Measurement-Based Composite Computational Work Modelling of Software

Vidar Vetland

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

August 11, 1993
To my parents, Randi & Svein

Il bruco, mediante l'esercitato studio di tessere con mirabile artificio e sottile lavoro intorno a sè la nova abitazione, esce poi fori di quella colle dipinte e belle ali, con quelle lanciandosi verso il cielo‡.

Leonardo da Vinci

‡English translation: The caterpillar, by diligence, weaves with wonderful skill and fine work its new dwelling around itself, then emerges from it with beautiful coloured wings, and rises on them towards the sky.
Abstract

The motivation for composite work modelling is twofold. Such modelling has the potential of reducing the cost of building performance models of existing systems. In addition, composite work models can be used, at a reasonable cost, in order to ensure that new systems can meet their performance requirements. The objective has been to find how to capture and reuse performance characteristics of software components in order to reduce the cost of obtaining sufficiently accurate parameter values for contention models of the computer system. This thesis presents a method for planning of measurement in order to obtain reusable parameter values at a reasonable cost and to provide reasonable prediction accuracy. The practical use of the measured data is also supported by the method.

The approach is to use composite computational work models to define the relationships between measurable characteristics of software and hardware components. The work model components are arranged in a hierarchy that represents multi-layered software, as well as the relationships between the components. The number of times an operation calls its suboperations is found by measurement. The composite computational work model gives an overview of the needed measurement data. Cost, accuracy and reusability of the measurement data may be assessed through analysis of the work model prior to the capture of the measurement data. Planning based on this analysis makes it possible to avoid cost overruns and significant accuracy problems. Since a composite work model contains operations at different levels in a multi-layered model of the software, it can provide parameter values for contention models at different levels of detail.

The feasibility of composite work modelling was evaluated by performing three case studies of distributed transaction-processing systems. Hence, typical modelling and measurement problems could be examined. The case studies showed that adequate data could not be obtained because measurement tools were insufficient and because relevant performance data are often regarded as proprietary.

This thesis illuminates the considerations that are involved in the trade-off between reusability, accuracy, and cost of composite work modelling. In addition, the need for performance information for such modelling is analysed thoroughly. Contemporary measurement tools are not suitable for composite work modelling. Therefore, requirements for new measurement tools are presented.
Acknowledgements

I would like to thank my supervisor, professor Arne Sølvberg, for inviting me to do my doctoral work in his group. The starting point for the doctoral work was his ideas about the need for performance engineering during development and maintenance of information systems. He read my ever-halting drafts when I needed feedback although I tended to give him the drafts when it didn’t fit into his time schedule. His skills in looking at things as a whole are unbeatable.

I am indebted to my advisor, professor Peter Hughes, for introducing me to the performance evaluation field and to his own ideas in particular. I am thankful for the hours we spent discussing interesting problems and for the valuable feedback he gave me.

During the doctoral study I worked together with my fellow doctoral students Andreas L. Opdahl and Gunnar Brataas. Andreas, who got his doctoral degree in December 1992, had lots of experience in performance evaluation when I entered the performance arena. He taught me a lot. Gunnar was the last of us to enter the field. He quickly joined the discussions and the three of us created a good working environment with frequent interesting discussions. Much of my understanding of the field is the result of co-operation with them.

The first two and a half years of my doctoral study I was working in the ESPRIT project IMSE. There I met several researchers in the performance field. I would like to emphasize my close relations to the Mid-England team, Peter Hughes, Eric Barber, Cydney Minkowitz, Nick Xenios, Graham Titterington, and the Milano/Pavia team, Maria Calzarossa, Giuseppe Serazzi, Luisa Massari, Paola Rossaro, Alessandro Merlo. They all gave me a lot of inspiration in my work. Eric Barber read parts of this thesis and gave me valuable feedback.

I would like to thank several people and companies for support and supply of information. I spent two months, June and July of 1992, working at International Computers Limited, Open Framework Division in Manchester, England. In particular, Ian Macadie (my host), Neeta Naran and Martin Tonge were of great help. I would also like to thank the EDP personnel at The Regional Hospital in Trondheim, Knut Ekren, Anne Hofstad, Olaf Schelderup, and Tandem Computers in Trondheim, Inge Sletbakk, Thore Smevik, Asmund Myrbostad, and Tandem Computers in Oslo, Herman Gunnufsen, for all their help.

I would also like to thank for helpful discussions with Anne Helga Seltveit, Odd Ivar Lindland, Tor Egge, Bjørn Gulla, Anund Lie, Guttorm Sindre, and Stig Gunnar Røthe.

Thanks to Catherine Churchill, Tore Amble, Alessandro Merlo and Paola Rossaro for participating in the translation of Leonardo da Vinci’s text. Also thanks to professor Crespi Reghizzi for letting me use the same citation as he used in his PhD thesis about 20 years ago.

Last but not least, I would like to thank all the members of the Information System (IS) group at NTH for a very friendly and stimulating working environment.
# Contents

Preface ix

I Overview 1

1 Intro. to Composite Computational Work Modelling 3
1.1 Three Aspects of Performance Modelling 5
1.2 Engineering Principles 7
1.3 The Usefulness of Measurement Data 8
1.4 Composite Computational Work Models 11
1.5 Composite Work Model Granularity 13
1.6 Degree of Parameterisation 14
1.7 Use of Composite Work Models 14
1.8 The Phases of a Performance Modelling Study 15
1.9 The Performance Modelling Cycle 17
1.10 Performance Considerations in Industry 18
1.11 Tool Support 19

2 Distributed Transaction-Processing Computer Systems 21
2.1 Basic Concepts 21
2.2 Distributed Computer Systems 22
2.3 Software Architectures 24
2.4 Software Components 26
2.5 Remote Procedure Calls (RPC) 27
2.6 Summary 29

3 Presentation of the Case Studies 31
3.1 The Client-Server Example 31
3.2 International Computers Limited (ICL) 35
3.3 The Regional Hospital in Trondheim 38

II State of the Art 43

4 Contention Modelling 45
4.1 Expected Results of Performance Model Runs 45
4.2 Contention for Computer Resources 46
4.3 Computer System Bottlenecks 47
4.4 Different Contention Modelling Formalisms 47
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5</td>
<td>Discrete Event Simulation</td>
<td>48</td>
</tr>
<tr>
<td>4.6</td>
<td>Queueing Network Models</td>
<td>52</td>
</tr>
<tr>
<td>4.7</td>
<td>Petri Nets</td>
<td>61</td>
</tr>
<tr>
<td>4.8</td>
<td>Performance Models of Software Processes</td>
<td>66</td>
</tr>
<tr>
<td>4.9</td>
<td>Summary of Needed Parameters</td>
<td>69</td>
</tr>
<tr>
<td>5</td>
<td>Workload Modelling</td>
<td>71</td>
</tr>
<tr>
<td>5.1</td>
<td>Properties of Workload Models</td>
<td>72</td>
</tr>
<tr>
<td>5.2</td>
<td>The Steps in Workload Modelling</td>
<td>75</td>
</tr>
<tr>
<td>5.3</td>
<td>Selection of Workload Components</td>
<td>75</td>
</tr>
<tr>
<td>5.4</td>
<td>Selection of Workload Parameters</td>
<td>78</td>
</tr>
<tr>
<td>5.5</td>
<td>Reduction of Workload Model Size</td>
<td>80</td>
</tr>
<tr>
<td>5.6</td>
<td>Workload Modelling of Distributed Systems</td>
<td>81</td>
</tr>
<tr>
<td>5.7</td>
<td>Need for Engineering Principles</td>
<td>82</td>
</tr>
<tr>
<td>6</td>
<td>Work Modelling</td>
<td>83</td>
</tr>
<tr>
<td>6.1</td>
<td>Operations in an SP Model</td>
<td>84</td>
</tr>
<tr>
<td>6.2</td>
<td>The Structure of an SP Model</td>
<td>88</td>
</tr>
<tr>
<td>6.3</td>
<td>Formulation of SP Models</td>
<td>93</td>
</tr>
<tr>
<td>6.4</td>
<td>Calculation of Devolved Work</td>
<td>97</td>
</tr>
<tr>
<td>6.5</td>
<td>Assignment of Parameter Values</td>
<td>100</td>
</tr>
<tr>
<td>6.6</td>
<td>Benefits of SP Modelling</td>
<td>101</td>
</tr>
<tr>
<td>7</td>
<td>Computer System Measurement</td>
<td>103</td>
</tr>
<tr>
<td>7.1</td>
<td>Suitable Workloads for Measurement</td>
<td>104</td>
</tr>
<tr>
<td>7.2</td>
<td>Classification of Measurement Data</td>
<td>105</td>
</tr>
<tr>
<td>7.3</td>
<td>Measurable Components</td>
<td>106</td>
</tr>
<tr>
<td>7.4</td>
<td>Measurement Principles</td>
<td>107</td>
</tr>
<tr>
<td>7.5</td>
<td>Characteristics of Measurement Tools</td>
<td>108</td>
</tr>
<tr>
<td>7.6</td>
<td>Types of Measurement Tools</td>
<td>109</td>
</tr>
<tr>
<td>7.7</td>
<td>Requirements on Measurement Tools</td>
<td>110</td>
</tr>
<tr>
<td>7.8</td>
<td>Measurement of Distributed Systems</td>
<td>110</td>
</tr>
<tr>
<td>7.9</td>
<td>Design for Measurement</td>
<td>112</td>
</tr>
<tr>
<td>7.10</td>
<td>Measurement Method</td>
<td>112</td>
</tr>
<tr>
<td>8</td>
<td>Tool Support for Performance Evaluation</td>
<td>115</td>
</tr>
<tr>
<td>8.1</td>
<td>Measurement Tools</td>
<td>115</td>
</tr>
<tr>
<td>8.2</td>
<td>Stand-alone Software Tools</td>
<td>121</td>
</tr>
<tr>
<td>8.3</td>
<td>Integration of Software Tools</td>
<td>123</td>
</tr>
<tr>
<td>9</td>
<td>Sizing of Computer Systems</td>
<td>127</td>
</tr>
<tr>
<td>9.1</td>
<td>The Sizing Procedure</td>
<td>127</td>
</tr>
<tr>
<td>9.2</td>
<td>Sizing Tools</td>
<td>129</td>
</tr>
<tr>
<td>9.3</td>
<td>Users of a Sizing Tool</td>
<td>129</td>
</tr>
<tr>
<td>9.4</td>
<td>Deficiencies of Contemporary Sizing Tools</td>
<td>130</td>
</tr>
</tbody>
</table>
### CONTENTS

#### III Composite Computational Work Modelling

10 Relationships between Work Model Components

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1 Prerequisites</td>
<td>133</td>
</tr>
<tr>
<td>10.2 Availability of Performance Information</td>
<td>134</td>
</tr>
<tr>
<td>10.3 Classification of Complexity Functions</td>
<td>135</td>
</tr>
<tr>
<td>10.4 Types of Complexity Function Parameters</td>
<td>137</td>
</tr>
<tr>
<td>10.5 Examples of Complexity Functions</td>
<td>138</td>
</tr>
<tr>
<td>10.6 The Parameter “Curse”</td>
<td>140</td>
</tr>
<tr>
<td>10.7 Examples from Case Studies</td>
<td>140</td>
</tr>
<tr>
<td>10.8 Overall Considerations</td>
<td>141</td>
</tr>
</tbody>
</table>

11 Formulation of Composite Work Models

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.1 Propose a Composite Work Model</td>
<td>143</td>
</tr>
<tr>
<td>11.2 Assess the Feasibility of Measurements</td>
<td>148</td>
</tr>
<tr>
<td>11.3 Assess Compatibility with Contention Models</td>
<td>151</td>
</tr>
<tr>
<td>11.4 Establish the Form of Complexity Functions</td>
<td>151</td>
</tr>
<tr>
<td>11.5 Assess Reusability</td>
<td>153</td>
</tr>
<tr>
<td>11.6 Assess Risk of Error Propagation</td>
<td>155</td>
</tr>
<tr>
<td>11.7 Assess Measurement Cost</td>
<td>158</td>
</tr>
<tr>
<td>11.8 Finding a Trade-off</td>
<td>161</td>
</tr>
</tbody>
</table>

12 Practical Composite Work Modelling

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.1 The Client-Server Example</td>
<td>163</td>
</tr>
<tr>
<td>12.2 International Computers Limited</td>
<td>178</td>
</tr>
<tr>
<td>12.3 The Hospital Case</td>
<td>187</td>
</tr>
</tbody>
</table>

13 Measurement of Computational Work

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.1 Necessary Measurement Data</td>
<td>195</td>
</tr>
<tr>
<td>13.2 Three Modes of Measurement</td>
<td>197</td>
</tr>
<tr>
<td>13.3 Derivation of Complexity Functions</td>
<td>199</td>
</tr>
</tbody>
</table>

14 The Use of Composite Work Models

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>14.1 The Composite Work Model</td>
<td>205</td>
</tr>
<tr>
<td>14.2 Static and Dynamic Aspects</td>
<td>206</td>
</tr>
<tr>
<td>14.3 Utilisation of Hardware Devices</td>
<td>207</td>
</tr>
<tr>
<td>14.4 Parameters in terms of Transaction Steps</td>
<td>208</td>
</tr>
<tr>
<td>14.5 Parameters in terms of Entire Transactions</td>
<td>211</td>
</tr>
<tr>
<td>14.6 Parameters in terms of Software Processes</td>
<td>212</td>
</tr>
</tbody>
</table>

15 Assessment of Completed Composite Work Models

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.1 Long-Term Cost of Parameter Capture</td>
<td>215</td>
</tr>
<tr>
<td>15.2 Qualitative Validation</td>
<td>216</td>
</tr>
<tr>
<td>15.3 Quantitative Validation</td>
<td>218</td>
</tr>
</tbody>
</table>

#### IV Concluding Discussions and Remarks

16 Concluding Remarks

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 Concluding Remarks</td>
<td>223</td>
</tr>
</tbody>
</table>
CONTENTS

16.1 Recapitulation of Motivation and Objectives ........................................ 223
16.2 Major Results ....................................................................................... 224
16.3 Discussion of Results ........................................................................... 226
16.4 Experience Gained from the Case Studies ............................................ 227
16.5 Experience with SP .............................................................................. 228
16.6 Commercial Exploitation ...................................................................... 229
16.7 Critique ................................................................................................. 229
16.8 Further Work ......................................................................................... 230
16.9 Summing Up ......................................................................................... 231

V Appendices and Bibliography ...................................................................... 233

A Abbreviations ............................................................................................ 235
B International Computers Limited .............................................................. 237
C The Client-Server Example ........................................................................ 241
List of Figures ............................................................................................... 245
List of Tables ................................................................................................ 249
Bibliography ................................................................................................. 251
Preface

Problem Statement

The current best practice in performance evaluation of computer systems is to build performance models of the systems from scratch. This leads to high modelling costs because modelling expertise is needed. Therefore performance modelling is only done for systems where the penalty is very high for not meeting the performance requirements, e.g., the response times of transactions provided by banking systems must be in accordance with the working pace of employees in the bank; the penalty is that the quality of customer service is reduced and that the working conditions for bank employees become unsatisfactory.

