrM process uses per execution. These estimates are used when deriving visit matrices in the basic framework.

The PrM specification is annotated with:
Figure 12.14: The PrM platform for the Customer transaction specification of fig.12.1.

1. a PrM platform for each top-level PrM model,\(^2\) representing the computer that the projected application will be implemented on, and

2. PrM visit counts for each primitive PrM process, estimating its average demand for each of the resources of the PrM platform.

The two will be treated in separate.

12.2.3.1 The PrM platform

The PrM platform is defined for the whole PrM specification in terms of (one of) its top-level models. A PrM platform may be either distributed or non-distributed. In either case, it is defined in terms of one or more virtual PrM machines. A virtual PrM machine is a set of PrM platform components that provide PrM platform operations to the application. Components are either processing, communication, memory access, or any combination thereof. Hence a PrM platform is more detailed than the basic framework platforms. This is because a PrM platform must be directly mappable onto an sp model. Although the basic framework platforms were inspired by sp, they did not provide all its power for reasons of simplicity.

Each virtual machine contains exactly one processing component, and any number of memory components. A non-distributed PrM platform consists of exactly one virtual machine and possibly one communication component, while a distributed one consists of more than one virtual machine and at least one communication component.

\(^2\)In principle, it is sufficient to annotate only one of the top-level models.
Figure 12.15: The hardware resource demands of the Write statement process of fig. 12.1.

Example
The non-distributed platform for the Customer transaction specification of fig. 12.1 is shown in table 12.12. The platform contains a single virtual machine and no communication services. This hardware platform is the extended version of the one given in table 8.3 of sec. 8.1.3. □

12.2.3.2 The resource demands

The PrM resource demands are defined for every primitive process of the PrM specification. The resource demands estimate the average demands for PrM platform resources per execution of that process.

Example
The resource demands for the Write statement process are specified in table 12.13 in terms of the platform of table 12.12. This is one of the simplest processes of fig. 12.1 from a resource demand point of view since it does not use the database. Therefore, it can be annotated at the top-level of PrM specification, while e.g. the Make withdrawal process is instead annotated in terms of its submodel of fig. 12.2. □
12.2.3.3 Tool support

In the PrM tool, demand descriptions are again entered in two steps. Pop-up menus are provided for all top-level PrM models for PrM platform definition, while primitive PrM processes have menus for entering resource demands in terms of this platform.

Example
The menu specifying the PrM platform for Customer transaction specification of fig. 12.1 is shown in fig. 12.14. The resource demands of the Write statement process are specified in terms of this platform in fig. 12.15. □

12.3 Interfacing the Basic Framework

From the PrM annotations of the previous sec. 12.2, entry, activity, and visit matrices of the basic framework can be derived. Each of them will be considered in separate.

12.3.1 Deriving entry matrices

Entry matrices of the basic framework are derived directly from the operation lists of sec. 12.2.1. In this derivation, PrM flow counts become vector elements while the associated flow names determine their position in the vector. All vector positions where the corresponding flows do not participate in triggering the operation are of course zero.

Example
The top-level PrM model of fig. 12.1 provides four operations to the organisation, as shown in fig. 12.5. The corresponding entry matrix becomes

\[
E_m = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0
\end{bmatrix},
\]

(12.1)

The Make withdrawal submodel of fig. 12.2 provides only one operation as depicted in fig. 12.7, i.e.

\[
E_{m_2} = [1 \ 0 \ 0].
\]

(12.2)

□

12.3.2 Deriving activity matrices

In the PrM realisation, activity counts are derived from the PrM execution characterisations provided for every primitive PrM model by the application developer. Deriving activity counts from PrM annotations proceeds in two steps:
• First, the concepts of production $p_f$ and consumption counts $c_f$ for every triggering flow $f$ in the PrM specification are defined,

• then the triggering flow is mapped onto a pair $(o, o')$ of basic framework operations so that $a_{o,o'} = p_f / c_f$.

The two steps will be treated in separate.

**Production and consumption counts**

First the concepts of production and consumption counts for flow $f$, $p_f$ and $c_f$, are defined for all triggering flows in the PrM specification. Let flow $f$ be an output flow from process $p$ and an input flow to process $p'$. The production and consumption counts are then derived from the annotated output connective for process $p$ and input connective for process $p'$ as follows:

1. For flow $f$ from process $p$ to process $p'$, initially set both $p_f = 1$ and $c_f = 1$.

2. For each output connective of process $p$ that flow $f$ is inside, modify $p_f$ as follows:
   
   (a) If the output connective is AND, do nothing.
   
   (b) If the output connective is XOR, multiply $p_f$ with the corresponding choice factor inside the connective.
   
   (c) If the output connective is COND, multiply $p_f$ with the selection factor of the output connective.
   
   (d) If the output connective is REP, multiply $p_f$ with the repetition factor of the output connective.

3. For each input connective of process $p'$ that flow $f$ is inside, modify $c_f$ as follows:

   (a) If the input connective is AND or XOR, do nothing.

   (b) If the input connective is REP, multiply $c_f$ with the repetition factor of the input connective. (Again remember that triggering input flows cannot be CONDed on input.)

In this way, the production and consumption counts come to represent the average numbers of items that are produced and consumed per execution of the corresponding processes.

**Activity counts**

Every triggering flow $f$ of a PrM model must participate in triggering an operation on its input process $p'$. This PrM process operation corresponds to an operation $o'$ of the basic framework. Furthermore, the PrM operations of the process $p$ from which the triggering flow $f$ is output also correspond to operations $o$ of the basic framework. Hence every triggering flow $f$ from process $p$ to process $p'$ corresponds to a set of activity counts $\{a_{o,o'}\}$, for basic framework
operations $o$ representing the PrM operations provided by process $p$. (This paragraph makes some subtle simplifications. The algorithm used in the PrM tool is slightly more sophisticated than this.)

The activity counts $a_{o,d}$ are derived from flow $f$ as

$$a_{o,d} = \frac{p_f}{c_f} \quad (12.3)$$

Note that the above derivation rule implies that $a_{o,d'} = a_{o',d'}$ for all operations $o$ and $d'$ which represent PrM operations provided by the same process $p$. This is because the PrM annotations of sec. 12.2 are in terms of PrM processes rather than in terms of individual operations on these processes. For other specification languages tailored to the basic framework, this does not have to be the case.

PrM execution characterisations revisited

Also note that only the production and consumption counts, $p_f$ and $c_f$, are used in the activity count derivation, and not the PrM execution characterisations themselves. This means that

- PrM models could have been annotated directly in terms of triggering flows rather than in terms of connectives;
- PrM execution characterisation that are not used in deriving the production or consumption counts for some triggering flow are redundant, and that
- deriving the activity counts of the basic framework does not rely on connectives at all.

In the PrM and IMSE realisation of the basic framework, it has nevertheless been chosen to enter execution characterisations in terms of connectives because it was believed to be simple and elegant, although the number of annotations needed was higher. Superfluous annotations can be used for validation.

Example

The PrM model of the banking example of fig. 12.1 has no triggering flows that are ANDed or REPed on input and none that are XORed, CONDed or REPed on output. This means that the corresponding activity matrix is easy to construct,

$$A_{m_1} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}. \quad (12.4)$$

Since the PrM submodel of fig. 12.2 contains an output XOR connective, its matrix formulation is a little more complicated,

$$A_{m_2} = \begin{bmatrix} 0 & 0.96 & 0.04 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}. \quad (12.5)$$
12.3.3 Deriving visit matrices

In the PrM realisation, demand descriptions have been provided by the application developer for all primitive PrM processes. These annotations were not operation dependent, resulting in a vector of resource demands $D_p$ for process $p$. This vector represents the resource demands for all PrM operations provided by the process. (Note that while primitive components of the basic framework can be annotated with different resource demands for each operation, this is not supported by this particular PrM tool realisation. Although supporting per operation resource demands would have been easy, it has been considered unnecessary since it increases the number of PrM annotations required by the tool. If per operation resource demands are vital, the PrM process can be decomposed prior to annotation.)

When the primitive process $p$ is represented as a primitive component $\gamma$ in the basic framework, its visit matrix therefore becomes

\[
V_{\gamma} = \begin{bmatrix}
D_p \\
\vdots \\
0 \\
D_p
\end{bmatrix}
\]  \hspace{1cm} (12.6)

Hence the row vectors of $V_{\gamma}$ representing the visit counts of each operation $o \in O_{\gamma}$ are just duplicates of the resource demand vector $D_p$ for PrM process $p$.

Example

The resource demand vector of process $p_5$ Write statement was defined in table 12.1 as

\[
D_{p_5} = (10000, 0.5, 20).
\]  \hspace{1cm} (12.7)

Since process $p_5$ only provides a single operation, the corresponding basic framework visit matrix becomes

\[
V_{\gamma_5} = \begin{bmatrix}
10000 & 0 & 0.5 \\
0 & 0 & 20
\end{bmatrix}.
\]  \hspace{1cm} (12.8)

\hfill \Box

12.3.4 Interfacing the IMSE

When a basic framework representation of the PrM specification has been derived according to the previous sec. 12.3, workload derivation can proceed as in sec. 8.2 of the basis framework. First, however, a PrM platform mapping must be provided to map the PrM platform resources of sec. 12.3.3 onto IMSE QNET model service centres. Hence the PrM platform mapping realises the platform mapping of sec. 8.3.5.

Example

The PrM tool menu for specifying PrM platform mappings for the Customer transaction specification is shown in fig. 12.16. This mapping maps the PrM platform of fig. 12.14 onto the IMSE QNET model of fig. 12.17. Since the platform resources are one-to-one with QNET service centres, this mapping is trivial. \hfill \Box
As soon as a PrM platform mapping has been specified, the PrM tool can automatically generate an sp module and corresponding module implementation to represent the design specification. Hence a workload module of the basic framework is realised in terms of an sp module and a corresponding sp module implementation. Additional sp modules can be created in the IMSE to represent existing applications, as in sec. 8.3.3 of the basic framework.
Example

The $sp$ module generated by the PrM tool from the platform mapping and operation lists of the top-level PrM model of fig. 12.1 is shown in fig. 12.18. This module is part of the $sp$ model for the bank example shown in fig. 4.3 of sec. 4.8.2.

(Part of) the corresponding $sp$ module implementation generated for the $sp$ module of fig. 12.18 is shown in fig. 12.19. □

Since the $sp$ representation of the projected application becomes a single, non-distributed $sp$ module, the use of $sp$ may seem redundant in this realisation. It has nevertheless been included for three reasons:

- As mentioned already, additional $sp$ modules are needed to represent other, already existing applications as in the basic framework.

- $sp$ is needed for software platform modelling, as will be outlined in sec. 13.3 of the next chapter 13.

- The $sp$ representation becomes a single module only for non-distributed applications. Therefore, $sp$ is needed to for representation of distributed applications as will be pointed out in sec. 13.7.
Figure 12.18: The *sp* module generated by the PrM tool for the *Customer transaction* application.

Figure 12.19: The *sp* module implementation generated by the PrM tool for the *sp* module of fig. 12.18.
Chapter 13

Integrating the Framework Extensions with the PrM CASE Tool

This chapter will discuss how the basic framework extensions of chapters 9, 10, and 11 can be realised in terms of the PrM language of sec. 3.3.2. The user-interface of the PrM tool introduced in the previous chapter 12 will be used to illustrate this for some of the techniques. These user-interfaces have been implemented in the tool. In contrast to the previous chapter 12 however, the associated analysis algorithms have not been implemented or tried out.

Each of the basic framework extensions are considered in a dedicated section.