Contemporary performance evaluation techniques comprise queueing network analysis, Petri net analysis, and simulation techniques. When the usage of computer system resources is properly described by performance model parameters, the performance evaluation techniques may produce estimates of response times, resource utilisations, and throughputs. While the literature on performance evaluation techniques is overwhelming, the literature covering cost-effective parameter capture for performance models is limited.

Traditionally performance modellers have tended to ignore the difficulty of obtaining parameter values which represent the resource demands of multi-layered software. In practice the use of performance engineering in large-scale systems development is limited by the cost of acquiring appropriate performance information about the various software components. There are two major problems in the capture of performance-related characteristics of software. The first problem is to determine the amount of computer resource demands per transaction at a level of detail which is suitable for a performance model. The second problem concerns the trade-off between the accuracy with which the resource demands of the software are measured, the reusability of such measurement data, and the cost of the required measurement experiments.

The motivation for composite work modelling is that it has the potential of reducing the cost of building performance models of existing systems. In addition, composite work models can be used, at a reasonable cost, in order to ensure that new systems can meet their performance requirements.
The objective of this thesis is to find how to capture and reuse performance characteristics of software components in order to reduce the cost of obtaining sufficiently accurate parameter values for contention models of the computer system. The approach is to structure software performance measurement data according to a structural model of the software, i.e., a composite computational work model, in order to facilitate reuse of measurement data. Such an approach requires careful planning because large initial investments in measurement experiments are necessary. The planning must take into consideration software modularity, possibility of measurement, and cost, accuracy, and reusability of measurement data.

## Approach

In order to analyse the performance of a computer system the following three aspects must be taken into account: the load, the computational work, and the contention for computer resources. A load model describes the user behaviour, e.g., how many transactions are initiated per time unit and what is the typical number of users using the computer resources at the same time. A computational work model describes the computer resource demand per job or transaction, e.g., a transaction demands execution of 250000 machine instructions and 310 disk accesses. A contention model describes how the different jobs or transactions “behave” and how they compete for computer system resources such as CPUs, disks, and physical memory. Typical outputs from a contention model are transaction response times, transaction throughputs, lengths of queues for computer devices, and utilisations of computer devices.

The relationships between different performance modelling activities are depicted in Figure 0.1: Contention modelling requires specific parameter values from a load model and from a (composite) work model. Contention, load, and (composite) work models must be mutually compatible. Measurement experiments provide parameter values for contention, load, and (composite) work models. It is assumed that workload models can be divided into separate load and work models. It is also assumed that a work model can be decomposed into components to make up a composite work model. A discussion of this assumption is necessary because useful performance information may be lost when the performance of the software is characterised in terms of its components and then combined.

The focus in this thesis will be on composite computational work models, in particular how to build and use them. However, relevant aspects of load models and contention models are described, discussed and included in examples. An overview of contention modelling techniques is given and the parameter needs of contention models are surveyed. These parameter needs serve as requirements on the outputs of load models and work models. A presentation of state-of-the-art of workload modelling is given. Performing measurement experiments to describe resource demands in a composite work model requires measurement-in-the-large, i.e., systematic measurement of interactions between software components. Systematic measurement of
software interactions is discussed in the context of the state-of-the-art of measurement of computer systems.

Desirable and undesirable effects of using composite work models are presented and discussed. The planning which is necessary to formulate a composite work model with desirable properties is treated in detail. This phase is critical because a bad formulation of a composite work model leads to superfluous measurement experiments. The availability of measurement data and compatibility with contention model parameter requirements are checked. If a composite work model passes these tests, composite work model accuracy, cost, and reusability must be assessed. Alternative formulations of composite work models may be tested and assessed in the same manner. The best formulation is chosen according to a trade-off between accuracy, cost, and reusability.

When a composite work model has been formulated according to the trade-off, measurement experiments are performed according to the composite work model in order to obtain complexity functions. When all the complexity functions have been determined, the composite work model must be validated with respect to accuracy and assessed with respect to completeness and robustness. In some cases, e.g., when some complexity functions are dependent on the load, it may be necessary to carry out more measurement experiments in order to validate the work model.

A major issue is to examine the assumption that to structure software measurement data according to a model of the software structure reduces the long-term cost of measurement for performance modelling. The structure of the measurement data is
selected based on a trade-off between short-term cost, reusability, and accuracy.

Peter Hughes has proposed the *Structure and Performance specification method*, SP, which supports composite computational work modelling. So far, this method has been tested in a few small cases. In this thesis, the SP method is applied on complex computer systems such as distributed heterogeneous transaction-processing computer systems. Problems in the application of the SP method are discussed and improvements are proposed.

Composite work modelling by using the SP method is illustrated through three case studies which all involve distributed systems. Two of the case studies are based on real-life systems: An open distributed heterogeneous computer system at ICL (International Computers Limited, Manchester) and the multiprocessor computer system at RiT (the regional hospital in Trondheim). The third case study involves a minimal distributed system, an artificial bank application, which will be used as a test-bed for experimentation as well as a source of simple illustrating examples.

Since each software component characterisation requires in-depth knowledge of the software component's implementation, composite work modelling is not a one-person endeavour and in many cases not even a one-company endeavour. The performance characterisations of basic software such as database management systems must be provided by the company which develops and maintains the software. Consequently, the information about the components that were encountered in the case studies was limited and therefore inhibited proper validation of the composite work models.

**Related Doctoral Works**

This doctoral thesis has been written in connection with two other doctoral theses. Andreas L. Opdahl finished his thesis in December 1992 while Gunnar Brataas will deliver his thesis in 1994.

The doctoral thesis of Andreas L. Opdahl [78] is about performance engineering of information systems in order to predict and improve the performance of projected application software during development. A basic framework for performance prediction is introduced. This framework is based on a conglomerate of separate models for computer hardware, existing and projected software applications, and the organisation using the applications. The framework is extended with support for parameter capture.

The doctoral thesis of Gunnar Brataas[20] will focus on performance engineering of office information systems. He will look at the information which is available during development of office information systems, and how it is possible to take care of performance during development of new applications which uses operations from existing applications modelled by SP.
About the Work

Most of the work that is presented was done in the context of the IMSE project on computer system performance modelling [57, 21, 77, 78]. The IMSE project was a collaborative research project supported by the CEC (ESPRIT project no. 2143). It was carried out by the following organisations: BNR Europe STL, 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.

Two practical case studies were performed, neither of which could be fully completed. In spite of this, much valuable information was obtained. The case study at the Regional Hospital in Trondheim was done from September 1989 to December 1991; at times intensively, at times quite sporadically. This case study was hampered by limited availability of measurement data. Since the hospital system was run in production, it was impossible to instrument the software. However, the case provided very useful hands-on experience on computer system performance measurements. Only the part of the case study that was carried out by the author is reported in detail in this thesis.

The other practical case study was done at International Computers Limited (ICL) in Manchester, England, during the summer of 1992. There were no problems that hampered progress. The investigated system was a prototype of a heterogeneous distributed computer system which could easily be instrumented. The only worry concerned the resolution of the measurement tools. Unfortunately, time didn’t allow completion of the measurement experiments.

No software or hardware component has been examined in detail. The focus has been on the feasibility and the overall considerations of composite work modelling.

Shared Work

The context in which this thesis is written was established by joint work with Andreas L. Opdahl and Gunnar Brataas and by discussions with professor Arne Sølvberg and professor Peter Hughes. Our work was heavily influenced by the IMSE project. All the modelling details that are presented in this thesis and the methods and conclusions based on these detailed studies are the author’s own contributions.
Outline of the Thesis

Part I of the thesis gives an overview of relevant concepts and principles of composite work modelling. Part II presents state-of-the-art of relevant areas of performance modelling. Emphasis is put on work modelling and measurement. Part III discusses properties and trade-offs concerning composite work models. The case studies are used in examples, and form the basis for general discussions of composite work modelling. Part IV provides concluding remarks and suggestions for further work. Part V contains lists of abbreviations and explanations of important concepts. In addition, a bibliography and an index are provided.

Typographical Conventions

References in the text to names of nodes, entities, components or operations in models are written in the teletype font. A concept is usually written in italics the first time it is used. Italics is also used to put strong emphasis on a word or several words in a sentence. Also contrast may be shown by putting the contrasting word(s) in italics. Any reference to the classification of operations according to the SP method (communication, processing, discrimination or memory) is written in italics.
Part I

Overview

This part consists of three chapters. Chapter 1 contains an introduction to composite computational work modelling. The most important concepts in this thesis are introduced there. Chapter 2 provides an introduction to distributed transaction-processing computer systems. The methods and techniques that are presented in subsequent chapters are primarily targeted at such systems. Chapter 3 presents the three case studies.
Chapter 1

Introduction to Composite Computational Work Modelling

Every computer user knows that good performance is vital to computerised information systems. Users are sensitive to the response times of a system. Therefore it is important that systems behave in accordance with each user’s pace of working. It is also important that the workload generated by all the users is in accordance with the resources available. Too much resource is a waste of money and too little resource makes the system impractical to use as depicted in Figure 1.1. There is a conflict between good response times and good utilisation of computer devices as shown in Figure 1.2. A trade-off between the two must be found.

![Diagram](image)

Figure 1.1: The (lack of) balance between resources needed and resources provided as described in [92]

Computerised information systems are so complex that one cannot rely on human intuition regarding the performance consequences of system modifications. Therefore, some kind of performance modelling is necessary. Performance problems occur when the supply of a computer resource for a significant amount of time is less than the demand. Performance models are capable of calculating the delays caused by contention for computer resources, the average number of transactions that must queue for a computer resource, and the number of transactions that are completed
Performance evaluations of information systems are undertaken only when the (high) modelling cost can be justified. The duration of a performance study can also be prohibitively long. Yet another problem is the limited supply of expertise to do a performance evaluation.

Performance evaluation is still considered as an art which lacks an accepted and validated theory. It is necessary to apply engineering principles to make performance evaluation techniques more useful. The current situation is that performance experts are called upon to build a performance model for a specific purpose. It is well-known that it is difficult to make general performance models that can be used to investigate different performance problems. It may nevertheless be worthwhile to investigate whether engineering principles can be used in some phases of a performance study.

Traditionally performance modellers have tended to ignore the difficulty of obtaining parameter values to represent the resource demands of complex software. In practice the use of performance engineering in large-scale systems development is limited by the cost of acquiring appropriate performance information about the various software components. However, if such information can be reused when components are combined in different ways, then the cost of measurement may be justified. Reuse may be achieved by means of structured composition in the calculation of the computer resource demands of the software. Composite work models have the potential of reducing the long-term measurement cost because measurement data are reused. The dependencies between the issues that must be addressed to investigate the usefulness of composite work models are illustrated in Figure 1.3. The captured measurement data are the basis for characterisation of the performance of software components. Based on a structural model of the software, a composite work model, the most appropriate composition and size of software components is chosen in order
Figure 1.3: Issues that must be addressed to reduce reduced long-term measurement cost

to achieve reusability of work model components. It should be demonstrated that reuse of measurement data, i.e., work model components, decreases the long-term cost of parameter capture for performance models.

The performance models and methods presented in this thesis are primarily targeted at transaction-processing computer systems. According to Highleyman [53], a transaction processing system is an on-line real-time multiuser system that accepts requests and returns responses to the requests. The requests usually involve database access. All transaction-processing systems are interactive; the requests to the system can usually not be planned in advance. The next request from a user may depend on the system's previous response. Some transaction-processing systems are real-time systems. This means that there are absolute limits on the allowable response times. In distributed transaction processing systems, transactions are executed by means of co-operating processes. If the same process is requested by two or more transactions, only one can execute at a time. Therefore contention for both the software and the hardware must be taken into account.

1.1 Three Aspects of Performance Modelling

Performance models must represent how the users influence the performance of a computer system by producing inputs (programs, data, commands). The amount of user's inputs to a computer system and the time-varying patterns of the inputs are modelled in a workload model. Contention models describe how the workload may
cause contention effects when computer resources are shared by several transactions. The performance of the system influences the workload because there is a feedback loop as depicted in Figure 1.4. Despite the existence of a feedback loop, it is usually assumed that the workload is insensitive to changes in a system's performance. The impact of performance changes on user behaviour is not discussed in this thesis.

![Diagram of User Community, Workload, Computer System, and Performance](image)

**Figure 1.4:** The relationship between workload and performance (from [42, pp. 5])

![Diagram of Services at User Interface, Composite Work Model, Devolved Work, and Contention Model](image)

**Figure 1.5:** The load parameters are applied directly to the contention model while the mapping between offered services and resources is modelled by a composite work model.

Three different aspects of performance models are distinguished: the load, the computational work, and the contention for computer resources (See also [21, 76]). Each aspect is described by a model which is compatible with models of the other two aspects. A load model provides *dynamic* information, i.e., intensities, to the contention model, while a work model provides static information to the contention model (Figure 1.5). In the literature, work models and load models are usually combined. In summary:

\[
\text{Performance Model} = \text{Load Model} + \text{Work Model} + \text{Contention Model}
\]
Load Model  A load model describes the user behaviour, e.g., how many transactions are initiated per time unit, what is the typical number of users using the computer resources at the same time. The load on each user-function must be characterised as either transaction (arrival rates), batch (constant number of active user-functions), or terminal (number of interactive users and their think times) [65].

Work Model  The work models and the load models share a definition of a work unit. A work unit is a suitable mix of the services that the computer system can provide, e.g., an average transaction or a typical transaction mix. A computational work model describes the computer resource demand per job or transaction, e.g., a transaction may demand execution of 250000 machine instructions and 310 disk accesses. The resource demands are devolved by transforming work units at the user interface to corresponding work units at the hardware level, e.g., from transactions to CPU instructions, disk accesses, network messages. The result of this transformation is called devolved work. The composite work modelling approach is only possible when work and work intensities (e.g., rate of work) are separated, because work intensities do not have any meaning for interactions between different layers in the software. A method is required to characterise the performance-related attributes of software components independently of its environment and independently of its behaviour in the time domain. Such an approach is provided by the Structure and Performance (SP) specification method which is presented in Chapter 6. Not only useful work is devolved. Overhead work because of contention can be accounted for by introducing load-dependent component specifications. However, such load dependencies “destroy” the neat separation of load and work as depicted in Figure 1.5. A model of a software function is called an operation (or work unit). This distinction is necessary because several operations may be needed to represent a function properly in the work model. In other cases one operation can represent several functions.

Contention Model  A contention model describes how the different jobs or transactions “behave” and how they compete for computer system resources such as CPUs, disks, and physical memory. Contention models are either analytic models or simulation models. They are the means for evaluation of contention effects such as queues, i.e., dynamic effects are modelled. The load model must provide dynamic information for each transaction type and the work model must provide static information for each transaction type. Typical outputs from a contention model are transaction response times, lengths of queues for computer devices, and utilisation of computer devices.

1.2  Engineering Principles

We are used to select, e.g., electrical components based on their performance characteristics. A calculation of the performance characteristics of an electrical system
is based on the characteristics of the components. It is therefore possible to calculate the characteristics of many different combinations of electrical components. The same principle applies to civil engineering and to most other engineering disciplines. However, such engineering principles are not widely used in the evaluation of properties of computer systems.

Langefors [63] has formulated a fundamental principle for system development. This principle combines top-down decomposition of a system followed by bottom-up synthesis of system component characteristics. This principle defines four tasks:

1. Definition of the system as a set of parts
2. Definition of the system structure
3. Definition of properties of system components
4. Derivation of system properties by synthesis

The steps are repeated until the system’s properties can be obtained based on the properties of its components. Step 4 is possible if the decomposition of the system is constructive. The main question concerning constructivity is how much information that is lost when system components are characterised and put together instead of characterising the entire system directly. Here, the fundamental principle for system development will be applied in performance characterisation of the software of computer systems.

1.3 The Usefulness of Measurement Data

Measurement data may be viewed as empirical evidence of what happens during execution of a piece of software on a particular set of hardware devices in a certain context. To obtain reliable measurement data, it is necessary to understand what the measurement data mean and to know the precise context in which the data were captured. Performance characterisations of software components and hardware devices may be derived from measurement data. The main problems are to control the cost of measurement and to generalise from the measurement data.

The current practice is to measure hardware resource usage in terms of the functions provided to the end-user. The software is viewed as a single layer during measurement (Figure 1.6). No interactions between layers of software are taken into account. Since measurement data can be very useful, it is important to capture measurement data which will remain the same for a long time until a software modification invalidates the data. Thus, the “invariant” parts of software and hardware should be identified. Measurement data should be stored in a library so that they can be retrieved easily on demand. It is important to specify when the measurement data are valid to enable detection of modifications that invalidates the data.
Figure 1.6: Software as a mapping between external workload and hardware

The level of measurement detail can be increased by measuring interactions between software components explicitly (Figure 1.7). Each software component is measured in order to characterise its work demand in terms of calls to other (lower-level) software components and in terms of calls to hardware devices. It is preferred that resource usage is characterised in terms of counts of hardware operation invocations. Examples of such primitive operations are block transfer to disk, transmission of a message over a network, and execution of CPU instructions.

To generalise from the measurement data it may be necessary to formulate parameterised functional relationships which represent the relationships between events in the computer system. The events of interest here are invocations of operations of software components and invocations of hardware operations. The following examples are given to illustrate the reusability of measurement data:

Remote procedure calls and UNIX The operations that are provided by the remote procedure call (RPC) library enable interaction between two software components across hardware platforms. If the UNIX operating system is running on both hardware platforms, the relationship between invocations of RPC operations and invocations of UNIX communication operations can be found by measurement. Some RPC operations invoke UNIX communication operations in exactly the same manner every time. Thus, the number of invocations of UNIX communication operations as a result of invocations of RPC operations can be determined once and for all. Some RPC operations may be dependent on the length of the message that is transmitted. Therefore the functional relationship between invocations of RPC and UNIX operations may have to be parameterised. The relationships between invocations of RPC and UNIX operations are valid until the RPC library is modified.

Curses and UNIX A library called “Curses” provides operations for screen handling independently of terminal types in UNIX. The CPU time required to execute
each of these operations can be found by measurement. Even better, the CPU demand should be characterised in terms of instruction counts in order to make the measurement data independent of the CPU speed. These measurement data may be reused every time the CPU demands of Curses operations are needed. However, measurement data must be captured for each terminal type and for each CPU type. In addition, some of the Curses operations may be dependent on the number of characters and their data types. Thus, the reusable measurement data may have to be parameterised to reflect these dependencies.

**Basic Operations of INGRES** The relationship between basic INGRES operations and the number of disk accesses may be represented by reusable measurement data. The number of disk accesses caused by an "insert row" operation may be specified in terms of number of indexes on the database table and the type of database table. Thus, the relationship between invocations of basic INGRES operations and disk accesses can be described by parameterised functional relationships.

From these examples it is obvious that the interactions between complex software components must be described by parameterised functions which are obtained by systematic measurement. The more complex the software components the more complex the functions. In fact, the functions which describe the interactions between components of a computer system are called *complexity functions*. The complexity function concept is defined by the SP method (Chapter 6).
1.4 Composite Computational Work Models

There is a need for structuring of reusable measurement data in order to get a complete picture of how the measurement data are related. A composite work model supports this kind of structuring and it can also be used for combination of reusable measurement data and planning of measurement experiments. The rationale for composite work modelling is to enable reduction of measurement cost by maintaining a library of work model components instead of performing new measurement experiments from scratch in every performance modelling study.

1.4 Composite Computational Work Models

The focus of this thesis is on the decomposition of existing software into appropriate components, how to find appropriate performance properties of each component by measurement experiments, and how to put the pieces together in order to describe the software's resource demands [99]. The means for such modelling are composite computational work models. The main benefit of this approach is that system developers are able to foresee performance consequences of new configurations based on performance characteristics of existing software and hardware components. In addition, the performance consequences of replacing one component in an existing system may be investigated. Ferrari writes [42]:

A workload model should ideally be designed with a clear knowledge of the most probable changes to be made to it.

A composite computational work model is a model of the entire software suite of a computer system. Only those software properties that are relevant for a performance study are included in the work model. There are three properties of a composite work model which in isolation and together must be satisfactory before a composite work model can be used successfully instead of a monolithic work model: Cost, reusability, and accuracy.

As an example, a composite work model of a client-server system is shown in Figure 1.8. The model describes a banking system where the Bank application component (the client) invokes the File server component. The Bank application component runs in CPU#1 and the File server runs in CPU#2. Both the components require the RPC library and the UNIX comms components in order to communicate with each other over the Network. The File server uses the File system to access the Disk. The hierarchic arrangement of the components and the edges describe the dependencies between the components. Operations are defined for each work model component in order to reflect the functionality of the corresponding software component. In a relationship between two components, the operations of the superior component initiates execution of operations in the inferior component. Thus, the operations that are invoked by the users of the system must be provided by the top-level components of the hierarchy.
The relationship between two work model components is described by a complexity matrix where each element in the matrix is a complexity function. A simplified example of a component-subcomponent relationship is shown in Figure 1.9. In this example the superior component, RPC Library, has two operations, clnt_call and svc_sendreply. The inferior component, UNIX comms, has two operations, recvfrom and sendto. The corresponding complexity matrix, which is shown in Figure 1.10, has one element for each possible combination of a superior operation and an inferior operation. Each complexity function represents the number of times the inferior operation is invoked every time the superior operation is invoked. In the example both recvfrom and sendto are invoked once every time either clnt_call or svc_sendreply is invoked.
1.5. Composite Work Model Granularity

\[ C_{\text{RPC-library}}^{\text{UNIX-comms}} = \begin{bmatrix}
  \text{clnt\_call} & \text{recv\_from} & \text{send\_to} \\
  1 & 1 & 1 \end{bmatrix} \]

Figure 1.10: An example of a complexity matrix

1.5 Composite Work Model Granularity

A granularity measure is needed which indicate the detail of characterisation of component interactions, i.e., how many software component interactions that are modelled explicitly. The following granularity measure is used in the rest of this thesis unless otherwise stated:

| The measure of composite work model granularity is the total number of operation-suboperation relationships where the operation invokes the suboperation at least once. |

This granularity measure is independent of the amount of source code that each work model component represents.

The finest granularity of a composite work model corresponds to the identified software components and their functions. The granularity is made coarser by collapsing components at adjacent levels, thus hiding operation-suboperations, and by aggregating component functions. The granularity of a composite work model should be chosen such that:

- its components correspond to the software components that are most likely to be changed or exchanged.
- the model is robust with respect to modifications.
- serious accuracy pitfalls are avoided. Such pitfalls occur when the composite work model have too many levels.
- it represents a measurement plan which can be completed without exceeding the cost limit of the measurement experiments.
- it is easy to calculate resource demands in terms of service demands.
- it is possible to extract resource demands which are compatible with the parameter needs of the contention model.

The process of formulating a composite work model combines considerations of cost, accuracy, and reusability of parameter capture for contention models.
The granularity of measurement data must be related to the granularity of a composite work model, since it is the composite work model which determines which measurement data that should be captured. Therefore:

| The higher the number of software interactions that are measured, the finer the granularity of measurement data. Measurement data fit into a composite work model if and only if the granularities of the two are the same. |

1.6 Degree of Parameterisation

The previous section presented one of the modelling decisions to be made during composite work modelling: the choice of granularity. Another modelling decision is the degree of parameterisation. In most cases there are some factors which influence the execution of operations. Some of the factors may have a lot of possible settings, e.g., the number of data records to be sorted, while other factors have just a few parameter settings, e.g., the size of a disk block. It must be decided which factors that should be modelled explicitly (by parameterisation). This decision is intertwined with the required reusability of the component to which the operation belong. As will be discussed later, the parameterisation also have consequences for the accuracy and cost of composite work modelling and the corresponding measurement experiments.

1.7 Use of Composite Work Models

There are mainly three uses of composite work models. Figure 1.11 shows that a work model component is derived from a software design specification while work model components for existing software components are fetched from a library. These work model components are put together and the resource demands on any component in the model can be calculated. Figure 1.12 shows that all work model components are fetched from the library in order to calculate resource demands to be used in a sizing study. Figure 1.13 shows that a work model component can be replaced by an alternative work model component when the interfaces of the two components are identical. In this way the consequences of software upgrades can be calculated with respect to resource demands.
1.8 The Phases of a Performance Modelling Study

Composite work modelling is just one of many activities in a performance modelling study. In this section, the context of modelling is extended to general guidelines for performance modelling.

A performance modelling study can be expensive. Thus, it is important to evaluate the potential outcomes and gains of such a study. A performance modelling study must be regarded as an investment for which a risk is involved because the study may turn out not to be useful, or even be impossible to carry out. According to Ferrari [42], cost and performance should be separated only for pedagogic purposes and performance evaluation should always be accompanied by some form of cost evaluation. The cost of a performance modelling study is influenced by personnel costs, purchase and maintenance cost for measurement tools and analysis tools. Possible benefits of performance modelling studies are postponement of an upgrade or expansion of the hardware, down-sizing of the system, increased workload on the system, and/or an increase of the users’ productivity.

Ferrari [42] defines five phases of a performance evaluation study:

1. The need for a study arises
2. The objectives of the study are formulated
3. A plan of the study is prepared
4. The plan is implemented

5. The results are interpreted

A (potential) performance problem is identified in phase 1. Phases 2 and 3 require careful cost estimation. Normally the cost is easier to predict than the outcome of a performance evaluation study. The objectives are of key importance because they guide the search for a solution of the performance problem. In phases 3, 4, and 5, a hypothesis is formulated and tested. The hypothesis is modified if the test is unsuccessful. There are feedback loops from a phase to the preceding phases. This enables the iterative procedure which is the basis for the scientific method. It should be noted that the more information that is available, the better hypothesis can be formulated.
1.9 The Performance Modelling Cycle

The formulation, assessment and use of a performance model follows a modelling cycle because the assessment and use provides feedback which may lead to a change in the formulation of the performance model.

Since there are strong relationships between the work, load, and contention aspects of a performance model, it may be difficult to decide which aspects should be treated first. Modelling decisions concerning one aspect influence the formulation of models for the other two aspects.