13.1 Parameter Validation

The parameter validation technique of sec. 9.1 was based on the linear algebra representation of the basic framework. Given a PrM process whose performance annotations have been invalidated by an annotated PrM process submodel, the internal parameter validation technique compares the resource demands derived from the submodel to the resource demands for the process entered through the menu of fig. 12.15. The external parameter validation technique, on the other hand, compares workload derivation results with measurements of the production code.

Internal parameter validation of a PrM specification requires

- an annotated PrM specification, and
- a set of now invalidated annotations of a non-primitive PrM process in the specification.

The following steps are undertaken:

1. derive a basic framework representation of the PrM specification;
2. carry out workload derivation according to sec. 8.2;
3. provide workload derivation results in terms of the PrM process, and
4. compare workload derivation results for the PrM process with the now invalidated annotations.

External parameter realisation has not been considered in the PrM tool realisation.

13.2 Residence Time Analysis

The residence time analysis technique of sec. 9.2 was based on the linear algebra formulation of the basic framework. Given a layer of PrM processes, the residence time analysis technique calculates the residence time of each PrM operation in the layer and determines a small number of PrM processes in the layer responsible for a large part of the overall response time. This section will outline how residence time analysis can be supported in a PrM realisation of the basic framework.

Residence time analysis of a PrM specification requires

- an annotated PrM specification, and
- a set of valid PrM platform sojourn times derived in an earlier IMSE QNET analysis.

The following steps are undertaken:

1. select process layer in the PrM specification;
2. derive a basic framework representation of the PrM process layer;
3. carry out residence time analysis as in sec. 9.2;
4. try to find appropriate set of $x\%$ of the processes, and
5. provide analysis results in terms of the PrM specification.

Analysis inputs and outputs will be considered in separate.

13.2.1 Analysis initiation

The residence time analysis is initiated by the PrM user, specifying:

- The type of analysis requested, which is either per operation or average.
  - In case of per operation analysis, the requested (top-level) operation(s) must also be given by the designer.
• The layer of PrM processes to be considered, which is either primitive layer, nth layer, or arbitrary layer.
  
  – In case of nth layer, the layer number $n$ must be provided, and
  – in case of arbitrary layer, the list of processes $p \in \mathcal{L}$ in the layer $\mathcal{L}$ must be provided.

This provides sufficient information for initiating a residence time analysis in accordance with sec. 9.2.

13.2.2 Analysis outputs

At the PrM level, the residence time analysis output is represented as

• a residence time for each PrM process in the layer;
• a response time percentage for each PrM process in the layer, and
• a set of important processes in the layer.

13.3 Software Platform Modelling

The software platform modelling technique of sec. 9.1 was based on the linear algebra representation of the basic framework. Given a software platform model of the target platform and operating system software components, the additionally induced workload and overhead is implicitly calculated during composite workload model analysis. This section will present how software platform modelling is supported by the PrM and $sp$ realisation.

Software platform modelling in connection with a PrM specification requires

• a PrM specification annotated in terms of a software platform;

The following steps are undertaken:

1. derive an $sp$ module and module implementation for the projected application using the PrM tool;
2. create $sp$ modules and module implementations for all major existing application and target platform components;
3. build a composite $sp$ model from the projected and existing $sp$ components;
4. analyse the $sp$ model;
5. use $sp$ model analysis results for IMSE QNET model analysis, and
6. provide analysis results in terms of the PrM specification.

Analysis inputs and outputs will be considered in separate.
13.3.1 Analysis initiation

Inside the PrM tool, software platform modelling is initiated when the PrM user specifies a PrM software platform rather than a hardware platform. (Sec. 9.3.2 pointed out that a hardware platform was a set of services to be mapped onto a performance model, while a software platform was a set of operations to be mapped onto a software platform model.) As result, an sp module and module implementation is derived to represent the projected application. (As pointed out in sec. 7.3, a workload module of the basic framework corresponds to an sp module and a module implementation.)

Using the sp tool, sp modules and module implementations are created to represent the major existing software platform and operating system software components. Of course, additional sp modules may be created to represent existing applications. Also, a set of primitive sp modules must be established to represent IMSE QNET model service centres. Hence, these primitive modules correspond to the hardware platform of the composite workload model.

An sp model is now built from these sp modules, with application modules at the top-level, and QNET modules at the bottom. This sp model corresponds exactly to the composite workload model of the basic framework. This sp model is now analysed by the sp tool to provide a set of visit counts for performance model analysis inside the IMSE.

13.3.2 Analysis outputs

At the PrM level, performance analysis outputs are represented as in the basic realisation, but with software platform operation sojourn times replacing hardware platform service sojourn times. These software platform sojourn times are calculated inside the basic framework as in sec. 9.3.4.2.

13.3.3 Tool support

<table>
<thead>
<tr>
<th>Non-distributed PrM platform $P$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P = { {M}, {} }$</td>
</tr>
<tr>
<td>$M =$ Software platform</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operations provided by $P$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O_M = {a_1, a_2, a_3, a_4, a_5, a_6, a_7}$</td>
</tr>
<tr>
<td>$a_1 =$ Execute statement</td>
</tr>
<tr>
<td>$a_2 =$ Select tuple</td>
</tr>
<tr>
<td>$a_3 =$ Select tuples</td>
</tr>
<tr>
<td>$a_4 =$ Update</td>
</tr>
<tr>
<td>$a_5 =$ Insert/delete</td>
</tr>
<tr>
<td>$a_6 =$ Read screen</td>
</tr>
<tr>
<td>$a_7 =$ Write screen</td>
</tr>
</tbody>
</table>

Table 13.1: Non-distributed PrM platform for the Customer transaction model of fig. 12.1.
13.4. BOUNDS ANALYSIS

Figure 13.2: The software platform for the banking model.

Example
A non-distributed PrM platform containing a single virtual machine and no communication services is shown in table 13.1. This software platform is the extended version of the one given for the bank example in table 9.2 of sec. 9.3.2.

The PrM tool menu for specifying a software platform for the banking example is depicted in fig. 13.2. The new platform is constituted by the programming language COBOL, an SQL database management system, and a Screen handler, as in sec. 9.3.2. □

13.4 Bounds Analysis

The bounds analysis technique of sec. 9.4 was based on the linear algebra formulation of the basic framework. Given at least one pair of minimum and maximum — as opposed to average — parameter estimates, the bounds analysis technique provides bounds for the performance measures required. This section will outline how bounds analysis can be supported by the PrM tool.

Bounds analysis of a PrM specification requires

- a PrM specification annotated with (at least one) minimum and maximum parameter.
The following steps are undertaken:

1. derive a bounded basic framework representation of the PrM specification;
2. carry out bounds analysis as in sec. 9.4, and
3. provide analysis results in terms of the PrM specification.

Analysis inputs and outputs will be considered in separate.

13.4.1 Analysis initiation

Bounds analysis is not selected specifically by the PrM user. Instead, the PrM tool automatically switches to bounds analysis when the first pair of minimum and maximum parameters are detected during workload analysis. Prior to bounds analysis inside the basic framework, minimum and maximum entry, activity, and visit matrices must be derived from the annotated PrM specification. Derivation of these matrices closely resembles the derivation of sec. 12.3, and it will not be outlined in detail here.

13.4.2 Analysis outputs

At the PrM level, performance analysis outputs are represented as in the basic realisation of chapter 12, but with minimum and maximum bounds replacing average measures.

13.4.3 Tool support

In the PrM tool interface, every input parameter may be either a pair of bounds or an average estimate. Bounded parameters are entered by choosing the Bounds rather than the Average option for the parameter in question, as in fig. 13.3. In response, the tool queries the user for both a minimum and a maximum estimate, rather than just an average estimate as in fig. 12.11.

Example

The PrM tool interface for entering minimum and maximum as opposed to average execution characterisations for the Accept withdrawal process of fig. 12.2 is shown in fig. 13.3. □

The menus for organisation workloads and demand descriptions provide the same support for bounded parameters, as does the organisation workload menu of fig. 12.6.

Minimum and maximum sp module generation is initiated by selecting the Generate module operation of the background menu as in fig. 12.

If a bounded parameter annotation is encountered during sp module generation, the tool automatically detects the case as a bounds analysis, and proceeds by generating both a minimum
and a maximum \( sp \) module implementation. These minimum and maximum implementations correspond to the minimum and maximum workload modules of secs. 8.3.2 and 8.3.3.

Performance analysis is initiated in the IMSE from the operation menu of the IMSE performance model object. The IMSE QNET tool does currently not support bounds analysis, thus only mean-value analysis as used for bounds analysis in sec. 9.4.4.6 is available.

### 13.5 Parametric Analysis

The parametric analysis technique of sec. 9.5 was based on the linear algebra formulation of the basic framework. Given one or more FREE — as opposed to bound — performance parameters, the technique performs an algebraic workload derivation and performance analysis. This section will outline how parametric analysis can be supported by the PrM and IMSE realisation.

Parametric analysis of a PrM specification requires
• a PrM specification annotated with (at least one) FREE parameter.

The following steps are undertaken:

1. derive a parametric basic framework representation of the PrM specification;
2. carry out parametric workload derivation as in sec. 9.4.3;
3. bind the FREE parameters of the resulting parametric sp module;
4. carry out IMSE performance analysis as in the basic framework realisation, and
5. provide analysis results in terms of the PrM specification.

Analysis inputs and outputs will be considered in separate.

13.5.1 Analysis initiation

Like bounds analysis, parametric analysis is not selected specifically by the PrM user. Instead, the PrM tool automatically switches to parametric analysis when the first FREE parameter is detected. Capital letters have been used for the term FREE here. This is because “FREE” parameters is an important IMSE concept.

As in the basic framework, free and bounded parameters may occur in the same PrM specification, and bounded parameters may also be free, although this is probably not very useful. Prior to parametric analysis inside the basic framework, average or minimum and maximum entry, activity, and visit matrices must be derived from the parametric PrM specification. These derivations will not be considered in detail here.

As a result of workload derivation, the PrM tool will create an sp module implementation with FREE variables. These FREE variables must be bound prior to sp analysis as required by the sp tool. From that point on, the analysis is no longer parametric, and IMSE QNET analysis may proceed according to the basic framework realisation of chapter 12.

13.5.2 Analysis outputs

Since FREE parameters in the IMSE must be fixed prior to sp analysis, all the produced performance measures are non-parametric. At the PrM level, analysis outputs are therefore represented as in the basic framework realisation of chapter 12.

13.5.3 Tool support

In the PrM tool interface, every input parameter may be either bound or FREE. Parametric annotations are entered by choosing the Make value FREE operation for the parameter in question. As shown in fig. 13.4, a Value name must be entered when the parameter is left FREE. This name will identify the parameter when fixing its value later. An optional Info
text may also be provided. This text will further help identifying the value when fixing it later.

Example
The PrM tool menu for entering FREE execution characterisations for the Customer transaction specification is shown in fig. 13.4. □

The menus for organisation workloads and demand descriptions provide the same support for FREE parameters, as does the organisation workload menu of fig. 12.6.

13.6 Interfacing Sensitivity Analysis with PrM

The sensitivity analysis technique of chapter. 10 was based on the linear algebra formulation of the basic framework. Given a (set of) residence time(s) of a processes and a (set of) PrM annotation(s), the technique determines the sensitivity(-ies) of the residence time(s) with
respect to the annotation(s) through differentiation. This chapter will outline how sensitivity analysis can be supported by the PrM tool.

Sensitivity analysis of a PrM specification requires

- an annotated PrM specification.

The following steps are undertaken:

1. specify the residence time(s) to derive the sensitivity of;
2. specify the PrM annotations to derive the sensitivity with respect to;
3. derive a basic framework representation of the PrM specification;
4. carry out sensitivity analysis according to chapter 10, and
5. present the sensitivity analysis results in terms of the PrM specification.

13.6.1 Analysis initiation

The sensitivity analysis is initiated by the PrM user, specifying

- the process whose response time(s) is (are) to be analysed;
- the type of sensitivity analysis requested, which is either per operation or average. In case of per operation analysis, the requested operation(s) must also be given by the designer.
- the annotations of the PrM specification to determine sensitivities with respect to.

As in chapter 10, this chapter will focus on sensitivities of PrM process residence times with respect to PrM specification annotations. A range of other sensitivity analyses are possible, as outlined in figs. 10.1 and 10.4.

This provides sufficient information for a sensitivity analysis to be carried out as in chapter 10. However, the possibilities for sensitivity analysis of IMSE QNET models are limited at the moment.

13.6.2 Analysis outputs

After sensitivity analysis inside the basic framework, the derived sensitivities are mapped back into the PrM tool. Sensitivities with respect to each selected annotation is available for inspection by the PrM user as an additional field of that annotation.
Figure 13.5: PrM tool main menu for specifying distributed platforms. The distributed hardware platform for the Accounting application has two virtual machines and one communication component.

13.7 Interfacing Distributed Analysis with PrM

The distributed information system analysis technique of chapter 11 is based on the linear algebra representation of the basic framework, and on additional GSPN models of software parallelism. This section will demonstrate how analysis of distributed information systems can be supported by the PrM tool.

Distributed system analysis requires

- an annotated, distributed PrM specification, and
- a set of IMSE QNET submodels.

The following steps must be undertaken:

1. derive a basic framework representation of the distributed PrM specification;
2. derive an $sp$ module and module implementation for each of the subspecifications of the PrM specification;
3. derive an IMSE PNT model for each non-primitive subspecification;
4. derive an overall $sp$ model for the subspecification hierarchy;
5. carry out IMSE performance model aggregations and top-level analysis, and
6. present analysis results in terms of the PrM specification.

13.7.1 Analysis Initiation

As for bounds and parametric analysis, distributed analysis is not selected specifically by the PrM user. Instead, the PrM tool automatically switches to distributed analysis when a distributed PrM platform is detected. The PrM tool then determines the PrM subspecifications
Figure 13.6: PrM tool submenus for specifying virtual machines and components for the distributed platform of fig. 13.5.

of the PrM specification according to sec. 11.3. For each subspecification, the tool generates an sp module and corresponding module implementation according to sec. 8.2 of the basic framework. For each non-primitive subspecification, the tool also generates a GSPN model using the algorithm of sec. 11.4. From the subspecification hierarchy, the PrM tool also generates an overall sp model from these modules and module implementations. The primitive modules of this model must correspond to IMSE QNET submodels as in chapter 11.

13.7.2 Analysis outputs

At the PrM level, distributed analysis outputs are represented as in the basic realisation of chapter 12.

13.7.3 Tool support

In the PrM tool, distributed information systems are specified through distributed PrM platforms according to sec. 12.3.3. In addition, resource demands for primitive PrM processes are annotated in terms of this distributed platform.

Example

The PrM menu for specifying the distributed resource platform for the Accounting specification of fig. 3.6 is shown in fig. 13.5. Each virtual PrM machine and communication component of this platform is specified in submenus, as shown in fig. 13.6.
Figure 13.7: PrM tool menu interface for specifying process resource demands. The process only uses a (non-proper) subset of the virtual machines and communication components provided by the platform of fig. 13.5.

The resource demands of the Present menu process of fig. 3.6 are specified in terms of this platform in fig. 13.7. The process uses only a subset of the virtual machines and communication components provided. □

The subcomponent rule of sec. 11.3 is automatically ensured by the PrM tool: Each PrM submodel has its own resource platform which is automatically generated by the tool. This resource platform contains only virtual machines and communication components that are used by the PrM process the submodel decomposes.

The PrM platform mapping menu of fig. 12.16 also allows distributed platform mappings.
Chapter 14

Conclusions and Further Work

The final chapter of this thesis will summarise the achievements of the approach to performance engineering of information systems and offer some conclusions and suggestions for further work.

Sec. 14.1 first summarises the novelties, improvements, and extensions of this thesis compared to previous work in the field. Sec. 14.2 then discusses to what extent the approach has met the “Requirements for Performance Engineering of Information Systems” presented in chapter 6, while sec. 14.3 concludes the thesis by outlining some future directions for theoretical and practical work on performance engineering of information systems.

14.1 Achievements

This thesis has presented and realised a framework for performance engineering of information systems. The framework integrated information system engineering and computer performance evaluation at the conceptual, modelling, and tool levels. As pointed out in chapter 6, this was also the main objective of the thesis. The basic framework and its extensions are also improvements on the state of the art due to the generality and to the high degree of automation which is facilitated.

In addition, the thesis has provided:

- A motivation of the importance of work assessment in the computer science and informatics fields at the practical level.
- A thorough discussion of the need for performance engineering of information systems, incorporating both known and novel considerations.
- A well-structured set of requirements for performance engineering of information systems, continuously refined as this work has progressed.
- A conceptually clear, integrated view of the information system engineering and computer performance evaluation fields.
• An open-ended basic framework for performance engineering of information systems which can be tailored to support a wide range of tools and techniques both in the information system and performance evaluation fields.

• A hierarchical approach to information system modelling supporting development of complex systems, and integration with state-of-art development tools.

• A basic framework based on matrices so that the resulting software workload models were analytically tractable.

• An integration with capacity planning by encouraging exchange of performance and workload models between information system developers and capacity managers.

• An application of known parameter validation and 80%-20% analysis techniques in terms of the basic framework.

• A generalisation of the 80%-20% rule into an (1 - x)%-%x% principle.

• A novel technique of target platform modelling to simplify parameter estimation, and provide a closer integration with capacity management.

• An application of known bounds and parametric analysis techniques in terms of the basic framework.

• A novel technique for sensitivity analysis of software workload models.

• A general result for sensitivity analysis of combined software workload and hardware performance models.

• A general view of distributed information systems integrating information system development with performance evaluation techniques for distributed systems.

• A concept of subspecifications which determine where in a distributed design software performance analyses must be carried out.

• A method for generating GSPN models from annotated PrM specifications.

• A realisation of the basic framework in terms of a working CASE tool, the PrM tool.

• A set of PrM tool interfaces supporting the framework extensions.

• Some suggestions for further work on performance engineering of information systems.

14.2 Requirements Met

Chapter 6 presented a set of requirements for performance engineering of information systems. This section points out which requirements have been met and which have not been met by the work of this thesis. Each requirements will be considered in separate.
Problems with Software Performance Engineering

**Cost-efficiency** has been ensured by closely integrating the framework and its extensions with the related fields of information system development and capacity planning.

**Integration with information system engineering** has been supported through a view of information systems which is open-ended towards many different specification languages. Furthermore, as many performance parameters as possible are entered in terms of the design specification, rather than at the hardware modelling level.

**Integration with capacity management** has been encouraged through the separation of existing from projected workload modules. Also, the concept of platform mapping makes it easy to tailor capacity planning performance models to support software performance studies.

**Parameter capture support** has been provided through several alternative and sometimes cooperating techniques. However, work remains on parameter capture support for distributed information systems.

Each of these four main points will be considered in separate.

**Cost-efficiency**

**Little additional development effort** has been ensured by making the basic framework open-ended with respect to specification languages. This means that the design specification can be established in a language of choice, provided it satisfies a minimum set of criteria. The basic framework only requires that this design specification must be *annotated* with a set of required performance parameters. An additional “software performance model” of the projected application is not required.

The workload derivation technique presented automatically creates a workload module from this specification without user interference. Furthermore, the dynamic software performance models needed for analysis of distributed applications are automatically generated.

**Re-)Use of existing design specifications** is not directly supported by the thesis, as this has been considered a methodology and tool issue. However, the framework in its basic or extended forms can easily be tailored to support design specification reuse.

**Re-)Use of existing performance and workload models** is also not directly supported by the framework. However, it is clearly in line with the main idea of the thesis. Again, performance and workload model (re-)use is a methodology and tool issue which is well-supported in the IMSE realisation of this thesis.

**Little additional interpretation effort** has been ensured by providing performance measures that are interpretable in terms of the design specification, i.e. operation responsiveness and component sojourn times.

In addition, the operational approach to queueing analysis provides performance measures that correspond to observable quantities of the real computer.
Integration with Information System Engineering

Usefulness for novices in performance evaluation has been aimed at by allowing most parameters to be entered in terms of the design specification. The PrM tool goes one step further in this respect, by regarding the organisation workload also as an aspect of the design specification.

However, parameter estimation remains a specialised task. Collection and management of experience data for software performance engineering is a possible path for further research.

Meaningful information system level results have been provided through the focus on user-level responsivenesses. Such responsivenesses are provided in terms of operations of the design specification. Hence they are easily interpretable.

Usefulness throughout the information system life-cycle is ensured by making only a minimal set of requirements about the design specification. Indeed, the basic framework can be interfaced with several different information system specification languages aimed at different stages of development, e.g. conceptual modelling, structural design, and pseudo-level coding.

Rapid feedback to designers has been supported through selection of rapid, interactive analysis methods both at the software and hardware levels.

What-if queries can be answered by repeated performance analyses with modified design parameters. The focus on rapid design feedback makes this possible. Also, parametric analysis is supported when a large number of what-if queries are all about a single or a few parameters.

Performance optimisation has been supported through residence time and sensitivity analysis, pointing out which design parameters where in the design specification that should be optimised. Furthermore, sensitivity analysis provide quantitative trade-offs between how much a design parameter should be improved at the cost of degrading another one.

Integration with Capacity Management

Estimation of impact of new large applications on system performance is directly supported by the outputs of performance analysis in the basic framework, or from aggregate analysis of distributed information systems. Although the thesis has only considered a single projected application competing with several existing ones, this extension can easily be made.

Estimation of impact on performance of existing applications is also directly supported, since existing applications are represented in the basic framework and its extensions alongside the projected one(s).

Existing applications are taken into account through the provision of workload modules for existing applications. A possibility not surveyed in this thesis, is the introduction of
hand-made GSPN models of distributed *existing* applications alongside the projected one(s).

**Maintaining a model of the computer system infrastructure** is not directly supported by the basic framework or its extensions, as it has been regarded a tool issue. However, the realisation of the framework in terms of the IMSE could support this through the IMSE object management system (OMS).

**Support for balancing studies** is also not directly supported by the thesis, as it is a tool and methodology issue. Again, the basic framework is a useful starting point on top of which support for balancing studies can be built. It may be possible to tailor the IMSE environment for such studies.

**Parameter Capture Support**

**Both coarse and detailed models** are supported since projected workload modules can be derived both from coarse and detailed design specifications, and existing workload modules can be established through measurements at any level of detail. In addition, target platform modelling supports highly detailed *composite* workload models.

**Workload modelling alternatives** is supported through the concept of workload modules which may be derived from design specifications or established through measurements. Composite workload modelling is also supported. In addition, subspecifications of a distributed design specification can be represented as dynamic GSPN models.

**Performance parameter validation** has been outlined through internal and external validation techniques. However, numerous other possibilities exist.

**Identification of performance-critical parts of the design** has been provided through residence time and sensitivity analysis

**Bounds analysis** has been provided for the framework, both at the software and hardware levels with respect to all possible parameter types. Also, bounds analysis has been outlined for *distributed* information systems.

**Parametric analysis** has been provided for the basic framework. However, parametric analysis of closed queueing networks was pointed out as a difficulty. Also, parametric analysis of distributed information systems is a topic for further research.