If some of the aspects have been modelled in a previous performance modelling study, these models should be used if it is possible. Reuse of models requires that the models are valid in the new contexts and that the cost of reuse is considerably less than the cost of developing new models from scratch. For instance, if a work model is available for the software to be evaluated, the load and contention models should be built so that they will be compatible with the existing work model. Of course, reuse should not take place if any modelling assumptions or decisions are violated.

When the work and load models have provided parameter values for the contention model, the contention model can be evaluated or simulated to estimate the effects of contention in the computer system.

In order to gain confidence in a performance model it is necessary to run the model for several cases where the correct performance indices are known. If the performance indices that are calculated by the performance model and the performance indices that are measured in the real system are reasonably close for several cases, the performance model can be used for prediction.

The rationale for performance modelling is that the performance of computer systems can be predicted prior to their construction. The performance model must be robust so that it can be changed and re-evaluated. The effects of changing the configuration of the software and/or the hardware and the effects of adding new software or hardware components should be predictable.
1.10 Performance Considerations in Industry

In a performance modelling study a contention model is built in order to investigate the effects of competition for computer resources. Quality criteria for model parameters are accuracy and timeliness. Timeliness means that the parameters are available when they are needed, i.e., that the parameters can either be reused from previous performance studies or be obtained from scratch reasonably quickly. Performance modelling studies must be cost-efficient and their cost must be kept within the given limits.

Scenarios

Obtain a Tender for a Big Customer Quickly. Computer sales persons must ensure that the computer systems they sell to customers have enough hardware resources to execute the customer's software with acceptable performance. It is common that the requirement specification of a computer system contains very strict requirements on response times, e.g., less than one second in 95% of all transactions and never longer than five seconds. The sales persons are usually not computer experts and they must often prepare tenders in a hurry. In order to obtain the performance estimates the relevant measurement data must be available either by reuse of previous measurement data or by running scripts to perform new measurement experiments.

A Capacity Calculation Tool If the computer vendor can provide a wide range of different hardware devices and many different software configurations it is difficult to make a survey of performance properties of alternative software and hardware configurations for a customer. A tool for capacity calculations would be necessary in order to conduct such a survey. The sales persons cannot perform extra measurement experiments themselves, and asking computer experts in the company to perform such experiments is costly.

The Ideal Situation Ideally all software and hardware components of a computerised information system should be standardised. Each component should also have a performance specification. Each performance specification should be encapsulated. Dependencies between components should be represented by parameters. By putting together encapsulated performance specifications it would be possible to do overall performance evaluations of an information system. It is a requirement that all components have a performance specification so that performance evaluation can take place before purchase. The software and hardware components should be guaranteed to perform better than the predictions of the corresponding performance specifications.
1.11 Tool Support

A performance modelling study may require separate models of the different aspects of a computer system (as explained earlier). The models are described in different formalisms and for each formalism there are often several calculation techniques. To enable the human performance modeller to concentrate on modelling and avoid doing tedious calculations, performance evaluation tool-sets have been built. Building and using a contention model by extensive tool support can be very efficient. However, the time and effort required to capture parameter values for such systems have not improved. Therefore tool support for parameter capture and reuse of measurement data is more important than ever. Some relevant tools will be presented in Chapter 8.
Chapter 2

Distributed Transaction-Processing Computer Systems

The trend towards distributed, heterogeneous, open, modular, and object-oriented software supports reuse of software components. Such system architectures enable application composition [74] as an alternative to application development. It would be interesting to examine such systems in order to obtain reusable performance characterisations of the software components as well. Then it could be possible to combine performance characteristics of software components in the same manner as the corresponding software is composed.

Distributed transaction-processing computer systems provide a frame for application of the techniques and methods that are presented in this thesis. Since these systems are extremely flexible, this frame is not a serious limitation. Distributed transaction-processing computer system architectures are general which implies that many other computer system architectures can be regarded as specialisations of such systems.

2.1 Basic Concepts

The process and transaction concepts are used frequently in descriptions and discussions of distributed transaction-processing systems.

A process is the basic logical unit within a distributed system. Each task in such systems is handled by a process. Processes are the means that the operating system provides for distribution of applications. The allocation of software components to processes determines the maximum degree of parallelism in the system. Of course, the maximum degree of parallelism can only be achieved when each process runs in a processor of its own. Processes are specialised; some run user applications while other processes are responsible for operating system tasks such as device handling. A distinct feature of distributed systems is that application processes are specialised as
well and must co-operate to complete a transaction. That implies that the processes must communicate; normally across computer boundaries.

A transaction is a sequence of interactions between the user and the computer system which together make up a unit of work for the user. During the transaction one or more of the services provided by the system is invoked by the user. The user may spend some time between invocations of the services in order to decide what to do next or in order to do some manual work. The definition of a transaction is more strict in the database community. There a transaction is defined as a basic unit of consistent and reliable computing. A transaction is either completed or all changes in the database are undone.

2.2 Distributed Computer Systems

The evolution towards distributed heterogeneous computer systems started by software which was running as independent operating system processes in one CPU (see Figure 2.1). In more recent systems, software components that are shared by several users, run as separate processes so that these components can serve several programs. This saves memory space and enables concurrency control of requests for software services and computer resources. Software components which run in separate processes are called servers and they must be regarded as computer resources just as hardware devices are regarded as computer resources. The major new aspect introduced here is that transactions are executed by co-operating processes. This type of computer system is depicted in Figure 2.2. So far, the computer systems are neither distributed nor heterogeneous. A distributed system is obtained when the processes which run the software are distributed among different CPUs between which there is a bus connection (as in loosely coupled multiprocessor systems) or a network connection. If the co-operating processes must co-operate across CPU boundaries, the system is distributed. In addition, if the CPUs are of different types, the system is distributed and heterogeneous. This type of computer systems is depicted in Figure 2.3.

![Figure 2.1: No co-operation between processes](image-url)
2.2. Distributed Computer Systems

Figure 2.2: Co-operating processes

Figure 2.3: Distributed co-operating processes
In a distributed system, a process executes an application software component and uses the communication library to communicate with other components of the application, which execute in other processes (see Figure 2.4). The "execution path" of a transaction is a path visiting several processes possibly in different computers (see Figure 2.5).

**Figure 2.4: Two communicating processes**

**Figure 2.5: Transaction execution paths**

### 2.3 Software Architectures

Software architectures have been defined in order to facilitate design of open, extendible, and reconfigurable computer systems. A software architecture specifies the components the software should consist of. Each component is either an infrastructure component or an application component. Infrastructure components are responsible for, e.g., the communication between components, the permanent data storage, transaction management, and monitoring. The same infrastructure components will appear in the software of all computer systems which are designed and
implemented according to the same software architecture. On the other hand, the application components are only used in a limited number of systems. They must conform to the same interface format in order to fit into the infrastructure that are defined by the software architecture.

Two software architectures were encountered during the case studies. The Advanced Network System Architecture (ANSA) and the PATHWAY architecture. In both software architectures the processes call other processes and wait for the result. The relationship between the processes is called client-server in ANSA and requester-server in PATHWAY. Such relationships may be implemented by remote procedure calls.

**ANSA** regards separation and heterogeneity in a distributed computing environment as the norm rather than the exception. It is assumed that all services in the system are remote. Co-location is an exception which allows optimisation. Common interfaces between processes are emphasized. In ANSA, remote services are shared by propagation of references to interfaces between clients and services. Each service is an "object" which encapsulates its data and its operations for manipulating that data. The relationship between objects is a client-server interaction: The client object must request a service from a server object. ANSA provides both synchronous and asynchronous service requests. In the synchronous case, the local execution is blocked until the remote activity is completed whereas the local execution is not blocked in the asynchronous case. More information about ANSA can be found in [68, 4, 5, 6].

**PATHWAY** is a group of related software tools that enable development, installation, and management of online transaction processing applications on Tandem computers [94]. The applications are composed of requester programs and server programs. Requester programs are screen programs that control what is shown on the screen and accept input from the user. Screen programs (written using SCREEN COBOL) may call other screen programs in order to implement a hierarchy of screens or menus. The screen programs send data to and receive results from server programs. The server programs typically access databases on request from screen programs. Terminal control programs (TCP) control terminals by executing SCREEN COBOL programs. A TCP runs in a separate process. When database services are needed, the TCP sends a request to a process called PATHMON which provides a link to a server program which can perform the database access. There is only one PATHMON process in each PATHWAY system whereas there can be several TCPs and several replications of servers. It is important to note that requester programs are executed by a TCP, while groups of server programs are run as separate processes.
2.4 Software Components

Software components are the building blocks during development, configuration and maintenance of a computer system. The allocation of software components to processes defines which activities in a system that may run in parallel.

Classification of Software Components

The following types of software component are always present in a distributed transaction-processing system:

- Application components
- User-interface components
- Data storage components
- Distribution infrastructure components
- System administration components

Encapsulation

Software components are the building blocks of the software of most computer systems. They provide the means for structuring of the software and enable separation of concerns. Components should be as independent as possible with well-defined interfaces. Preferably, the components of the software should be encapsulated. All the dependencies between the components should be defined by the interfaces. This is the objective of modular programming and object-oriented programming. In addition, software configuration management requires well-defined and independent software components. Encapsulation is achieved by maximising component cohesion and minimising component coupling:

Cohesion is a measure of the strength of functional association of the pieces of code in a component. The highest level of cohesion is achieved when every piece of code within a component contributes to performing one single function and when the function is completely performed by the component. The other extreme is coincidental cohesion where the code is divided into components by chance. Usually system developers strive for high cohesion.

Coupling is a measure of interconnection among components in a software structure. It depends on the interface complexity between components. Coupling ranges from low coupling which forbid components access to data areas of other components, i.e., all data have to be passed as parameters, to high coupling where components can
access internal data in any other component. To achieve component independence system developers strive towards low coupling.

The primary rationale for dividing the software into components is to preserve locality of change when the software is changed. That implies that typical or anticipated changes should be described prior to definition of software components.

State-Dependent Operations on Components

A software component may contain data which describe the state of the component (e.g., [55, 35]). The execution of the operations of such components may depend on the current state of the component, e.g., overflow of indexed files. Hence, the invocation of an operation may influence the execution of following invocations of the same operation or following invocations of other operations that the component offers.

An example of a component with state-dependent operations is a relational database management system (DBMS) which provides cursors. A cursor describes a selection of data in a database. The cursor is defined by the a database query but instead of returning all the selected data in one chunk, the data can be fetched by repeated calls to the DBMS. Each time data are fetched by means of a cursor, the cursor is updated so that new data are read the next time a fetch call is issued.

2.5 Remote Procedure Calls (RPC)

Remote procedure calls (RPC) are the "glue" in distributed systems. Therefore, a closer look at RPC may be worthwhile. In a client-server system, processing is distributed among client processes which request services from shared processes called servers. Normally, client and server processes run in different CPUs, possibly of different types. The client processes invoke procedures or functions in the server processes as if they were running in the client process. The remote procedure protocol provides (nearly) transparent invocation of functions in other processes.

Tanenbaum [97] presents ten steps that are required to execute a remote procedure call:

1. Client program calls the stub procedure within its own address space. Parameters are passed as normal.

2. The client stub packs the parameters from the client into a message (called parameter marshalling) and passes the message to the communication infrastructure.
3. The communication infrastructure (LAN, MAN, WAN) passes the message along to the destination.

4. The communication infrastructure at the destination passes the message to the server stub.

5. The server stub unpacks the message and passes the parameters to the server.

6. The server performs the functions it is expected to perform according to the parameters and sends the results to the server stub.

7. The server stub packs the results into a message that is passed to the communication infrastructure.

8. The communication infrastructure passes the message to the origin of the remote procedure call.

9. The message is passed to the client stub.

10. The client stub unpacks the results and passes them to the client just as if an ordinary procedure call returns the results.

There are many different modes of operation of the remote procedure call protocol: Synchronous, asynchronous (nonblocking, callback), broadcast, batch [35]. The details of remote procedure calls will not be pursued further.
2.6 Summary

This chapter has presented the basic properties of distributed transaction-processing computer systems. Such systems typically consists of a lot of specialised processes which co-operate by sending messages to each other or by invoking each other. The processes are spread among the available processes in the distributed system. The next chapter will present the case studies which fit into the framework established by this chapter.
Chapter 3

Presentation of the Case Studies

Three practical case studies were performed in order to exploit composite work modelling. The computer systems that were examined during the case studies are all distributed systems (in somewhat different senses). The application part of these systems consists of distributed co-operating processes. The hospital case study involved a loosely-connected parallel Tandem computer, whereas the International Computers Limited (ICL) case study involved a distributed heterogeneous computer system. A small client-server system was implemented using remote procedure calls on SUN computers in order to provide a "test-bed" for examination of distributed systems. The hospital and ICL systems are both built according to software architectures which promote exchange and reuse of software components. These three computer systems provide a basis for assessment of the composite work modelling approach. The client-server example will be used as the running example in the thesis, while examples are drawn from the two other case studies when it is appropriate.

3.1 The Client-Server Example

The application in the example is a menu-driven banking system. The system is run as one file server and one or more bank application clients. The clients and the server can run on different hardware platforms which are connected to a common communication network. The clients communicate with the server by issuing remote procedure calls. The clients compete for access to the server. The bank client-server system is illustrated in Figure 3.1.

Hardware Architecture

The example has been run on SUN SPARC ELC, SUN SPARC 2, and SUN 3/80. SUN 3 and SPARC have different instruction sets. Hence, recompilation is neces-
Figure 3.1: The example application as a client-server system

ecessary when the software is transported. SPARC ELC and SPARC 2 have the same instruction set, but the processing speed is different. Thus, if a client process is run on a SUN 3 and a server process is run on a SPARC 2, the system is distributed.