**Sensitivity analysis** has been provided for the basic framework, both at the software and hardware levels. This represents a major novelty of the thesis. Also, most of the additional analyses outlined in figs. 10.1 and 10.4 have been outlined. However, sensitivity analysis of *distributed* applications needs more concern.

In summary, *most* if not all of the requirements have been met by the basic framework and its extensions, and possibilities exist for further enhancements, as will be pointed out in the next sec. 14.3.
14.3 Further Work

This section concludes the thesis by outlining some future directions for theoretical and practical work on performance engineering of information systems. The four chapters of part III will be considered in separate, before discussing further work of the PrM realisation of part IV.

14.3.1 Basic framework

To a large extent, the basic framework of chapters 7 and 8 is finished in its present form. However, more realisations in terms of existing information system specification and performance modelling languages are needed to verify its generality.

At the organisation level, the concept of workload scenarios should be studied further, and alternative ways of specifying workload intensities sought for. Dynamic organisation modelling is one possible path to follow (e.g. [15]).

At the application modelling level, object-oriented realisations should be considered. Also, the process of collecting performance parameters as part of the system design process has not been considered in the thesis as it is the domain of a related doctoral work [41].

At the hardware level, experience is needed with non-separable queueing networks, as well as GSPN and simulation models. Queuing network representations of synchronisation and blocking should also be studied (e.g. [167, 119]). However, most of these techniques are approximate, as was pointed out in sec. 8.3.7. A possibility is therefore to introduce statistical distributions and confidence intervals into the framework. However, this would represent a deviation from the goal of simplicity which has been supported by the operational view of queueing network analysis in the thesis. An alternative path to follow, is therefore the provision of error bounds, both for performance parameters input, and for the performance measures produced.

Finally, methodology issues have been deliberately left out in this thesis to make the results more widely applicable.

14.3.2 Parameter capture support

The three main areas where further work is immediately suggested on the parameter support techniques are

1. PrM tool support for the techniques;
2. case study evaluation, and
3. providing parameter capture support for distributed information systems.

Regarding the first point, the PrM tool provides interfaces to most of the parameter capture support techniques presented in chapter 13. Implementing the associated analysis techniques is assumed to be easy. The second area is partially covered by the practical case study reported in [40, 39]. The third area however, represents a challenge for further research.
As mentioned in sec. 9.3.6, another path for further work is dynamic modelling of target platform software systems and existing application software. Generalised stochastic Petri-nets would be a natural choice of modelling approach. The resulting workload model would include both static and dynamic workload modules, and could be analysed by adaptation of the aggregation methods presented in chapter 11, or by application of approximate algorithms (e.g. Rolfa & Sevcik [156]).

14.3.3 Sensitivity analysis

The sensitivity analysis techniques of chapter 10 are also finished to a large extent. However, not all the possibilities outlined in figs. 10.1 and 10.4 have been worked out in detail. Also queueing networks with synchronisation and blocking must be surveyed. This might require derivates from solution techniques other than standard MVA.

Also, tool support for sensitivity analysis is required.

Finally, sensitivity analysis results are only presented for non-distributed information systems. In particular, sensitivity analysis of dynamic software performance models must be provided. Since the software performance analysis models of this thesis are generalised stochastic Petri-nets, reported Markov-chain sensitivity analysis results (e.g. [30, 77]) are usable. However, these techniques still need to be integrated with the results of chapters 10 and 11.

14.3.4 Distributed information systems

The distributed information system analysis techniques of chapter 11 should be extended to include estimates of aggregation errors. This can either be done through confidence intervals, or through error bounds. In either case, similar extensions must be made to the basic framework.

In addition, tool support and case study evaluation is needed.

Distributed software specifications with more than two levels of subspecifications is also a topic for further research.

14.3.5 The PrM tool

The PrM tool improves on state-of-art experimental tools in the areas focused on. It gains on its advanced user-interface and high degree of automation. Its most important advancement, however, is integrating recent advances in performance evaluation with an integrated CASE tool for software development, making both the IMSE and PPP toolsets available to the software performance engineer.

Only the basic framework, and none of the extensions of chapters 9, 10, and 11, have been implemented in the tool. However, its user-interface has been designed for including these additional analysis techniques at a later stage.
Appendices
Appendix A

List of Contributions

This appendix lists the most important achievements of the thesis, and points out clearly what has, and what has not been done by its author. The appendix has two sections: Sec. A.1 discusses the originality of the main parts of the thesis, and points out the main sources of inspiration. Sec. A.2 points out exactly which parts of the thesis that have been implemented and which parts that have been tried out in practice.

A.1 Statement of Originality and Sources

- **introduction, motivation, and requirements;**
  These discussions are partially based on the work of Smith [167], but many of the points are new. The first drafts of chapters 2 and 6 (e.g. [136, 137]) were written by the author alone. The motivation and requirements have since been developed further in collaboration with doctoral co-students, Vidar Vetland and Gunmar Brataas.

- **three-level view of information systems;**
  The three-level view of information systems (e.g. fig. 1.1) upon which the thesis is based is the author’s own idea. However, most other approaches to software performance engineering — e.g. Buzen [47] and Smith [167] — are built on similar ideas.

- **integrated baseline view;**
  This is the work of the author alone, although it is of course partially based on consensus in the information system and performance evaluation fields.

- **basic framework;**
  The performance parameters and workload derivation algorithm are made by the author, although eq. 5.1 from Lowe [127] and Oftedahl & Sølvberg [135] was an important starting point. However, the two latter references do not consider *hierarchical specification* which is a major concern in the thesis.

  The idea of combining projected and existing workload modules is taken from Hughes [103] and Smith [173], while the platform mapping concept is inspired by Beilner’s HIT [95]. Furthermore, the performance analysis algorithms are not original, but the integration of all these techniques into a single framework has been done by the author.
open-endedness and tailorability;
These are the author's own ideas, though quite common in other fields of computer science.

integration with capacity planning;
This idea has previously been pointed out by both Smith [167] and Berry [23]. However, the integration provided in this thesis is made "tighter" through the concept of target platform modelling.

application of 80\%-20\% analysis;
The 80\%-20\% analysis technique was introduced by Smith [167]. However, the generalisation into an \( (1 - x)\% - x\% \) principle is a new idea of this thesis.

target platform modelling;
The use of target platform models for software performance engineering is new in this thesis. The idea of static, composite workload models comes from Hughes' sp approach [103].
The idea of providing a closer integration with capacity management through target platform models established through measurements has been developed in cooperation with Vidar Vetland.

bounds analysis;
Software performance bounds were first presented by Smith [167]. However, the demonstration that minimum and maximum bounds are preserved in the workload derivation has been done by the author. At the performance model level, existing analysis techniques are applied.

parametric analysis;
Parametric analysis of software performance is again indebted to Smith [167]. The idea of parametric average and bounds analyses at the hardware level may however be new in this context.

sensitivity analysis;
The integrated software and hardware sensitivity analysis technique is a major contribution of the thesis. In particular, eq. 10.1 and its derivation is the author's own idea and work. Furthermore, this result is believed to be general.

hardware sensitivity analysis;
At the hardware level, algorithms are provided for operational sensitivity analysis of separable queueing networks. Although similar results exist for product-form networks (e.g. [126]) the presented algorithms are original.

software sensitivity analysis;
The software sensitivity analysis results are the author's own. However, the idea of sensitivity analysis for software performance engineering is due to Oftedahl & Solvberg [135]. Also, the derivation of sec. D.3.2 in the appendix is due to them, but again, hierarchic design specifications are not considered.

an analysis framework for distributed information systems;
The performance prediction framework for distributed systems is based on several ideas
from other authors, but the integration of those ideas is a contribution of the author. Also, significant parts of the framework are new.

- **the distributed platform concept;**
  The distributed platforms of chapter 11 are new, but have been heavily inspired by \textit{sp} \[103\].

- **subspecifications;**
  The concept of \textit{subspecifications} which determine where in a distributed design software performance analyses must be carried out is the author's own contribution. The corresponding definition may be generally applicable for performance prediction of distributed software using other specification and analysis methods.

- **GSPN-model generation;**
  The GSPN generation technique is the author's own contribution, partly based on his earlier work with Lindland \[122\].

- **aggregation;**
  The use of aggregation to solve hierarchical performance models is heavily inspired by Beilner's work on HIT \[17\]. The aggregation technique indicated is just an application of existing methods.

- **the PrM tool and case study;**
  A detailed description of the PrM tool and case study contributions will be given in the next sec. A.2.

- **suggestions for further work;**
  The suggestions for further work are the author's own, but they are of course inspired by general developments in the information system engineering and computer performance evaluation fields. Also, \[173\] points out several directions for the field of software performance engineering itself.

### A.2 Implementation and Case Study

- **tool support for the basic framework;**
  The PrM tool realisation of the basic framework has been implemented by the author. The tool incorporates a graphical user interface based on the IMSE UIS \[187\]. The mapping of annotated PrM specifications into design specifications is implemented according to chapter 12. Also, the workload derivation technique of sec. 8.2 has been implemented by the author.

- **integration with the IMSE;**
  The PrM tool generates a workload module and module implementation in \textit{sp} format. This has been done by the author. Hence workload model analysis is taken care of by the \textit{sp} tool \[129\], and interfacing with the performance model becomes an IMSE responsibility.
• **user-interface for parameter capture support;**
  The PrM tool provides user-interfaces for the parameter capture support techniques of chapter 9. However, the associated mapping and analysis algorithms have not been implemented.

• **use of the PrM tool;**
  The PrM tool has not been used by the author for practical case study purposes. Tool improvements have continuously been suggested by Gunnar Brataas who has been the primary PrM users throughout this work.

• **case study validation;**
  The case study reported e.g in [39] was initiated by the author. However, the study of existing software and hardware was later taken over and extended by Vidar Vetland [189], while the study of the projected application was initiated by Gunnar Brataas [40].
Appendix B

Operational Queueing Network Analysis

B.1 Open Queueing Network Analysis

Queueing networks whose operations all have transaction workload intensities are open. Such networks are analysed by straightforward application of the operational laws of sec. 4.4.2. Since the analysis is dependent on the set of arrival rates \( \lambda_0, \ldots, \lambda_n \) the performance measures derived in an open queueing network analysis become functions of the vector \( \bar{\lambda} \) of arrival rates, \( \bar{\lambda} = (\lambda_0, \ldots, \lambda_n) \).

Assuming flow balance, i.e. that the observation interval is sufficiently long, the throughput equals the (given) arrival rate for each operation \( o \),

\[
    x_o(\bar{\lambda}) = \lambda_o. \tag{B.1}
\]

Furthermore, the throughput of operation \( o \) at service centre \( k \) is given by the forced flow law,

\[
    x^o_k(\bar{\lambda}) = x_o(\bar{\lambda})v^o_k, \tag{B.2}
\]

and the total throughput at service centre \( k \) becomes

\[
    x_k(\bar{\lambda}) = \sum_{o \in O_S} x^o_k(\bar{\lambda}). \tag{B.3}
\]

The utilisation of operation \( o \) at service centre \( k \) is given by the utilisation law,

\[
    u^o_k(\bar{\lambda}) = x^o_k(\bar{\lambda})\tau_k, \tag{B.4}
\]

and the total utilisation at service centre \( k \) becomes

\[
    u_k(\bar{\lambda}) = \sum_{o \in O_S} u^o_k(\bar{\lambda}) = x_k(\bar{\lambda})\tau_k \tag{B.5}
\]

The sojourn time of operation \( o \) at service centre \( k \) is given by

\[
    z^o_k(\bar{\lambda}) = \begin{cases} 
    \tau_k & \text{(delay centres)} \\
    \tau_k[1 + \alpha^o_k(\bar{\lambda})] & \text{(queueing centres)}
\end{cases}, \tag{B.6}
\]
where \( \alpha^o_k(\bar{\lambda}) \) is the average number of operations already residing at service centre \( k \) when an operation \( o \) arrives. For delay centres eq. B.6 is obvious since no time is spent queueing for service and the time spent in the service centre therefore equals the service required. For FCFS queueing centres, the result can be explained as follows: The time spent in the service centre equals the sum of the service time, \( \tau_k \), and the time spent queueing while waiting for the \( \alpha^o_k \) customers before it in the queue to receive service, \( \tau_k \alpha^o_k \). (Remember that all operations at a service centre must have the same service time by definition in this thesis.) For preemptive-resume LCFS service centres, eq. B.6 is less obvious, but is nevertheless true [119].