It is assumed that the hardware devices in the example provide the following services:

- A CPU offers an average machine instruction where the execution time can be characterised by the average number of clock cycles that are required. For a reduced instruction set computer (RISC) such as SUN SPARC, the average execution time of an instruction will be close to one clock cycle.
- The network offers two services: send a message and receive a message.
- A disk offers two services: read block from disk and write block to disk.

The User Interface

The user interface of the banking system is menu-oriented. The menu-hierarchy is shown in Figure 3.2. The letter on each edge indicates which letter the user has to type in order to select the desired path in the menu-hierarchy. The leaves in the hierarchy are screen forms in which the user must fill in the required information. When the user has requested a list of customers according to a sort criterion, he is allowed to browse screenfull by screenfull through the list, both forwards and backwards.
3.1. The Client-Server Example

Figure 3.2: The menu hierarchy in the client-server system

The Account Data

The information about accounts is stored in a “flat” file in the following format:

18  Smith  John   12000.00
19  Walker Johnnie 3000.00
20  Normann Ola   200.00

A 5-digit account number, a maximum 30-character long last name, a maximum 30-character long first name, a 9-character long balance.

Transactions offered by the Bank System

The transactions that are provided by the system involve one or a few system services. If more than one service is involved, the control is given back to the user
between the services to let him think before he proceeds. For instance, to update the balance of an account (i.e., deposit or withdrawal), it is necessary the read the account data first so that the user can confirm that the correct account was selected. If it wasn’t, the user has to read the data of another account. When the correct account has been selected, the user can type in the amount of money that is deposited or withdrawn. Here, it is assumed that a service does not involve the user before the service is completely finished.

The following services are provided by the banking system

- Create bank account
- Read bank account info
- Update bank account balance (does not entail a read)
- Delete bank account (does not entail a read)
- List bank customers sorted by name
- List bank customers sorted by account balance

The first call to a list operation of the file server reads all the account information, sorts it by means of quicksort, and writes the result to a temporary file. Then information about the first account is returned to the client. The following list calls will read information about the next account in the temporary file. When all the information has been fetched by the client, the temporary file is deleted.

Software Components

The bank system consists of application components and standardised libraries (see Figures 3.3 and 3.4): The bank application component calls functions that are provided by the remote procedure call library, the “Curses” library which provides screen handling functions, the standard C library, and the remote file server. The remote file server calls functions in the remote procedure call library, the C library, and the file access functions provided by the operating system. The remote procedure call library calls functions for communication over a network provided by the operating system. These overviews\(^1\) of the components demonstrate a potential for component-wise replacement of software. The functionality for each software component is listed in Appendix C.

\(^1\)The diagrams in Figures 3.3 and 3.4 are not composite work models
3.2 International Computers Limited (ICL)

ICL's (International Computers Limited) Distributed Application and Integration Services (DAIS) enable applications to be distributed between heterogeneous hardware platforms. The hardware platforms may range from PCs to mainframes. The software architecture is defined by the Advanced Networked Systems Architecture (ANSA) which provides a transparent distributed system for applications. All software processes which run in one of the computers in the network, are encapsulated so that any "capsule" can interact with any other "capsule" in the distributed heterogeneous system.

The prototype at ICL is running a banking application. Figure 3.5 shows the configuration: 4 PCs, 4 Unix servers from ICL's DRS range of Unix servers, and one SUN workstation. All the computers are connected to the local area network (LAN). All the computers run Unix except for the third PC from the left. That PC runs MSDOS. The hardware platforms in Figure 3.5 are annotated (by dashed lines) with names of
different processes which run in each of the platforms. There are two application processes, the teller application and the dealer application, which run on two different PCs. The database server process receives queries which are translated to subqueries which are passed to suitable database management systems (DBMS). The results of the subqueries are combined by the database server and the result is returned to the application process. There is one and only one Lisadmin process in each computer which runs a DBMS, either Ingres or Oracle. The Lisadmins are responsible for creation and identification of DBMS processes. There is also a Trader process which is responsible for keeping track of which services are provided by which processes. Thus, the Trader process can direct requests for a particular service to a process which can provide the requested service. In the following, some of the most important software components of the DAIS system are given a more thorough description. The operations provided by these components are listed in Appendix B.

![Diagram of DAIS system configuration at ICL](image)

**Figure 3.5: Configuration of a DAIS system at ICL**

**Application Component** Only the teller application was considered during the case study. The use of the teller application proceeds in two steps. First, the customer or the company (the application distinguishes between accounts for private customers and company customers) must be identified. Either the account number is provided by the user or the user provides a (partial) name. If a name is input by the user, a screen (or screens) of matching customers or companies pops up and the user picks one of them. Secondly, either withdraw, deposit, statement, or balance is chosen.
Terminal Handling Component  The terminal (screen) handling functions provided by the Ingres library were used.

Database Server Component  This component receives queries to the distributed heterogeneous database from the application components. The queries are formulated as “Conceptual SQL statements”. This SQL language is meant to be the common language for any database system (e.g., Ingres, Oracle, Sybase) which is part of the distributed heterogeneous database system. The fragmentation of the database is described in a file which the database server can access. When data are moved, e.g., to another computer in the network, the file which describes the fragmentation must be changed accordingly. Each query is parsed and transformed into subqueries; one subquery for each database that have to be accessed. When the database server receives the results of all the subqueries, the results are merged and returned to the application. The database server can run on any computer in the heterogeneous system (if the program can be transported to any computer in the network).

Lisadmin Component  One and only one Lisadmin process must run in every computer in the heterogeneous system that host one or more database management systems. It is responsible for starting up database management systems on “its” computer and giving them unique identifications. Each DBMS is “disguised” as a logical information server (LIS). The disguise is exactly the same for all DBMSs.

LIS Components  A logical information server (LIS) is a “disguised” database management system. A LIS has a standardised interface which receives (sub)queries in a textual form. The query is then parsed and evaluated on the fly (dynamic queries). Each type of database management system (e.g., Ingres, Oracle, Sybase) requires a different implementation of the LIS.

Trader  A trader process is run on one of the computers in the heterogeneous system. All services that are provided by any component in the system are “exported” to the trader component so that other components can ask, “import”, the trader about available services. The trader is the only component in the entire system which has a fixed address in the network so that all components can ask it about available services.

Remote Execution Protocol – REX  provides a simple service for process-to-process interactions across a network. It uses the remote procedure protocol with some extensions. Synchronous, reliable calls and asynchronous casts are the two types of interaction that are supported. REX must be used in conjunction with a message-passing service (MPS) that provides an end-to-end transport service.
Message Passing Services – MPS provides either connection-less or connection-oriented protocols for transport of messages from a process to another.

3.3 The Regional Hospital in Trondheim

The regional hospital in Trondheim (RiT), Norway, runs a computerised information system [21, 66] which automates and simplifies many important activities such as patient and laboratory administration. The software runs on a centralised fault-tolerant loosely coupled 6 CPU VLX Tandem computer. The system has 6 gigabytes of mirrored disk space which makes a total of 12 gigabytes. 400 terminals or PCs are connected to the Tandem computer\(^2\).

Tandem computers (see Figure 3.6) are based on doubling of both hardware and software to ensure non-stop functioning of the system. A processing module consists of a CPU, private memory, and interfaces to interprocess buses and to device controllers. The processing modules are independent computers. Each processing module has its own complete copy of the operating system. No controlling executive software is necessary to coordinate the operating system with other copies, making the processor module a completely autonomous unit. To take advantage of the multiprocessor architecture, a network operating system permits a user process in any processor to access resources belonging to any other processor in the same system. Note that when a process is started in a processor the operating system will never move the process to another processor to balance the workload.

The Applications at RiT The information system at RiT currently consists of three subsystems:

- PAS - Patient administration system
  - System for administration of patients visiting the hospital. It provides functions for appointment registration, waiting lists, resource planning, economy functions and lists, reports and overviews.
  - System for administration of patients staying one or more nights in the hospital. It provides functions for waiting lists, admission registration, diagnose registration and departure registration and lists and reports.

- NSK - Clinic laboratory system which supports ordering of laboratory tests and reporting of results. The system also supports the daily work in the laboratory e.g., by providing working lists for the analysts.

- NSM - Micro-biology laboratory system supports the analysis of microorganisms such as bacteria and virus in samples. The functions it provides to its users are basically the same as for the clinic laboratory system.

\(^2\)The configuration by the end of the case study
Common Features for the Applications  Each of these subsystems are collections of programs. The system is open-ended, i.e., new programs may be developed according to the standards of Tandem software (PATHWAY) and included in the collection of programs. All programs in the same subsystem share the same database. There are also "bridges" between the three subsystems to avoid multiple registrations of the same information and to simplify the practical use of the subsystems. There is a common security system for ensuring that only authorised personnel is granted access to the sensitive data stored in the databases. Functions are provided to the users as either online transactions or as batch jobs. Usually functions which concern one or a few patients are run as online transactions while report and statistics generation run as bath jobs. It is important to distinguish the two kinds of jobs because online transactions run at high priority while batch jobs run at very low priority.

PATHWAY is the name of the Tandem software environment which is used by the hospital information system. In Pathway, software is distributed on operating system processes which, in turn, are distributed on CPUs in order to execute. The processes cooperate by sending messages to each other. The most important software with
respect to performance runs in the processes which are described below. The batch jobs are different from online jobs because batch job execution can be postponed to a suitable time and its batch application process do all the work without any usage of TCPS and communication with user terminals. Usually batch jobs demand a lot more computer resources than online jobs.

**Terminal Control Programs (TCPs)** perform the following tasks:

- Display data entry forms on terminals running the application.
- Accept data entered at the terminals.
- Check data to eliminate most data entry errors.
- Pass data to programs that update the database; the application processes.
- Minimise data transferred to terminal by maintaining a buffer of what the user can see on the screen. Send only information which *change* what the user can see on the screen.

**Online Application Processes** perform these tasks:

- Read the transaction request from a TCP.
- Access the database (by sending messages to DBMS processes) to list, add, delete or update data and to perform calculations like totalling an order.
- Send a reply to the TCP with the results of these operations.

**Batch Application Processes** are involved in initiating and running batch jobs.

- The parameters of the batch job (e.g., when the job should start) are input by the user. This description of the batch job is put in the batch queue with a specific starting time.
- When the batch job is due, typically at night, an application process is created and the batch job is run by that process.
- When the job is finished the application process is no longer needed and the process dies.
3.3. The Regional Hospital in Trondheim

Communication processes are responsible for communication between the Tandem computer and its terminals:

- Receive information from TCP processes and forward it to the users’ terminals via the terminal communication line.
- Receive information from the users’ terminals via terminal communication link and forward it to the correct TCP process.

Tandem offers two different modes of communication between a Tandem computer and a terminal; asynchronous (ATP) and synchronous (SNA) communication.

Disk Processes are responsible for database management.

- Receive request for either read block or write block.
- Check if any locks obstruct the request. If yes, wait.
- Check if data can be found in the cache. If yes, no disk access.
- Perform necessary disk accesses.

nonstopSQL is an implementation of ANSI SQL which runs in Tandem computer systems [34]. It offers distributed data, distributed execution, distributed transactions with full location transparency. nonstopSQL is integrated with the Guardian 90 operating system in order to adopt the security provided by Guardian. The main components of nonstopSQL are the SQL compiler, the catalog manager, the SQL executor, the file system, and the disk processes. The responsibility for database accesses is divided between the file system and the disk processes. The file system is responsible for management of the physical schema, for opening of files and indexes, for partitioning, for sending requests to the appropriate disk processes, and for buffering the replies. Each disk volume is managed by a set of disk processes which have a common request queue and a shared buffer pool. They manage the disk space, the access paths, the locks, log records, and a main memory buffer pool of recently used blocks. The nonstopSQL is intended to replace the ENCOMPASS family of database products. The ENSCRIBE database record manager is an ENCOMPASS product that provides a record-at-a-time interface for programmers.
Chapter 4 presents an overview over contention modelling formalisms and their parameter needs. These parameter needs serve as requirements on the outputs of load and work models. Chapter 5 presents the most important properties of workload models and how such models are formulated. Work modelling is treated in detail in chapter 6. Relevant aspects of measurement of computer systems are presented in Chapter 7 with emphasis on techniques and tools for measurement of the workload of distributed systems. Chapter 8 presents tools which support measurement, workload modelling, and contention modelling as well as integrated environments for such tools. Finally, in Chapter 9, sizing of computer systems is given special attention since sizing is closely related to composite work modelling which is the topic of Part III.
Chapter 4

Contention Modelling

Contention models are built as an alternative to direct measurement of the performance of a computer system. Contention models are used in order to predict the performance of a non-existing computer system or to predict the performance consequences of changes to an existing computer system.

Contention models can be analysed by various techniques to discover contention effects for scarce computer resources. Before contention model analysis can proceed, all parameters of the model must be available. The parameters are made available by either estimation, measurement, or combinations of the two. The purpose of this chapter is to present some of the applicable performance analysis formalisms. This chapter is not meant to be a comprehensive survey of available techniques for performance model evaluation. Such a survey is beyond the scope of this thesis. However, it is important to survey the parameters that the different contention evaluation techniques need. Composite work models should be capable of calculating values for these parameters.

The contention models are discussed in the context of distributed systems, where simultaneous resource possession is one of the features. As will be explained, there are several cases where non-traditional modelling techniques must be used in order to deal satisfactorily with the features of such systems.

4.1 Expected Results of Performance Model Runs

There are two main classes of performance measures: system-oriented measures and user-oriented measures.

System-oriented performance measures describe how well the computer system resources are used over time. The utilisation of a device is the fraction of time that
the device is busy. The queue length for a device is the number of transactions or jobs that is awaiting service at the device. Usually, the queue includes the transaction or job that is being served. Queue lengths may be quoted as an average over time, as the maximum queue length in a time interval, or as the integral of queue lengths over time. Beware that average queue lengths that are measured over a long period, which includes both busy periods and quiet periods, may be misleading. The throughput of a device or a system of devices is the number of transactions or jobs that is completed per unit of time.