In an open, separable queueing network, the number of customers already present at service centre \( k \) when a new customer enters, equals the average number of customers present at the centre. Therefore

\[
\alpha^o_k(\bar{\lambda}) = q_k(\bar{\lambda}).
\]  
\[
\text{(B.7)}
\]

According to Little’s law for flow balanced service centres (eq. 4.11), the queue length of operation \( o \) at service centre \( k \) is given by

\[
q^o_k(\bar{\lambda}) = x^o_k(\bar{\lambda}) z^o_k(\bar{\lambda}),
\]  
\[
\text{(B.8)}
\]

and the total queue length at service centre \( k \) becomes

\[
q_k(\bar{\lambda}) = \sum_{o \in O_s} q^o_k(\bar{\lambda}) = \sum_{o \in O_s} x^o_k(\bar{\lambda}) z^o_k(\bar{\lambda}).
\]  
\[
\text{(B.9)}
\]

By insertion of eqs. B.7 and B.8 into eq. B.9,

\[
z^o_k(\bar{\lambda}) = \tau_k [1 + \sum_{o' \in O_s} x^o_{k'} z^o_{k'}(\bar{\lambda})],
\]  
\[
\text{(B.10)}
\]

for queueing service centres. Since the bracketed expression of this equation is independent of operation \( o \),

\[
\frac{z^o_k(\bar{\lambda})}{\alpha^o_k(\bar{\lambda})} = \frac{\tau_k}{\tau_k} = 1,
\]  
\[
\text{(B.11)}
\]

so that

\[
z^o_k(\bar{\lambda}) = z^o_k(\bar{\lambda}) = z_k(\bar{\lambda})
\]  
\[
\text{(B.12)}
\]

for open queueing networks. In secs. B.2 and B.3 it will be demonstrated that this does not in general hold for closed and mixed queueing networks.

Insertion of eq. B.12 into eq. B.10 gives

\[
z_k(\bar{\lambda}) = \tau_k [1 + \sum_{o \in O_s} x^o_k(\bar{\lambda}) z_k(\bar{\lambda})], \text{ and}
\]  
\[
\text{(B.13)}
\]

\[
z_k(\bar{\lambda}) = \tau_k [1 + z_k(\bar{\lambda}) \sum_{o \in O_s} x^o_k(\bar{\lambda})].
\]  
\[
\text{(B.14)}
\]

Thus

\[
\tau_k = z_k(\bar{\lambda}) - \tau_k z_k(\bar{\lambda}) \sum_{o \in O_s} x^o_k(\bar{\lambda})
\]  
\[
\text{(B.15)}
\]

\[
\tau_k = z_k(\bar{\lambda}) - z_k(\bar{\lambda}) \sum_{o \in O_s} x^o_k(\bar{\lambda}) \tau_k
\]  
\[
\text{(B.16)}
\]
\[ z_k(\lambda) = z_k(\lambda) \left[ 1 - \sum_{o \in O_s} x_k^o(\lambda) \tau_k \right] \]  
(B.17)

\[ = z_k(\lambda) \left[ 1 - \sum_{o \in O_s} u_k^o(\lambda) \right] \]  
(B.18)

\[ = z_k(\lambda) \left[ 1 - u_k(\lambda) \right] \]  
(B.19)

\[ (\text{delay centres}) \]

so that

\[ z_k^o(\lambda) = \begin{cases} \frac{\tau_k}{1 - u_k(\lambda)} & \text{(delay centres)} \\ \frac{\tau_k}{1 - u_k(\lambda)} & \text{(queueing centres)} \end{cases} \]  
(B.21)

The queue length of operation \( o \) at service centre \( k \) now becomes

\[ q_k^o(\lambda) = x_k^o(\lambda) z_k^o(\lambda) \]  
(B.22)

\[ = \begin{cases} x_k^o(\lambda) \tau_k(\lambda) & \text{(delay centres)} \\ x_k^o(\lambda) \tau_k(\lambda) & \text{(queueing centres)} \end{cases} \]  
(B.23)

\[ = \begin{cases} u_k^o(\lambda) & \text{(delay centres)} \\ u_k^o(\lambda) & \text{(queueing centres)} \end{cases} \]  
(B.24)

and the total queue length at service centre \( k \) becomes

\[ q_k(\lambda) = \begin{cases} u_k(\lambda) & \text{(delay centres)} \\ u_k(\lambda) & \text{(queueing centres)} \end{cases} \]  
(B.25)

The response time for operation \( o \) is given by

\[ r_o(\lambda) = \sum_{k \in K_M} v_k^o z_k^o(\lambda), \]  
(B.26)

and the average number of operations \( o \) in system is given by

\[ q_o(\lambda) = \lambda_o(\lambda) r_o(\lambda). \]  
(B.27)

Table B.1, adapted from [119], summarises the resulting algorithm.

### B.2 Closed Queueing Network Analysis

Queueing networks whose operations all have terminal or batch workload intensities are closed. Such networks are analysed by recurrent application of the operational laws of sec. 4.4.2. Since the analysis is dependent on the set of operation populations \( n_0, \ldots, n_{o_n} \) and think times \( \theta_0, \ldots, \theta_{o_n} \) the performance measures derived in a closed queueing network analysis become functions of the vectors \( N \) of operation populations \( N = (n_0, \ldots, n_{o_n}) \), and \( \Theta \) of think times \( \Theta = (\theta_0, \ldots, \theta_{o_n}) \). (By definition, \( \theta_0 = 0 \) for batch operations \( o \).)

Again assuming flow balance, i.e. that the observation interval is sufficiently long, the throughput of operation \( o \) is given by the interactive response time law (eq. 4.16)

\[ x_o(N, \Theta) = \frac{n_o(N)}{\theta_o(\Theta) + r_o(N, \Theta)}, \]  
(B.28)
throughputs: \[ x_0(\bar{\lambda}) = \lambda_0 \]
\[ x_k^o(\bar{\lambda}) = x_0(\bar{\lambda})v_k^o \]
\[ x_k(\bar{\lambda}) = \sum_{o \in O_S} x_k^o(\bar{\lambda}) \]
utilisations: \[ u_k^o(\bar{\lambda}) = x_k^o(\bar{\lambda})\tau_k \]
\[ u_k(\bar{\lambda}) = \sum_{o \in O_S} u_k^o(\bar{\lambda}) = x_k(\bar{\lambda})\tau_k \]
sojourn times: \[ z_k(\bar{\lambda}) = \begin{cases} \tau_k & \text{(delay centres)} \\ \frac{\tau_k}{1 - u_k(\bar{\lambda})} & \text{(queueing centres)} \end{cases} \]
queue lengths: \[ q_k(\bar{\lambda}) = \begin{cases} \frac{u_k(\bar{\lambda})}{1 - u_k(\bar{\lambda})} & \text{(delay centres)} \end{cases} \]
response times: \[ r_0(\bar{\lambda}) = \sum_{k \in K_M} u_k^o(\bar{\lambda}) = r_0(\bar{\lambda}) \]
average number in system: \[ q_0(\bar{\lambda}) = \lambda_0(\bar{\lambda})r_0(\bar{\lambda}) \]

where the yet unknown responsiveness \( r_0(N, \Theta) \) for population \( N \) will be derived in eq. B.37. Furthermore, the throughput of operations \( o \) at service centre \( k \) is given by the forced flow law,
\[ x_k^o(N, \Theta) = x_0(N, \Theta)v_k^o, \] (B.29)
and the total throughput at service centre \( k \) becomes
\[ x_k(N, \Theta) = \sum_{o \in O_S} x_k^o(N, \Theta). \] (B.30)
The utilisation of operation \( o \) at service centre \( k \) is given by the utilisation law,
\[ u_k^o(N, \Theta) = x_k^o(N, \Theta)\tau_k, \] (B.31)
and the total utilisation at service centre \( k \) becomes
\[ u_k(N, \Theta) = \sum_{o \in O_S} u_k^o(N, \Theta) = x_k(N, \Theta)\tau_k. \] (B.32)
The sojourn time of operation \( o \) at service centre \( k \) is given by
\[ z_k^o(N, \Theta) = \begin{cases} \tau_k & \text{(delay centres)} \\ \frac{\tau_k}{1 + \alpha_k^o(N, \Theta)} & \text{(queueing centres)} \end{cases} \] (B.33)
where \( \alpha_k^o(\bar{\lambda}) \) is again the average number of operations already residing at service centre \( k \) when operation \( o \) arrives. The justification for these equations is as in the open queueing network analysis case.

For closed queueing networks, however, it is no longer true that the number of operations already present at a service centre when a new operation enters equals the average number of
operations present at the centre. This is due to the fact that when an operation is entering a service centre, that particular customer cannot already be present at the centre. Therefore, the number of operations already present at the centre will tend to be smaller than the average number present. Reiser & Lavenberg [154] have shown that for closed separable queueing networks,

$$q_k(N, \Theta) = \alpha_k(N, \Theta)$$

where $N_{-o}$ is the population vector with one less operation $o$ than in $N$. Thus the number of operations already present at a service centre when an operation $o$ enters equals the average number of operations present at the centre in a queueing network with one less operation $o$. This very important result, derived by Reiser & Lavenberg [154] is the key equation in the mean-value analysis (MVA) method. Consequently, the closed queueing network analysis technique for population vector $N$ requires prior analyses of the networks for operation population vectors $\{N_{-o}\}$, $o = 1, \ldots, O_S$.

According to Little’s law for flow-balanced service centres (eq. 4.11) the queue length of operation $o$ at service centre $k$ is now given by

$$q^o_k(N_{-o}, \Theta) = x^o_k(N_{-o}, \Theta)z^o_k(N_{-o}, \Theta),$$

and the total queue length at service centre $k$ becomes

$$q_k(N_{-o}, \Theta) = \sum_{o \in O_S} q^o_k(N_{-o}, \Theta) = \sum_{o \in O_S} x^o_k(N_{-o}, \Theta)z^o_k(N_{-o}, \Theta).$$

The right-hand side of eq. B.35 can be calculated from eqs. B.29 and B.33 with operation population $N_{-o}$ replacing $N$. This in turn requires analysis of yet more scarcerly populated networks. Thus closed queueing network analysis works recurrently from operation population $N$ down to the trivial case of an empty network with $N_0 = (0, \ldots, 0)$, for which $q_k^0 = 0$.

For a given operation population, the system response time for operation $o$ is furthermore given by

$$r_o(N, \Theta) = \sum_{k \in K_M} v^o_kz_k^o(N, \Theta),$$

and the average number of operations $o$ in system is given by

$$q_o(N, \Theta) = x_o(N, \Theta)r_o(N, \Theta).$$

For purposes of computational efficiency, forward application of eqs. B.28 to B.38 is preferable, starting with the empty network $N_0$ and analysing networks with growing populations in succession. Table B.2, adapted from [119] summarises the resulting algorithm.

This particular algorithm calculates the main performance measures of the queueing network. Of course, the inner for loop of this algorithm could have been extended with additional equations to compute other measures as well. Although the MVA analysis algorithm provides these measures for all populations $N' \leq N$, only the full-population measures for population $N$ are actually used in the basic framework. In chapter 11 however, additional measures for intermediate populations $N'$ will be used in aggregate analysis of distributed information systems.
Because of eq. B.34, the relation \( z_k^o(N, \Theta) = z_k^d(N, \Theta) \) — which was derived for open queueing networks in eq. B.12 — is not in general true for closed (and hence mixed) queueing networks. Therefore, the sojourn time vectors for performance models and hardware platforms should have been represented as matrices for such networks. The vector representations of \( Z_M \) and \( Z_P \) are therefore approximations only, as was pointed out in sec. 8.3.6.

\[
\begin{array}{l}
q_k(N_0, \Theta) \leftarrow 0, \quad k \in K_M \\
\text{for } n = 1 \text{ to } \sum_{o \in O_S} n_o(N_S) \text{ do begin} \\
\quad \text{for all } N' \text{ so that } \sum_{o \in O_S} n_o(N') = n \text{ do begin} \\
\hspace{1em} z_k^o(N', \Theta) = \left\{ \begin{array}{ll}
\tau_k & (\text{delay centres}) \\
\tau_k[1 + q^k(N_{-o}', \Theta)] & (\text{queueing centres}),
\end{array} \right. \\
\hspace{1em} r_o(N', \Theta) = \sum_{k \in K_M} v_k^o z_k^o(N', \Theta) & o \in O_S, \ k \in K_M \\
\hspace{1em} x_o(N', \Theta) = \frac{\lambda_o(\Theta) + r_o(N', \Theta)}{\lambda_o(\Theta) + r_o(N', \Theta)} & o \in O_S, \ k \in K_M \\
\hspace{1em} x_k^o(N', \Theta) = x_o(N', \Theta) v_k^o & o \in O_S, \ k \in K_M \\
\hspace{1em} q_k(N', \Theta) = \sum_{o \in O_S} x_k^o(N', \Theta) z_k^o(N', \Theta) & k \in K_M
\end{array}
\]

Table B.2: Closed mean value analysis technique.

### B.3 Mixed Queueing Network Analysis

Queueing networks whose operations are both transaction, as well as terminal and/or batch workload intensities are mixed. Such networks are analysed by a hybrid of the analysis techniques presented in the two last sections B.1 and B.2. Since the analysis is dependent on both

- the set of arrival rates \( \lambda_o \) for transaction operations \( o \);
- the set of numbers of users \( n_o \) for terminal and batch operations \( o \), and
- the set of think times \( \theta_o \) for terminal operations \( o \),

the performance measures derived in a closed queueing network analysis become functions of the vectors

- \( \tilde{\lambda} \) of operation arrival rates;
- \( N \) of numbers of users, and
- \( \Theta \) of operation think times.
Mixed queueing network analysis proceeds in three steps:

1. An open queueing network analysis is undertaken for the transaction operations to determine how much capacity they “consume” at each service centre.

2. A closed queueing network analysis is undertaken for the terminal and batch operations using “the remaining” service centre capacities.

3. Performance measures calculated for the transaction operations in step 1 are recalculated to compensate for the additional service centre capacities used by the terminal and batch operations.

The above step 1 is an application of eqs. B.1 to B.5 of sec. B.1, for all transaction operations \( o \). Again assuming flow balance, i.e. that the observation interval is sufficiently long,

\[
\begin{align*}
x_o(\lambda, N, \Theta) &= \lambda_0, \\
x_k^o(\lambda, N, \Theta) &= x_o(\lambda, N, \Theta)u_k^o, \\
x_k^s(\lambda, N, \Theta) &= \sum_{\Theta \in OS} x_k^o(\lambda, N, \Theta), \\
u_k^q(\lambda) &= x_k^q(\lambda, N, \Theta)\tau_k, \text{ and} \\
u_k^s(\lambda, N, \Theta) &= \sum_{\Theta \in OS} u_k^o(\lambda, N, \Theta) \\
&= x_k(\lambda, N, \Theta)\tau_k,
\end{align*}
\]

where the asterisks of eqs. B.41 and B.44 serve as reminders that the throughput \( x_k^s(\lambda, N, \Theta) \) and utilisation \( u_k^s(\lambda, N, \Theta) \) of service centre \( k \) only account for transaction operations \( o \). Eq. B.44 in particular is the basis for the above step 2.

Step 2 is initiated as soon as transaction operation utilisations \( u_k^s(\lambda, N, \Theta) \) have been derived for all service centres \( k \). Throughputs, queue lengths, and responsivities for all terminal and batch operations are now derived through load concealment \( [119] \). Analysis proceeds according to the algorithm of table B.2, with one important modification: The terminal and batch operations no longer have the full capacities of service centres \( K_M \) at their disposal. Fortunately, this situation can be accounted for through scaling of the service times \( \tau_k \),

\[
\tau_k^s(\lambda) = \frac{\tau_k}{1 - u_k^s(\lambda, N, \Theta)}.
\]

The scaled service times \( \tau_k^s(\lambda) \) represent the average time spent in service by a terminal or batch operation when \( u_k^s(\lambda, N, \Theta) \) of the service centre capacity is already used by transaction operations. Terminal and batch operations are now analysed with \( \tau_k^s(\lambda) \) replacing \( \tau_k \).

The throughputs, queue lengths, and responsivities output from this analysis provide the needed performance measures for the terminal and batch operations. The queue length \( q_k^s(\lambda, N, \Theta) \) is however not the total queue length at service centre \( k \), as it does not take transaction operations into account. It is however, as will soon be seen, important in determining these additional transaction operation queue lengths.
The utilisation of terminal or batch operation \( o \) at service centre \( k \) is finally given by the utilisation law,\(^1\)

\[
    u^o_k(\bar{\lambda}, N, \Theta) = x^o_k(\bar{\lambda}, N, \Theta) \tau_k. \tag{B.46}
\]

In step 3, the responsivenesses and queue lengths for transaction operations \( o \) are derived using the performance measures for the terminal and batch operations \( \mathcal{O}' = \mathcal{O}_S^T + 1, \ldots, \mathcal{O}_S \),

\[
z^o_k(\bar{\lambda}, N, \Theta) = \begin{cases} 
    \tau_k & \text{(delay centres)} \\
    \frac{\tau_k [1 + q^*_k(\bar{\lambda}, N, \Theta)]}{1 - u^*_k(\bar{\lambda}, N, \Theta)} & \text{(queueing centres)}
\end{cases}, \tag{B.47}
\]

\[
r_o(\bar{\lambda}, N, \Theta) = \sum_{k \in K_M} u^o_k z^o_k(\bar{\lambda}, N, \Theta), \text{ and} \tag{B.48}
\]

\[
q_o(\bar{\lambda}, N, \Theta) = \lambda_o(\bar{\lambda}) r_o(\bar{\lambda}, N, \Theta). \tag{B.49}
\]

Of course, the per operation throughputs and service centre utilisations calculated in eqs. B.39 to B.44 do still hold.

Finally, the the total throughput at service centre \( k \) becomes

\[
x_k(\bar{\lambda}, N, \Theta) = \sum_{o \in \mathcal{O}_S} x^o_k(\bar{\lambda}, N, \Theta), \tag{B.50}
\]

and the total utilisation of service centre \( k \),

\[
u_k(\bar{\lambda}, N, \Theta) = \sum_{o \in \mathcal{O}_S} u^o_k(\bar{\lambda}, N, \Theta) = x_k(\bar{\lambda}, N, \Theta) \tau_k. \tag{B.51}
\]

This concludes mixed queueing network analysis. The resulting algorithm is not summarised in a table.

---

\(^1\)Note that the algorithm of table B.2 does not calculate utilisations.
Appendix C

Queueing Network Bounds Analysis

As was the case for app. B, this is not in any way novel work of this thesis. However, it has been included because the notation used is somewhat unconventional, and for easy reference.

For notational convenience some additional parameters (already known from [119]) are introduced. The minimum and maximum service demand for operation $o$ at service centre $k$ are defined as

$$d_k^o- = v_k^o - \tau_k^-,$$

$$d_k^o+ = v_k^o + \tau_k^+,$$  \hspace{1cm} \text{(C.1)}

and the minimum and maximum total demand for operation $o$ as

$$d_o^- = \sum_{k \in K_M} d_k^o-,$$  \hspace{1cm} \text{(C.3)}

$$d_o^+ = \sum_{k \in K_M} d_k^o+,$$  \hspace{1cm} \text{(C.4)}

both for transaction, terminal, and batch operations $o$. Furthermore, the minimum and maximum service demand for operation $o$ are defined as

$$d_k^{o*-} = \max(d_k^o- | k \in K_M),$$  \hspace{1cm} \text{(C.5)}

$$d_k^{o*+} = \min(d_k^o+ | k \in K_M).$$  \hspace{1cm} \text{(C.6)}

Note that neither of these parameters are load dependent.

C.1 Open Queueing Networks

The key to bounds analysis of open queueing networks are eqs. B.2 and B.4 of sec. B.1, giving

$$u_k^o(\bar{\lambda}) = x_o(\bar{\lambda}) v_k^o \tau_k,$$ \hspace{1cm} \text{(C.7)}

now rewritten as

$$u_k^o(\bar{\lambda}) = x_o(\bar{\lambda}) d_k^o.$$ \hspace{1cm} \text{(C.8)}
Since a service centre utilisation must be less than one,

$$u^p_k(\bar{\lambda}) < 1,$$  \hspace{1cm} (C.9)

so that

$$x_o(\bar{\lambda}) < \frac{1}{d^o_k}.$$  \hspace{1cm} (C.10)

Thus the maximum throughput of transaction operation \( o \) becomes

$$\lambda^+_o = \frac{1}{d^o_k s^-}.$$  \hspace{1cm} (C.11)

The lower bound on response times is given by the case of no queueing for service. In this case, the responsiveness of operation \( o \) equals its minimum total demand,

$$r^-_o = d^-_o.$$  \hspace{1cm} (C.12)

However, there exists no upper bound, because queue lengths of the bottleneck service centre grows unlimited as \( \lambda_o(\bar{\lambda}) \) approaches \( \lambda^+_o \),

$$r^+_o = \infty.$$  \hspace{1cm} (C.13)

Lazowska et al. [119] therefore notes that these results are somewhat unsatisfactory. However, bounds analysis provide more useful results for closed queueing networks, and for terminal and batch operations of mixed networks. Hence tighter bounds are obtainable when transaction type workload intensities are approximated with terminal type intensities or “bounded” in the sense of sec. 8.3.1.