User-oriented performance measures describe how well the computer system performs each time the user sends a request to the system. The system-oriented measures cannot be used for this purpose. The response time is the most common user-oriented performance measure. The problem with this measure is that there exist several definitions. One definition of response time is the elapsed time starting when the user finishes his request and ending when the system begins to output the results. Another definition is similar, but the elapsed time ends when the all the results have been output.

If there are several classes of transactions or jobs, the measures are calculated or measured for each class. Note that all performance indices must be presented together with a description of the corresponding workload.

4.2 Contention for Computer Resources

Contention in a computer system is not confined to hardware devices. In distributed and multiprocessor systems, the co-operating processes are potential bottlenecks even if sufficient hardware resources are available. There may be competition for several kinds of resource:

- Hardware devices such as CPUs, disks, physical memory, and communication links
- Access to software servers which have a limited number of pseudo-concurrent threads, e.g., DBMS servers and I/O processes
- Locks on files, database tables or tuples
- Space in buffers and caches.

However, no software resource can be utilised without utilising a hardware resource at the same time, i.e., the use of a software server requires execution in a CPU. If not all of the needed resources are available, the transaction is forced to wait. Synchronisation may also force a transaction to wait. If the computational work is delegated to more than one process, all the involved processes must wait for the others to finish before the partial results can be combined.
4.3 Computer System Bottlenecks

It is unlikely that all computer system resources saturate at the same time. Bottlenecks occur when some computer resource in either software or hardware is fully utilised while the other computer resources are not fully utilised. If it is impossible to increase the utilisation of the other resources because of the fully utilised resource, this resource is called the bottleneck in the system. In a performance study it is important to detect the bottleneck in the system and investigate it thoroughly. The primary bottleneck is the current bottleneck in the computer system. It may also be interesting to determine the secondary bottlenecks, i.e., the bottleneck candidates if the primary bottleneck is removed. An interesting observation is that the bottleneck in a computer system may move to another computer resource when the computer system is changed, even if the change is a minor one and does not alter the functionality of the system. Also changes in the workload may move the bottleneck in the system. The moves of the bottleneck are often unpredictable. The moving-bottleneck problem strongly affects the reusability of contention models because they are based on certain assumptions about the bottlenecks in the system. Consequently, there is no such thing as a general performance model.

4.4 Different Contention Modelling Formalisms

Model-based performance evaluation is obviously only possible if the performance model can be transformed into the formalisms (e.g., simulation language, queueing network, Markov chains) used by one of the analysis techniques [15]. After all, the prime objective of performance modelling is to provide numerical estimates of performance. In modelling studies, the models should be as coarse as possible to minimise cost and time needed to perform the performance evaluation study. It is most important that the primary bottleneck is represented in the contention model. In addition, representations which make the contention model analytically tractable should be chosen if such a model is sufficiently accurate.

The parameter needs of three evaluation techniques will be presented here: discrete event simulation, queueing network and Petri net analyses. These techniques are among the most common techniques in practical use (see [3, Chapter 1] and [52]). Often, the choice of contention modelling formalism depends on the modelling assumptions that are made. Different formalisms require parameters in different forms.
4.5 Discrete Event Simulation

Principles of Discrete Event Simulation

A discrete-event simulation model is suitable for modelling of dynamic systems, such as computer systems, which are subject to a sequence of instantaneous events. These events are generated within the simulation model based on the actions associated with the previous events in the model. Hence, the actions associated with events in the model generate new events that will take place at specific times in the future. Events are assumed to be instantaneous while activities may have a significant duration. Descriptions of future events are stored in a future-event list. There are two operations on this list. Find the next event to take place in the simulation model, and store newly derived future events. Thus, there is no fixed time increment during the simulation. It is assumed that the state of the simulated system remains the same between one event ending and the next event beginning.

Typically, a discrete-event simulation model describes how resources provide service to a flow of entities, e.g., transactions in a computer system may be regarded as the entities which request services from computer resources like software servers and hardware devices. A simulation strategy must be chosen in order to design and implement a simulation model. Which strategy to choose is to some degree dependent on the system to be simulated. Evans [39] identifies three strategies: event-scheduling, activity-scanning, and process-interaction. The first two strategies will not be treated here. The third strategy is chosen here because it is appropriate for representation of distributed systems. According to the process-interaction strategy, the behaviour, resource requirements and the duration of activities of each type of entity is described separately. Each instance of an entity will have a copy of its description where the state of its execution is stored. Thus each instance of an entity has a life of its own. This modularisation simplifies the design and implementation of the simulation model for certain systems such as distributed computer systems.

A simulation model can be represented by different diagramming techniques. Pooley provides a brief survey of such techniques in [83]. In this thesis activity diagrams [18, 84] are chosen as the graphical representation of process interaction models.

Activity Diagrams

A very simple activity diagram is shown in Figure 4.1. It shows two transactions each of which is represented as a “start” node, a “hold” node, and a “termination” node. A “hold” node represents a delay which may be drawn from a parameterised distribution, e.g., exponential with mean 2 milliseconds. The continuous lines between the nodes denote control flow. By default, the direction of the control flow is downwards. Both transactions compete for the same resource, the CPU, which is represented as a “resource” node. Each transaction has to acquire the resource before
processing and release the resource after processing. If a resource is unavailable the transaction must queue for the resource until it becomes available. Acquisition and release of resources are shown as directed dashed lines. In Figure 4.2 the simple “resource” node of Figure 4.1 has been replaced by a “server” node which can represent a time-shared resource. This means that transactions do not have to wait until other transactions are finished with their processing. The processing rate experienced by each transaction is decreased according to the number of transactions that are using the time-shared resource simultaneously. A “server” that is used during a hold is shown as an undirected dashed line between the “hold” node and the “server” [9].

It is easy to represent simultaneous resource possession in activity diagrams as shown in Figure 4.4. Both transaction#1 and transaction#2 require both the CPU and
the Disk to execute\(^1\).

An activity diagram for a distributed transaction-oriented system may be based on one of at least two viewpoints. The first possible viewpoint is to regard the cooperating software processes as the entities to be modelled (see Figure 2.3). The main loop in the simulation model of a software server (Figure 4.5) is infinite. When it is not giving service, it waits for the next request for service. A software server can only provide a limited number of pseudo-concurrent threads. Thus, a thread must be acquired before requests for service can be accepted. Each thread is represented as an instance of the software server entity in Figure 4.5. These instances will compete for the CPU in order to execute. When a request arrives, a CPU must be available in order to receive the remote procedure call and to provide the requested service. It is not necessary to acquire the network to receive the call if it is a typical local area network (LAN), e.g., Ethernet. If the software server itself needs to call a software server, it must acquire the network before it can send the remote procedure call. Then the network and the CPU can be released. Then the control flow (in the simulation model) is transferred to the other software server (note the gap in the activity diagram). The thread of the software server will remain inactive until the remote procedure call returns. Thus, other threads, if any, can execute. When the control flow returns, the CPU must be acquired again in order to receive the response of the remote procedure call and to finish the service. Then the network must be acquired again in order to send the response for the remote procedure call. Finally, all acquired resources are released and the software server will accept the next pending request or wait if none is pending. A disadvantage of this system-centered viewpoint is that the entities may need to behave very differently depending on the transaction that is requesting service.

The second viewpoint is to represent each type of transaction as an entity (see Figure 2.5). According to this viewpoint a transaction may need service from several software servers and hardware devices. The servers and devices are regarded as resources:

- *time-shared* resources, e.g., a CPU. This kind of resource can be modelled as a "server" in the activity diagram.

\(^1\)Note that if there is only one Disk resource, there will be no time-sharing of the CPU
4.5. Discrete Event Simulation

Figure 4.5: Typical activity diagram of a software server

- exclusive resources which cannot be used by other transactions before they are explicitly released, e.g., disks, database locks, server threads. This kind of resource can be modelled as a “resource” in the activity diagram.

Advantages of the transaction-centered viewpoint are that transactions can be modelled in isolation (see Figure 4.3) and that it is trivial to remove or add representations of transactions. An activity diagram which represents one type of client-server transaction according to the transaction-centered viewpoint is depicted in Figure 4.6. This activity diagram represents the entire execution of one transaction which is carried out in different processes which may run on different hardware platforms. Note that there is no transfer of control as in Figure 4.5. The boxes in the diagram shows the steps that the transaction goes through. The two first steps are executed in the client, steps four to six are executed in the server while the two last steps are executed in the client after having returned from the remote procedure call. The directed dashed lines show how exclusive resources are acquired and released while the undirected dashed lines show how time-shared resources are used.
4.6 Queueing Network Models

Queueing network models are widely used as a means for predicting the performance of a computer system. Queueing network models can be solved analytically in some cases, otherwise they must be simulated or approximated. A lot of research goes into the search for new kinds of queueing network which can be solved analytically. The advantages of analytical solutions are that the performance indices of a queueing network model can be found by mathematical treatment and the models can be solved much faster than by simulation. Note that a queueing network model is system centered because focus is on the computer resources while the transactions or jobs are represented by the routing between the resources. Compare with activity diagrams (Section 4.5) where the transactions are in focus and resource usage is depicted by acquisition and release of resources.

A comprehensive survey of queueing network modelling is given by Lavenberg in [64]. A basic introduction to queueing network modelling is provided by Lazowska et al. in [65].
4.6. Queueing Network Models

Concepts and their Graphical Representation

The two most common concepts in a queueing network model are the queueing centre and the delay centre. A queueing centre (Figure 4.7a) models a resource which offers service to one transaction at a time while other transactions have to wait in a queue for the resource. There is also a kind of queueing centre which models a finite number of resources which share the same queue. The delay centre (Figure 4.7b) offers simultaneous service to an infinite number of transactions, i.e., there are no queueing delays. Delay centres and queueing centres with a common queue for several resources are examples of load-dependent queueing centres where the service rate depends on the queue lengths in the queueing centre.

![Figure 4.7: A queueing centre (a) and a delay centre (b)](image)

A queueing network model consists of a network of queueing centres and delay centres. When the parameters of all the transactions are regarded as drawn from the same distributions, a single class model suffices. However, different transactions may need different service times when they are serviced by a resource. This is modelled by multi-class queueing networks where each kind of transaction is represented by a workload class. A multi-class queueing network is shown in Figure 4.8.

![Figure 4.8: An example of a queueing network model with 2 workload classes](image)

If transactions that have completed service can be thought of as leaving the model and being replaced instantaneously by waiting transactions, a closed queueing network model may be appropriate. Closed networks are appropriate when there is a limit on concurrent transactions and that there are enough transactions to utilise the maximum concurrency all the time. This assumption also works well for batch jobs. On the other hand; if the transactions arrive and leave the system and the number
of jobs in the system is indetermined, the model is open. An important property of open models is that there is no limit on the number of concurrent transactions. Both open and closed workload classes may appear in the same queueing network model. In that case the model is mixed.

![Diagram](image)

**Figure 4.9: Extra features in queueing network models**

Additional concepts have been introduced in queueing network modelling (e.g., [90]) because modelling of recent computer systems requires modelling of synchronisation, locking, and simultaneous resource possession.

Four concepts have been introduced in order to allow simultaneous resource possession in queueing network models. An *allocate* node (Figure 4.9a) models acquisition of (a number of instances of) a resource. The available instances of a resource are modelled as a pool of tokens. If too few instances of the resource are available to a transaction, it is forced to wait in the queue that is associated with the allocate node. A *release* node (Figure 4.9b) models that a transaction gives back the resources it has acquired. A *create* node (Figure 4.9c) models that a transaction produces a number of instances of a resource while a *destroy* node (Figure 4.9d) models that some or all of the instances of a resource is consumed by the transaction.

Other nodes have been introduced in order to model synchronisation, etc. For further details, see [90].

**Stochastic versus Operational Modelling Approaches**

Queueing theory may be viewed from two different viewpoints: the stochastic view and the operational view. These viewpoints are discussed in Denning and Buzen [36], Buzen [23], and Lazowska et al. [65].

By considering distributions of inter-arrival and service times it is possible to calculate the joint probability distribution of queue lengths at every queueing centre in the model. All the common performance indices (utilisations, throughputs and response times) can be calculated based on this joint probability distribution. Initially, the stochastic viewpoint was based on the following assumptions [36]:

...
4.6. Queueing Network Models

- The system is modelled by a stationary stochastic process
- Jobs are stochastically independent
- Job steps from device to device follow a Markov Chain
- The system is in stochastic equilibrium
- The service time requirements at each device conform to an exponential distribution
- The system is ergodic, i.e., long-term time averages converge to the values computed for stochastic equilibrium

Such queueing network models have some particularly desirable features which simplifies the evaluation of the models. These models are called product-form queueing network models because the joint probability of queue lengths at all the queueing centres in the network is as follows:

\[ P(n_1, n_2, \ldots, n_M) = \frac{1}{G(N)} \prod_{i=1}^{M} f_i(n_i) \]

where \( f_i(n_i) \) is some function of the queue length at the \( i \)th queueing centre and \( G(N) \) is a normalising constant which is only dependent on the total number of jobs in the system. Since \( f_i(n_i) \) can be determined independently for all the queueing centres in the network, a product-form queueing network can be evaluated by first evaluating each queueing centre separately. Secondly, the joint probability can then be found according to the formula above. This evaluation procedure is illustrated in Figure 4.10.