### C.2 Closed Queueing Networks

The throughput of terminal operation \( o \) was given by eqs. B.29 and B.31 of sec. B.2

$$u^o_k(N, \theta) = x_o(N, \theta) \bar{u}^o_k \bar{r}_k.$$  \hspace{1cm} (C.14)

It is now rewritten as

$$u^o_k(N, \theta) = x_o(N, \theta) d^o_k.$$  \hspace{1cm} (C.15)

again giving the maximum throughput of terminal operation \( o \),

$$x_o(N, \theta)^+ = \frac{1}{d^o_k s^-}.$$  \hspace{1cm} (C.16)

Under light load, however, the maximum throughput is bounded further for the case when no time is spent queueing for service. In this case, each of the \( n_o(N, \theta)^+ \) operations spend an average \( d^-_o \) of time demanding service and \( \theta_o(N, \theta)^- \) thinking, giving a throughput of

$$x_o(N, \theta)^+ = \frac{n_o(N, \theta)^+}{\theta_o(N, \theta)^- + d^-_o}.$$  \hspace{1cm} (C.17)
The minimum throughput occurs when each of the \( n_o(N, \Theta)^- \) operations have to wait for all the \( n_o(N, \Theta)^+ - 1 \) other operations at all the service centres before receiving service. In this case, each operation will spend an average amount \((n_o(N, \Theta)^+ - 1)d_o^+\) of time queuing for service, \(d_o^+\) in service, and \(\theta_o(N, \Theta)^+\) thinking. This gives a minimum throughput of

\[
x_o(N, \Theta)^- = \frac{n_o(N, \Theta)^-}{\theta_o(N, \Theta)^+ + n_o(N, \Theta)^+d_o^+}
\]  

(C.18)

The above can be summarised by the following throughput bounds for terminal operations

\[
\frac{n_o(N, \Theta)^-}{\theta_o(N, \Theta)^+ + n_o(N, \Theta)^+d_o^+} \leq x_o(N, \Theta) \leq \min\left(\frac{1}{d_o^+}, \frac{n_o(N, \Theta)^+}{\theta_o(N, \Theta)^- + d_o^-}\right),
\]  

(C.19)

and for batch operations (\(\theta_o(N, \Theta)^- = \theta_o(N, \Theta)^+ = 0\))

\[
\frac{1}{n_o(N, \Theta)^+d_o^-} \leq x_o(N, \Theta) \leq \min\left(\frac{1}{d_o^-}, \frac{n_o(N, \Theta)^+}{d_o^-}\right).
\]  

(C.20)

For some population \(n_o(N, \Theta)^*\), the light-population maximum bound \(\frac{n_o(N, \Theta)^*}{\theta_o(N, \Theta)^- + d_o^-}\) must be replaced by the heavier-population bound \(\frac{1}{d_o^+}\). This occurs where the two bounds are equal,

\[
\frac{1}{d_o^+} = \frac{n_o(N, \Theta)^*}{\theta_o(N, \Theta)^- + d_o^-}
\]  

(C.21)

so that

\[
n_o(N, \Theta)^* = \frac{\theta_o(N, \Theta)^- + d_o^-}{d_o^+}.
\]  

(C.22)

From these equations, response time, utilisation, and queue length bounds can be derived.

### C.3 Mixed Queueing Networks

Bounds for mixed queueing networks can be derived through application of the techniques of the two previous sections, C.1 and C.2 in combination. However, mixed queueing networks are also hampered by the problems of weakly bounded transaction operations.
Appendix D

Sensitivity Analysis

D.1 Performance Model Sensitivities with Respect to Service Times

The algorithm proceeds through differentiation of eqs. B.28 to B.38 of sec. B with respect to service times \( \tau_{k'} \). From differentiation of eq. B.33 we have

\[
\frac{\partial z_k^o(N_{+o}, \tau_{k'})}{\partial \tau_{k'}} = \begin{cases} \frac{\partial \tau_k}{\partial \tau_{k'}} & (\text{delay centres}) \\ \partial \left[ \tau_k[1 + q_k(N, \tau_{k'})] \right] / \partial \tau_{k'} & (\text{queueing centres}) \end{cases}
\]

\( (D.1) \)

\[
= \begin{cases} 0 & (k' \neq k, \text{ delay centres}) \\ 1 & (k' = k, \text{ delay centres}) \\ \tau_k \frac{\partial q_k(N, \tau_{k'})}{\partial \tau_{k'}} & (k' \neq k, \text{ queueing centres}) \\ 1 + q_k(N, \tau_{k'}) + \tau_k \frac{\partial q_k(N, \tau_{k'})}{\partial \tau_{k'}} & (k' = k, \text{ queueing centres}) \end{cases}
\]

\( (D.2) \)

Differentiating eq. B.36 gives

\[
\frac{\partial q_k(N, \tau_{k'})}{\partial \tau_{k'}} = \partial \left[ \sum_{o \in O_s} x_o(N, \tau_{k'}) z_k^o(N, \tau_{k'}) \right] / \partial \tau_{k'}
\]

\( (D.3) \)

\[
= \sum_{o \in O_s} \frac{\partial x_o(N, \tau_{k'})}{\partial \tau_{k'}} z_k^o(N, \tau_{k'}) + \sum_{o \in O_s} x_o(N, \tau_{k'}) \frac{\partial z_k^o(N, \tau_{k'})}{\partial \tau_{k'}}
\]

\( (D.4) \)

\[
\frac{\partial x_k^o(N, \tau_{k'})}{\partial \tau_{k'}} = \partial \left[ x_o(N, \tau_{k'}) v_k^o \right] / \partial \tau_{k'}
\]

\( (D.6) \)

\[
= \frac{\partial x_o(N, \tau_{k'})}{\partial \tau_{k'}} v_k^o
\]

\( (D.7) \)

where \( \frac{\partial q_k(N, \tau_{k'})}{\partial \tau_{k'}} \) can be derived by eq. B.33 with operation population \( N \) replacing \( N_{+o} \).

Differentiating eq. B.29 gives
while differentiating eq. B.28 in turn gives

\[
\frac{\partial x_o(N, \tau_{k'})}{\partial \tau_{k'}} = \partial \left[ \frac{n_o(N)}{[\theta_o + r_o(N, \tau_{k'})]} \right] / \partial \tau_{k'} \tag{D.8}
\]

\[
= - \frac{n_o(N) \frac{\partial r_o(N, \tau_{k'})}{\partial \tau_{k'}}}{[\theta_o + r_o(N, \tau_{k'})]^2}, \tag{D.9}
\]

the negation sign again being consistent with our intuition that throughputs decrease when visit counts increase. From eq. B.37 we obtain the final derivate

\[
\frac{\partial r_o(N, \tau_{k'})}{\partial \tau_{k'}} = \partial \left[ \sum_{k \in K_M} v_k^o z_k^o(N, \tau_{k'}) \right] / \partial \tau_{k'} \tag{D.10}
\]

\[
= \sum_{k \in K_M} v_k^o \frac{\partial z_k^o(N, \tau_{k'})}{\partial \tau_{k'}.} \tag{D.11}
\]

Recurrent application of eqs. D.1 to D.11 provides sensitivities for visit counts on all operation responsivenesses, as well as service centre sojourn times and utilizations. Again, a forward algorithm would be more computationally efficient.

## D.2 Performance Model Sensitivities with Respect to Think Times

The algorithm proceeds through differentiation of eqs. B.28 to B.38 of sec. B with respect to think time \( \theta_o \). From differentiation of eq. B.33 we have

\[
\frac{\partial z_k^o(N, \theta_o)}{\partial \theta_o} = \begin{cases} \frac{\partial \theta_o}{\partial \theta_o} & \text{(delay centres)} \\ \partial \left[ \tau_k [1 + q_k(N, \theta_o)] \right] / \partial \theta_o & \text{(queueing centres)} \end{cases} \tag{D.12}
\]

\[
= \begin{cases} 0 & \text{(delay centres)} \\ \tau_k \frac{\partial q_k(N, \theta_o)}{\partial \theta_o} & \text{(queueing centres)} \end{cases} \tag{D.13}
\]

\[
\frac{\partial q_k(N, \theta_o)}{\partial \theta_o} = \partial \left[ \sum_{o \in O_S} x_o(N, \theta_o) z_k^o(N, \theta_o) \right] / \partial \theta_o \tag{D.15}
\]

\[
= \sum_{o \in O_S} \frac{\partial x_o(N, \theta_o)}{\partial \theta_o} z_k^o(N, \theta_o) + \sum_{o \in O_S} x_o(N, \theta_o) \frac{\partial z_k^o(N, \theta_o)}{\partial \theta_o}, \tag{D.17}
\]

where \( \frac{\partial z_k^o(N, \theta_o)}{\partial \theta_o} \) can be derived by eq. B.33 with operation population \( N \) replacing \( N_{+o} \).
Differentiating eq. B.29 gives
\[
\frac{\partial r_o^s(N, \theta_o')}{\partial \theta_o' } = \frac{\partial [x_o(N, \theta_o') v^o_k]}{\partial \theta_o'} \quad (D.18)
\]
while differentiating eq. B.28 in turn gives
\[
\frac{\partial r_o(N, \theta_o')}{\partial \theta_o' } = \partial \left[ \frac{n_o(N)}{\theta_o + r_o(N, \theta_o') } \right] / \partial \theta_o' \quad (D.20)
\]
\[
\begin{cases}
- \frac{n_o(N)}{[\theta_o + r_o(N, \theta_o')]^2} \frac{\partial r_o(N, \theta_o')}{\partial \theta_o'} & (\theta' \neq \theta) \\
- \frac{n_o(N)}{[\theta_o + r_o(N, \theta_o')]^3} [1 + \frac{\partial r_o(N, \theta_o')}{\partial \theta_o'}] & (\theta' = \theta).
\end{cases}
\]  
(D.21)

Here, the negation sign is less obvious, as increased think times impact on throughputs in two ways:

- On one hand, the increase should decrease throughput because each user spends more time thinking.
- On the other hand, the increase should increase throughput because queue lengths of the service centres are reduced.

While the first of these contrary effects is accounted for by the minus sign, the latter is represented by the potentially negative term \( \frac{\partial r_o(N, \theta_o')}{\partial \theta_o'} \). In particular, if \( \theta' = \theta_o \) a derivative \( \frac{\partial r_o(N, \theta_o')}{\partial \theta_o'} < -1 \) will lead to a throughput increase. From eq. B.37 we obtain the final derive
\[
\frac{\partial r_o(N, \theta_o')}{\partial \theta_o' } = \partial \left[ \sum_{k \in K_M} v^o_k \phi_k^o(N, \theta_o') \right] / \partial \theta_o' \quad (D.22)
\]
\[
= \sum_{k \in K_M} v^o_k \frac{\partial \phi_k^o(N, \theta_o')}{\partial \theta_o' } . \quad (D.23)
\]

Recurrent application of eqs. D.12 to D.23 provides sensitivities for think times on all operation responsivenesses, as well as service centre sojourn times and utilizations. Again, a forward algorithm would be more computationally efficient.

### D.3 Design Specification Sensitivities for Non-primitive Components

Let \( X_{i,\gamma} \) be the \( i \)’th column vector and \( X_{i,\gamma} \) be the \( i \)’th row vector of any matrix \( X_\gamma \), then eq. 8.21
\[
V_\gamma = E_\gamma (I - A_\gamma)^{-1} V^*_\gamma
\]  
(D.24)
gives
\[
v_{0,\sigma} = (E_{0,\gamma})^T (I - A_\gamma)^{-1} V^*_{\sigma,\gamma}, \quad (D.25)
\]  
The two kinds of derivates possible will be treated in separate.
D.3.1 Analysis with respect to entry counts

The sensitivity with respect to entry counts becomes

\[
\frac{\partial v_{o,\sigma}(E_{o',\gamma})}{\partial E_{o',\gamma}} = (I - A_\gamma)^{-1}V_{o,\gamma},
\]

(D.26)

when \( o = o' \) and a vector of 0's otherwise.