A lot of research aims at extending the class of product-form models. Important (initial) contributions in this area are reported in Jackson [60] and Gordon and Newell [46]. A landmark paper on product-form queueing networks was published by Baskett, Chandy, Muntz and Palacios [12]. They extended the class of product-form networks as follows:

- Open and closed multi-class networks are allowed; also mixed networks.
- The same performance measures are obtained regardless of which of the following queueing disciplines that are used: First-come-first-served (FCFS), processor-sharing (PS), last-come-first-served preemptive-resume (LCFS-PR), or infinite server (a delay).
- All classes in an FCFS queueing centre must have exponential service times with identical means whereas the three other types of queueing centre can have service times drawn from almost any distribution\(^2\) for each class.

\(^2\)The distribution must have rational Laplace-transforms
Figure 4.10: The procedure for solving product-form queueing networks

- The service times at a queueing centre may depend on the queue length at that centre.
- Class transitions with fixed probability are allowed

The assumptions of the stochastic viewpoint are difficult to test by observing or measuring the computer system to be modelled. Even if many of these assumptions are not satisfied in real systems, the queueing network models produce reasonably accurate results. Thus, an attempt was made to find the relationships between operational measures that could be used as parameters in queueing network models instead of stochastic measures. This led to the following operational assumptions [36]:

- All quantities should be defined so as to be precisely measurable, and all assumptions stated so as to be directly testable. The validity of results should depend only on assumptions which can be tested by observing a real system for a finite period of time
4.6. Queueing Network Models

- The system must be flow balanced, i.e., the number of arrivals at a given device must be (almost) the same as the number of departures from that device during the observation period.

- The devices must be homogeneous, i.e., the routing of jobs must be independent of local queue lengths, and the mean time between service completions at a given device must not depend on the queue lengths of other devices.

According to [36] these assumptions lead to the same mathematical equations as the stochastic assumptions. However, information about for instance queue length distributions is lost. Therefore it is not possible to find, e.g., the fraction of time that all the disks in a system is busy at the same time, or to determine the probability of having more than 6 transactions waiting for access to a disk.

Unfortunately, the following features cannot be represented in a product-form or separable queueing network model: simultaneous resource possession, memory constraints, blocking, adaptive behaviour, process creation and synchronisation, high service time variability, priority scheduling, and response time distributions [65]. The reason is that these features create dependencies between the queueing centres.

The measurable quantities\(^3\) are [65]:

\[
\begin{align*}
T & \quad \text{length of an observation interval} \\
A_k & \quad \text{number of arrivals observed} \\
C_k & \quad \text{number of completions observed} \\
\lambda_k & \quad \text{arrival rate} \\
X_k & \quad \text{throughput} \\
B_k & \quad \text{busy time} \\
U_k & \quad \text{utilisation} \\
S_k & \quad \text{service requirement per visit} \\
N & \quad \text{customer population} \\
R & \quad \text{residence time} \\
Z & \quad \text{think time of a terminal user} \\
V_k & \quad \text{number of visits} \\
D_k & \quad \text{service demand}
\end{align*}
\]

The following relationships are regarded as fundamental laws:

\[
\begin{align*}
U_k &= X_k S_k = XD_k \quad \text{Utilisation Law} \\
N &= XR \quad \text{Little’s Law} \\
R &= \frac{N}{X} - Z \quad \text{Response Time Law} \\
X_k &= V_k X \quad \text{Forced Flow Law}
\end{align*}
\]

Additional relationships exist between the measurable quantities:

\(^3\)Subscript \(k\) refers to the \(k\)-th resource
\[ \lambda_k \equiv \frac{A_k}{T} \quad X_k \equiv \frac{C_k}{T} \quad U_k \equiv \frac{B_k}{T} \]

\[ S_k \equiv \frac{B_k}{C_k} = \frac{U_k T}{C_k} \quad V_k \equiv \frac{C_k}{C} \quad D_k \equiv V_k S_k = \frac{B_k}{C} = \frac{U_k T}{C} \]

It is often necessary to assume that the number of arrivals to a system equals the number of completions. This is called the "flow balance assumption".

\[ A = C \quad \Rightarrow \quad \lambda = X \quad \text{System flow balance} \]
\[ A_k = C_k \quad \Rightarrow \quad \lambda_k = X_k \quad \text{Service center flow balance} \]

It is important to know if the queueing network is in product-form because parameterisation is easier in such networks; only the value of the product \( S_k \cdot V_k = D_k \) is needed, not the individual values of \( S_k \) and \( V_k \). This implies that the actual routing of the jobs in the model does not influence the calculated performance indices!

**Parameters of Operational Queueing Network Models**

The parameters required for queueing network models are job descriptions, queueing centre descriptions, queueing disciplines for each queueing centre, visits counts and service times. These are all operational (i.e., measurable) parameters.

**Job descriptions** describe the types of job that appear in a queueing network model. It is common to choose one of the following three types of job:

- Jobs that arrive at the system, receive service and leave the system are described by an arrival rate for each job class
- A kind of job that is replaced immediately by a new job when it finishes can be described by the (average) number of concurrent jobs in the system at any time
- Jobs which involve some processing and then go idle for some before they re-enter processing can be described by the number of concurrent jobs (regardless of the state of the job) and by the idle time (or "think time")

**Queueing centre descriptions** describe the types of queueing centre that the queueing network consists of: a single resource with a queue, multiple resources which share a common queue, or a delay.

**Queueing disciplines** of the queueing centres in the network are most commonly one the following:

- First come, first served (FCFS)
- Round-Robin (RR) where each job that contends for a resource, gets access to the resource for a short pre-defined time (the time-slice) before access is granted to another job in a cyclic manner.
• Processor Sharing (PS) which is equivalent to Round-Robin when the
time-slide gets infinitesimal
• Last come, first served, preemptive resume (LCFS-PR)

Visit counts describe how many times each kind of job "visits" each of the queue-
ing centres before it completes

Service times describe how long time each kind of job needs service at a queueing
centre on each visit to the centre

Queueing Network Models of Distributed Systems

As discussed earlier, simultaneous resource possession is common in distributed
heterogeneous computer systems. Unfortunately, this cannot be represented in a
product-form queueing network and can therefore not be solved exactly. Different
approximations have been proposed by Jacobsen and Lazowska [61, "method of sur-
rrogates"], Heiderberger and Trivedi [50, 51], and Rolia and Sevcik [87]. In most cases,
however, the queueing networks which model simultaneous resource possession must
be simulated.

Another common feature of distributed systems is that a transaction may visit each
hardware device several times but with different service times. For instance, this is
ture for systems that are implemented using remote procedure calls. The service time
before the remote procedure call and the service time after the remote procedure call
has completed, must be distinguished. This can be implemented by class transitions
in the queueing network model. A queueing network model with class transitions
can still be in product form [12]. However, the number of classes in the model grows
dramatically.

![Queueing Network Model](image)

**Figure 4.11: A queueing network model of the client-server system**

A queueing network is shown in Figure 4.11. The model assumes that a transaction
requires service from the hardware devices in the following order: CPU#1, Network,
iteration between CPU\#2 and the Disk, then Network again, and finally CPU\#1. This corresponds to typical transaction behaviour in a client-server system. This model is awkward because the behaviour of a transaction is state dependent when it receives service in CPU\#1 and Network. Not only the next queueing centre is dependent on whether it has been to CPU\#2 yet. Also the service time may be state dependent in these two queueing centres. Class transitions are required to model a transaction in this model. The model is system-centered because the transactions are described by their routing between the queueing centres.

Figure 4.12: A partial queueing network model which takes simultaneous resource possession into account

Figure 4.12 shows a different queueing network model of the client-server transaction which models simultaneous resource possession explicitly. The transaction steps are the same as in the simulation model in Figure 4.6. Note that there are no queueing centres at all in the model. Queueing takes place when resources are allocated. Hence, the delay centres correspond to the hold nodes in Figure 4.6. Unfortunately, the model in Figure 4.12 does not take into account the fact that, e.g., CPUs may be time-shared. If time-sharing is needed, the model fragment in Figure 4.13 shows a possible approach. The CPU is reserved for a limited time slot. In order to complete the required service at the CPU, \( V \) time slots are required.

Another problem is how multiple classes should be modelled. There are at least two possibilities:

- the system-centered approach: each kind of transaction is assigned to one class which are routed among the same service centres, or
Figure 4.13: Explicit modelling of time-sharing of a CPU. The total resource demand on the CPU is \( V \) times the duration of the time slot

- the transaction-centered approach: each kind of transaction has its own single-class queueing network model. The queueing networks for all types of transaction allocate resources from the same pools of resource tokens.

The first alternative leads to messy models when the transactions (partly) use different software servers and different hardware devices, or use the same software servers and the same hardware devices in a different order. In such cases the second alternative is better. In fact, this alternative corresponds in most respects to the activity diagrams that were presented in the previous section.

### 4.7 Petri Nets

Petri nets are appropriate for modelling computer systems that are concurrent, asynchronous, distributed, parallel, nondeterministic and/or stochastic. The graphical notation of Petri nets is easy to understand. An important feature of such nets is the mathematical foundation which enables formal analysis of their properties. However, the set of state equations which describes a Petri net grows exponentially with the size of the net. Thus, a large Petri net must be simulated. Murata provides a thorough survey of Petri nets in [72]. The presentation of Petri nets in this thesis is based on Murata’s paper. In addition, various extensions of standard Petri nets are discussed in [3].

#### Concepts and their Graphical Representation

A Petri net is a directed, weighted, bipartite graph which consists of two kinds of node: *places* and *transitions*. Directed arcs can be drawn between a place and a transition in either direction. It is not possible to draw arcs between two places or between two transitions. The basic concepts of a Petri net graph are shown in Figure 4.14. In the graphical notation the places are drawn as circles while transitions
are drawn as bars or boxes. There are no limits on the number of incoming arcs to a node or outgoing arcs from a node. Each arc is labeled by its weight (a positive integer). A \( k \)-weighted arc can be interpreted as \( k \) parallel arcs. A marking (or state) is defined by assigning a non-negative integer to each place in the Petri net. A place is said to be marked with \( k \) tokens if the marking assigned number \( k \) to the place. In the graphical notation a token is marked as a dot within a place. A Petri net may have an initial marking which describes how tokens initially are assigned to places.

![Petri Net Diagram](image.png)

**Figure 4.14: The basic concepts in Petri nets**

Possible interpretations of input places, transitions, and output places are described in the following table.

<table>
<thead>
<tr>
<th>Input places</th>
<th>Transitions</th>
<th>Output places</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input data</td>
<td>Computation step</td>
<td>Output data</td>
</tr>
<tr>
<td>Resources needed</td>
<td>Task or job</td>
<td>Resources released</td>
</tr>
<tr>
<td>Buffers</td>
<td>Processor</td>
<td>Buffers</td>
</tr>
</tbody>
</table>

The formal definition of a Petri net is as follows: A Petri net is a 5-tuple, \( PN = (P, T, F, W, M_0) \) where:

\[
P = p_1, p_2, \ldots, p_m \text{ is a finite set of places,}
T = t_1, t_2, \ldots, t_n \text{ is a finite set of transitions,}
F \subseteq (P \times T) \cup (T \times P) \text{ is a set of arcs (flow relation),}
W : F \to 1, 2, 3, \ldots \text{ is a weight function,}
M_0 : P \to 0, 1, 2, 3, \ldots \text{ is the initial marking,}
P \cap T = \emptyset \text{ and } P \cup T \neq \emptyset.
\]

A Petri net structure \( N = (P, T, F, W) \) without any specific initial marking is denoted by \( N \). A Petri net with a given initial marking is denoted by \( (N, M_0) \).
The current marking of a Petri net describes its current state. The marking or state changes when one or more transitions in the net can fire (are enabled). The transition rules are:

1. A transition \( t \) is said to be \textit{enabled} if each input place \( p \) of \( t \) is marked with at least \( w(p,t) \) tokens, where \( w(p,t) \) is the weight of the arc from \( p \) to \( t \).

2. An enabled transition may or may not fire (depending on whether or not the event actually takes place).

3. A firing of an enabled transition \( t \) removes \( w(p,t) \) tokens from each input place \( p \) of \( t \), and adds \( w(t,p) \) tokens to each output place \( p \) of \( t \), where \( w(t,p) \) is the weight of the arc from \( t \) to \( p \).

Transitions without any input arcs are called \textit{source transitions}. They are unconditionally enabled and generate tokens when they fire. Transitions without any output arcs are called \textit{sink transitions}. Such transitions consume tokens without producing any new tokens. The capacity of a place may be limited. In that case the Petri net is called a \textit{finite capacity net}. If a place contains tokens which enable two or more transitions, the Petri net is nondeterministic.

A common extension to standard Petri nets is the \textit{inhibition arc}. An inhibitor arc connects a place to a transition. The arc is represented as a line terminating with a circle instead of an arrowhead at the transition. A transition with one or more inhibitor arcs can fire when each of all its normal input places contains at least \( w(p,t) \) tokens and none of the inhibitor arcs contains any tokens. When the transition fires, tokens are removed from input places and put into output places as normal. Petri nets extended with inhibitor arcs where the number of number of tokens in any of the places is not limited, have the computational power of a Turing machine. Thus any computable system can be modelled by such Petri nets.

**Petri Nets as Performance Models**

In order to model performance it is necessary to introduce time in the Petri nets. In the previous subsection, standard Petri nets were presented where the transition time of all transitions are assumed to take zero time; they are \textit{immediate}.

In \textit{stochastic Petri nets} (SPN), an exponentially distributed delay is associated with each transition. When a transition is enabled, its firing is delayed according to the exponential distribution with rate \( \lambda \). The rates associated with the transitions may be different. If several transitions are enabled simultaneously, the one with the shortest delay will fire first. According to the memoryless property of the exponential distribution, the delay before the next pending transition is the same whenever the Petri net is examined.
In order to reduce the state space of a stochastic Petri net, Generalised Stochastic Petri Nets (GSPN) have been introduced. In GSPN nets, it is possible to have both immediate and timed transitions (Figure 4.15a&b). Thus, the use of timed transitions can be restricted to those cases that are necessary for modelling purposes. Enabled immediate transitions will always fire before any timed transition. If there is more than one enabled immediate transition, the transition to fire must be chosen according to a probability distribution which is called a random switch [3] (Figure 4.15c). In this way routing probabilities can be introduced in a Petri net model.