D.3.2 Analysis with respect to activity counts

The sensitivity of \( v_{o,\sigma} \) on activity matrix \( A_\gamma \) becomes

\[
\frac{\partial v_{o,\sigma}(A_\gamma)}{\partial A_\gamma} = \frac{v_{o,\sigma}(A_\gamma + dA_\gamma) - v_{o,\sigma}(A_\gamma)}{dA_\gamma}.
\]

(D.27)

Oftedahl & Solvberg [135] gives

\[
\begin{align*}
v_{o,\sigma}(A_\gamma + dA_\gamma) &= (E_{o,\gamma})^T(I - A_\gamma - dA_\gamma)^{-1}V_{o,\gamma} \\
&= (E_{o,\gamma})^T[(I - A_\gamma)(I - A_\gamma)^{-1}dA_\gamma]^{-1}V_{o,\gamma} \\
&= (E_{o,\gamma})^T(I - (I - A_\gamma)^{-1}dA_\gamma)^{-1}(I - A_\gamma)^{-1}V_{o,\gamma},
\end{align*}
\]

Setting

\[
Q_\gamma = (I - A_\gamma)^{-1}
\]

for simplicity gives

\[
\begin{align*}
v_{o,\sigma}(A_\gamma + dA_\gamma) &= (E_{o,\gamma})^T(I - Q_\gamma dA_\gamma)^{-1}Q_\gamma V_{o,\gamma} \\
&= (E_{o,\gamma})^T[\sum_{i=0}^{\infty}(Q_\gamma dA_\gamma)^i]Q_\gamma V_{o,\gamma} \\
&= (E_{o,\gamma})^T[I + Q_\gamma dA_\gamma + (Q_\gamma dA_\gamma)^2 + \cdots]Q_\gamma V_{o,\gamma} \\
&= (E_{o,\gamma})^TQ_\gamma V_{o,\gamma} + (E_{o,\gamma})^TQ_\gamma dA_\gamma Q_\gamma V_{o,\gamma} \\
&= (E_{o,\gamma})^T(I - A_\gamma)^{-1}V_{o,\gamma} + (E_{o,\gamma})^TQ_\gamma dA_\gamma Q_\gamma V_{o,\gamma} \\
&= v_{o,\sigma}(A_\gamma) + (E_{o,\gamma})^TQ_\gamma dA_\gamma Q_\gamma V_{o,\gamma},
\end{align*}
\]

This means that

\[
\frac{v_{o,\sigma}(A_\gamma + dA_\gamma) - v_{o,\sigma}(A_\gamma)}{dA_\gamma} = \frac{(E_{o,\gamma})^TQ_\gamma dA_\gamma Q_\gamma V_{o,\gamma}}{dA_\gamma}.
\]

\[
= [(E_{o,\gamma})^TQ_\gamma]T[I - V_{o,\gamma}]T \\
= (Q_\gamma)^T(E_{o,\gamma}(V_{o,\gamma})^T(Q_\gamma))^T \\
= [(I - A_\gamma)^{-1}]^T(E_{o,\gamma}(V_{o,\gamma})^T(I - A_\gamma)^{-1})].
\]
List of Figures

1.1 The three levels of computing in an information system. .......................... 6

3.1 An integrated information systems engineering environment. .................. 34

3.2 A schematic overview of the PPP environment for information system engi-
neering. ........................................................................................................... 36

3.3 The PPP information system engineering strategy. ................................. 36

3.4 PrM constructs: process (a), store (b), agent (c), and flow (d). ............... 38

3.5 Top-level PrM model for an Customer transaction processing system. ..... 39

3.6 Top-level PrM model for an Accounting system. ................................... 40

3.7 PrM connectives: AND and XOR (a), and REP, COND, REPCOND, and
CONDREP (b). .............................................................................................. 41

3.8 PrM submodel for the Make withdrawal process of fig. 3.5. ..................... 43

3.9 PrM submodel for the Update balances process of fig. 3.6. ....................... 43

3.10 PrM submodel for the Get balances process of fig. 3.6. ......................... 44

3.11 Overview of the Customer transaction PrM specification of figs. 3.5 and 3.8
with processes and PrM models. ................................................................. 45

3.12 Overview of the Accounting PrM specification of figs. 3.6, 3.9, and 3.10 with
processes and PrM models. ................................................................. 45

4.1 A computer performance evaluation environment. .................................. 76

4.2 The open-ended IMSE architecture. ......................................................... 80

4.3 An sp model for the bank example. ......................................................... 80

5.1 Three major breakthroughs in the development of software performance engi-
neering (SPE): Analytic hardware performance models (a), analytic software
performance models (b), and analytic software and hardware models combined. 86

5.2 Node types in software execution graphs. ............................................. 89
5.3 Edge types in software execution graphs. .......................... 89
5.4 The basic model of Booth: A general representation of a computational system. 92
5.5 A dataflow diagram. ......................................................... 93
5.6 A precedence graph. ....................................................... 94
5.7 The three basic types of cost combing used for computation structure performance analysis. .............................. 95

6.1 “Ideal views” of the integrated environment for information system engineering (a) and capacity managers (c). ................................. 102

7.1 Three alternative views of information systems. The design specification and performance evaluation views of secs. 7.2 and 7.3 will be abstractions of this. 114
7.2 Functional abstraction (a) and performance abstractions (b) of the information system view. ................................. 114
7.3 A design specification abstracts an information system application (a). Both design specifications (b) and components (c) provide operations. ................. 115
7.4 A design specification has a component hierarchy (a) which is a DAG. A component contains a set of subcomponents (b). A non-primitive subcomponent is at the same time a component at the next lower level (c). ......................... 116
7.5 Design specification for the Customer transaction system of fig. 3.5. (The v’s of this and later figures are intended to resemble the component symbol γ. The number represent component indexes.) .......................... 117
7.6 Design specification for the Accounting system of fig. 3.6 before decomposing the Get balances process. ................................. 118
7.7 Design specification for the Accounting system of fig. 3.6 after decomposing the Get balances process. ................................. 118
7.8 The performance modelling view abstracts the concepts of workload, computer, and performance. ................................................. 119
7.9 Performance modelling views of performance model (a), workload model (b), organisation workload (c), and workload module (c-d). ................................. 119
7.10 The workload module visit matrix maps workload module operations onto performance model service centres. ................................. 120
7.11 Overall picture of the performance modelling view. ................................. 121
7.12 A design specification from the functional abstraction mapped onto a workload module of the performance abstraction. ................................. 122

8.1 Design specification for the Customer transaction system of fig. 3.5. (This figure is a duplicate of fig. 7.5 for easy reference.) ................................. 127
8.8 Queueing network performance model for the hardware platform of table 8.3. 140

9.1 Higher-level (a) and lowest-level (b) layers in the Customer transaction specification of fig. 8.1. ...................................................... 150

9.3 Software platform (a) and composite workload model (b) for the bank example. 158

10.1 The matrix of possible performance model sensitivity analyses. ............. 184

10.4 The matrix of possible design specification sensitivity analyses. .......... 188

10.7 Validation of the sensitivity analysis technique. ............................... 191

11.1 Extended design specification for the accounting software and hardware system. Allocation of components among computers is also indicated. .......... 194

11.3 Queueing network performance model for the distributed hardware platform of table 11.2. ......................................................... 197

11.4 Primitive subspecifications for the (extended) design specification of fig. 11.1. 201

11.9 Primitive and top-level subspecifications for the (extended) design specification of fig. 11.1. .............................................................. 206

11.11 GSPN subtemplates for binary AND (a) and XOR (b), as well as REP (c) input connectives, and for binary AND (d) and XOR (e), as well as COND (f) and REP (g) output connectives. .............................. 208

11.12 The "short-hand" output REP connective template of fig. 11.11 (a) and the same template in expanded form (b). ................................. 208

11.13 GSPN workload templates for terminal (a), batch (b), and short-circuit (c) workload intensities. ......................................................... 209

11.14 Generalised stochastic Petri-net model generated for the top-level PrM model of fig. 3.6. ................................................................. 209

11.15 Subspecifications for the refined Accounting system design specification. ... 213

11.16 Generalised stochastic Petri-net model generated for the intermediate subspecification of fig. 11.15. ............................................... 213

11.17 An integration of the PPP and IMSE environments. ............................ 216

12.1 The Customer transaction top-level model of fig. 3.5 drawn in the PrM tool. 222

12.2 The Make withdrawal submodel of fig. 3.8 drawn in the PrM tool. .......... 223

12.5 The operations provided by the PrM model of fig. 12.1. ...................... 225

12.6 The organisation workload on the PrM model of fig. 12.1. .................... 225

12.7 The operations provided by the Make withdrawal submodel of fig. 12.2. ... 226
12.8 The operations provided by a submodel of the Accept withdrawal process of fig. 12.1. (Such a submodel does not exist at the current point of development.) 226

12.10 Execution characterisations for the Make withdrawal process of fig. 12.1 specified in the PrM tool. ........................................... 228

12.11 Execution characterisations for the Accept withdrawal process of the PrM submodel of fig. 12.2. ........................................... 229

12.14 The PrM platform for the Customer transaction specification of fig. 12.1. ........................................... 231

12.15 The hardware resource demands of the Write statement process of fig. 12.1. 232

12.16 The PrM platform mapping for the Customer transaction specification of fig. 12.1. ........................................... 237

12.17 The queueing network of fig. 8.8 drawn in the IMSE QNET tool. ......... 238

12.18 The sp module generated by the PrM tool for the Customer transaction application. ........................................... 239

12.19 The sp module implementation generated by the PrM tool for the sp module of fig. 12.18. ........................................... 239

13.2 The software platform for the banking model. .......................... 245

13.3 PrM tool menu interface for minimum and maximum execution characterisations. 247

13.4 PrM tool menu interface for parametric execution characterisations. .... 249

13.5 PrM tool main menu for specifying distributed platforms. The distributed hardware platform for the Accounting application has two virtual machines and one communication component. ............................ 251

13.6 PrM tool submenus for specifying virtual machines and components for the distributed platform of fig. 13.5. ......................... 252

13.7 PrM tool menu interface for specifying process resource demands. The process only uses a (non-proper) subset of the virtual machines and communication components provided by the platform of fig. 13.5. ......................... 253
List of Tables

1.1 A brief outline of this thesis. ........................................ 9

8.2 Entry and activity matrices for the customer transaction specification of fig. 8.1. 128
8.3 Hardware platform for the Customer transaction model of fig. 3.5. .......... 129
8.4 Visit matrices for the customer transaction specification of fig. 8.1. ....... 129
8.5 Workload derivation technique. ....................................... 130
8.6 Workload derivation results. ......................................... 134
8.7 Performance analysis results. ....................................... 139
8.9 Overview of combined workload and performance model sensitivity analysis. 143

9.2 Software platform for the Customer transaction model of fig. 3.5. ....... 158
9.4 Composite workload model analysis technique. .......................... 161

10.2 Open queueing network sensitivity analysis technique. .................. 184
10.3 Closed mean value analysis technique. ................................ 187
10.5 Non-primitive component sensitivities. ................................ 189
10.6 Software sensitivity analysis results. .................................. 190

11.2 Distributed hardware platform for the Accounting model of fig. 3.6. .... 196
11.5 Form of the Screen manager subspecification aggregate. ................ 202
11.6 Form of the component aggregates of the Screen manager subspecification. 202
11.7 Operation list for the PrM model of fig. 3.6. ........................... 203
11.8 Organisation workload for the PrM model of fig. 3.6. .................... 203
11.10 Execution characterisations for the Present menu and Update balances processes of fig. 3.6 in textual form. .......................... 207

12.3 Operation list for the PrM model of fig. 12.1. ........................... 224
12.4 Organisation workload for the PrM model of fig. 12.1. ...................... 224
12.9 Execution characterisations for the Make withdrawal process of fig. 12.1 in
  textual form. ................................................................. 227
12.12 Non-distributed hardware platform for the Customer transaction model of
  fig. 3.5. ................................................................. 230
12.13 Visit counts of the Write statement process of fig. 12.1. ...................... 230
13.1 Non-distributed PrM platform for the Customer transaction model of fig. 12.1. 244

B.1 Open mean value analysis technique. ........................................... 272
B.2 Closed mean value analysis technique. ....................................... 274
Bibliography


[106] Information systems factory (ISF). Presentation Slides.