A coloured Petri net is obtained if a colour is assigned to each token. In such Petri nets the behaviour of tokens can be different in the same sense as jobs in different classes in queueing networks can behave differently.

![Diagram of Petri nets](image)

**Figure 4.15:** Extensions to standard Petri nets. A timed transition is shown in a) and an immediate transition is shown in b). In c) a random switch is necessary to determine which immediate transition that should fire.

The basic results of the evaluation of a Petri net are the state probabilities. All the usual performance indices (queue lengths, utilisations, response times and throughputs) can be calculated from these state probabilities. The evaluation techniques for the different kinds of Petri net are outside the scope of this thesis. Mainly, the required parameters of Petri nets are of interest here.

### The Parameters of Petri Net Performance Models

The parameters of a Petri net can be classified as either contention model parameters or as workload parameters. The contention model parameters are:

- the Petri net structure $N = (P, T, F, W)$
- the initial marking of places which represent the number of available computer resources
The workload parameters are:

- the initial number of tokens in the places which represent a pool of transactions
- the firing rate which determine the delay time for each timed transition
- the probabilities of random switches if routing is modelled in the performance model

If coloured tokens are used, there will be a set of workload parameters for each colour.

**Petri Net Models of Distributed Systems**

Petri nets are suitable for modelling simultaneous resource possession and synchronisation. They are therefore appropriate for modelling distributed systems where the software is executed by co-operating processes. There is a problem in the generation of transactions in the model. There may be no limit on the simultaneous number of concurrent transactions in the real system. If a source transition is used for generating transactions the state space will be infinite. Therefore an analytic solution of the Petri net is not available. In some cases this problem is overcome by choosing a maximum number of concurrent transactions.

As an example, a transaction of a client-server system is modelled as a Petri net in Figure 4.16. The transaction steps are the same as in the simulation model in Figure 4.6. It is difficult to distinguish different types of transaction unless the tokens are coloured. The best way is probably to model the steps in the execution of transactions separately for each kind of transaction. The places which represent a pool of resources are then all that is shared between the transaction models.

Time-sharing of computer resources in a Petri net must be defined analogously to the queueing network fragment in Figure 4.13. That requires the introduction of routing probabilities (by transition switches) in the Petri net. Queueing disciplines are difficult to represent in Petri nets.
4.8 Performance Models of Software Processes

Contention for both software servers and hardware devices is common in distributed systems. It is less common in non-distributed systems where applications often run as independent processes. There is therefore the need for performance evaluation techniques which take both software servers and hardware devices into account. Such a method, called Method of Layers (MOL), has been developed by Rolia in [87]. Related work have been reported by Woodside et al. in [102, 73, 103]. They work on “Stochastic Rendezvous Network (SRN) Models” which represent hardware and software objects which execute concurrently and where there may be contention for software objects as well as hardware objects. The performance of such systems can be estimated based on SRN models.

Again, only the structure of the contention model and the corresponding parameter needs are of interest in the context of composite work modelling. The Method of Layers is chosen as an example in order to demonstrate the typical parameter needs when contention for software components as well as hardware components are modelled.
Method of Layers

In MOL, requests for service among software processes are described in a graph; a software contention model. Processes that do not request service from other processes are placed at the lowest level in the graph. The other processes are placed at higher levels in the graph. Processes which have identical statistical behaviour may be put into the same process group and be represented as a single process in the software contention model. It is assumed that there are no cycles in the graph, i.e., there should be no cyclic interactions between the processes such as recursive interactions where more than one software process is involved.

The average response times of processes at one level of the software model are estimated by viewing the processes at the next lower level as servers in a separable queueing network model. Processes that request service are considered as customer classes and those that provide service are represented as servers. This representation captures the possible queueing delay incurred by the processes requesting service if the serving process is busy doing work on behalf of another calling process. Each pair of adjacent levels in the hierarchy defines a submodel. One iteration of the evaluation algorithm consists of solution of all these submodels. The algorithm is iterated until the calculated process response times for two successive iterations are sufficiently close.

The assignment of software processes to hardware devices is represented in a device contention model. A queueing network model is derived from the device contention model in order to determine queueing delays at the hardware devices. In this model, each process in the system is represented as a customer and each hardware device as a server.

The results of the software and device contention models are combined by iteration to provide performance estimates for the entire computer system. A simple example of a software contention model and a device contention model are shown in Figure 4.17: The Application invokes a distributed File server. The Application component uses CPU#1 and the Network while the File server component uses CPU#2, the Network and the Disk.

In the simplest version of the Method of Layers it is assumed that all service times at the software processes are exponentially distributed and that each process only offer one operation. Rolia has proposed three extensions to the method. The first extension allows a process to do some post-processing after the results are sent back to the invoking process. The second extension allows processes to have multiple operations where each operation may have different service times. It is assumed that all the invocations are served first-come-first-served and that all the operations share the same queue. The third extension allows invocations to share two or more identical processes, i.e., the same service can be processed concurrently. In addition, some support for modelling of synchronisation in the software is provided.
### Input Parameters for the Method of Layers

The required parameters describe how many times superior software components invoke inferior software components and how many times each software component invokes operations on hardware devices. In addition, the service time for each hardware operation is needed. The main difference to the parameter needs of a closed queueing network model (Section 4.6) is that the visit counts for invocations between software components are needed.

The full list of required parameters is as follows:

- $G, K$: Set of groups and devices
- $L$: The number of software levels in the LGM hierarchy
- $G_n$: The set of groups at level $n$ of the hierarchy
- $N_g$: Population of group $g$
- $V_{g,h}$: The average number of visits from group $g$ to group $h$
  - Group $g$ is assumed to be one level higher than group $h$
- $V_{g,k}$: The average number of visits from group $g$ to device $k$
- $S_{g,k}$: The average service time of a visit by group $g$ at device $k$
- $Z_g$: The think time for group $g$
- $\psi_j$: The queueing discipline of group or device $j$

Note that visit counts are required both for software processes and hardware devices. The think time, $Z_g$, is assumed to have value zero for serving groups, i.e., only groups in the highest layer in a software contention model may have a think time. Additional parameters are needed for the extensions to the basic method. For instance, to model multiple operations on processes, visit counts and service times in terms of each operation must be given as input parameters.
4.9 Summary of Needed Parameters

All performance modelling formalisms and their evaluation techniques require information about the resource demands of the software in terms of operations on hardware devices and in some cases also in terms of operations on software components. Both static and dynamic information are needed. Workload models, which are presented in the next chapter, are needed to provide the contention models with this information. The static relationships between the resource demands of the software components are modelled in a composite work model. Chapter 14 demonstrates how composite work models can provide some of the required parameter values for the contention models that were shown in this chapter.
Chapter 5

Workload Modelling

A workload model describes the computer resource demands of the software when it is executed as well as how the users invoke the software. Workload characterisation is the quantitative description of a workload's characteristics. Every statement regarding the performance of a computer system must be accompanied by a description of the corresponding workload. Therefore, workload characterisation is fundamentally important in all performance evaluation problems and it is needed in order to design workload models [44].

Figure 5.1: Different kinds of workload model
The real workload is often not suitable for controlled and repeatable performance measurement or modelling experiments. Workload models are better suited for this purpose. Workload models can be classified as either natural or artificial. A natural workload model is made up of samples from the production workload. Otherwise, the workload model is artificial. Artificial workload models are classified as either executable or non-executable. Natural workload models and executable artificial workload models are run on the real computer whereas non-executable artificial workload models are applied as parameters to a contention model of the real computer. The different kinds of workload model are illustrated in Figure 5.1.

An artificial executable workload model is a piece of software which represents the real workload. This software is in most cases not usable for any other purpose than consuming predefined amounts of hardware resources in a predefined sequence. Such workload models are run on the real computer system while performance indices are measured. The software should be parameterised in order to allow for flexible workload modelling and reuse of the workload models. A typical parameter is the number of loop iterations. Sequence and concurrency should also be described by the workload model.

Artificial non-executable workload models are simply parameter values for a contention model of the computer system. The sequence and concurrency are represented in the contention model as routing probabilities and workload intensities respectively. The use of computer resources is represented as visit counts and service times.

Workload modelling for performance models is the main issue in this thesis. This chapter presents both the dynamic and static aspects of workload models. Chapter 6 gives special attention to the static description of the computer resource demands of the software. Application of engineering principles is made possible by structuring the resource demands according to a composite work model which resemble the multi-layered structure of the software.

Comprehensive presentations of workload modelling techniques can be found in [42] and [44]. A collection of relevant papers on workload modelling was published in [91]. A survey of workload characterisation techniques is provided in [32].

5.1 Properties of Workload Models

Ferrari presents eight properties of workload models in [42]. They are system independence, representativeness, flexibility, compactness, compatibility, reproducibility, simplicity of construction, and usage costs.
5.1. Properties of Workload Models

System Independence

A workload model is system independent if it can be transported from system to system while remaining sufficiently accurate. This property of a workload model is particularly important in heterogeneous computer systems where it is likely that software components are moved from one type of hardware to another. The parameters of workload models are usually system-dependent and the interactions between the parameters may be system-dependent as well. Separation of concerns is necessary in order to isolate the system-dependent features from the system-independent features. A workload may be modelled at various levels corresponding to the levels at which a computer system may be described [44]:

Physical level The workload model is based on the consumptions and consumption rates of the system's hardware devices and software resources.

Virtual level The workload model is based on consumptions and consumption rates of the system's logical resources such as high-level language statements, logical disk accesses, and demand for virtual memory.

Functional level The workload model is based on rates of invocation of functions at the end-user level. This level of workload modelling requires consideration of operations provided by the software.

Workload models at the physical level are normally system-dependent whereas workload models at the virtual or functional level potentially are system-independent. To obtain workload characterisations that are independent of the speed of hardware devices, the hardware demands should be found in terms of number of invocations of hardware device operations. Then, hardware demands can easily be calculated for similar hardware devices with different speeds.

System independence may be achieved by means of a layered model of execution of software. The software is described in terms of programming language statements and programming language statements are given hardware-dependent characterisations in terms of machine instructions. This division provides flexibility in work modelling when the software is re-compiled by using a different compiler, when the software is moved to a faster CPU of the same type as before, or when the software is moved to another kind of CPU. A classification of programming language statements is presented in [101]. Operand data types, operators, and operand locality are also classified. Unfortunately, the number of statement variants gets very large. The cost of characterising a program in terms of these statements and finding the machine instructions that are executed for each class of statement, is very high. In addition to this functional variety problem, there seems to be three other inherent problems of workload characterisation [55]. They are data dependencies, state dependencies and sequence dependencies. A more common approach is to calculate factors for CPU demand on different CPUs. Then, if the CPU demand is known for a program which is executed on a specific CPU, the corresponding CPU demand on another CPUs can be calculated by multiplication by the correct factor.
Representativeness

A workload model must be an accurate representation of the real workload, i.e., it must be representative. Ferrari presents four definitions of workload model \( (V) \) representativeness in [43] and [42]:

- \( W' \) is a perfect model of \( W \) if it demands the same computer resources in the same proportions as \( W \).
- \( W' \) is a perfect model of \( W \) if it demands the same computer resources at the same rates as \( W \).
- \( W' \) is a perfect model of \( W \) if it performs the same functions in the same proportions as \( W \).
- \( W' \) is a perfect model of \( W \) if it produces the same values of the performance indices \( P \) as \( W \) when running on the same system.

The three first definitions focus on finding a workload model \( W' \) which uses computer resources or software functions in the same way as the real workload \( W \) with respect to proportions or rates. These three definitions are independent of the workload's influence on the computer systems. This may be difficult to achieve because a workload model is normally more compact than the real workload it represents. Compact in this context means that the workload model contains less information than the real workload. To remedy this problem, Ferrari has proposed the performance-oriented criterion for representativeness where the focus is shifted towards the workload's influence on the system; refer to the fourth definition of representativeness above. The workload model and the real workload are regarded as equivalent with respect to a computer system \( S \) if they influence \( S \) in the same manner, i.e., the corresponding values of selected performance indices are equal.

The performance-oriented criterion of representativeness can be discussed in the context of different performance evaluation techniques such as in [98] where the influence of different workload parameters (job-classes, number of jobs, and service time distributions) on the results of queueing network analysis is investigated.

Other Properties of Workload Models

Flexibility is the possibility of easily and inexpensively modifying a workload model to reflect variations in the real workload. A flexible workload model must also allow changes to reflect a reconfiguration of the computer system. The characteristic of compactness is related to degree of detail and hence to the representativeness and usage cost of the workload model. A compact workload model cannot reproduce the exact behaviour of the workload. If a performance-oriented validation procedure is followed, a compact model may be just as representative as a detailed model.
5.2. The Steps in Workload Modelling

Compatibility with the system or the system model to be driven is needed to make a workload model usable. Sequence-dependent workload models must map to routing in the contention model. The simplicity of construction of a workload model includes the cost and complexity of gathering the information necessary to design it and to make it operational. Usage costs are closely related to the flexibility and compactness of the workload model.

5.2 The Steps in Workload Modelling

Heidelberger and Lavenberg have proposed a method for workload modelling in [52]:

1. Selection of the workload components to be characterised, e.g., transactions, functions, jobs, or job steps.
2. Selection of the parameters used to characterise a workload component. The parameters may be hardware resource demands, e.g., number of CPU instructions or CPU time, memory space demand, or number of disk accesses. The parameters of a workload component may also describe software resource demands, e.g., number of calls to communication servers or DBMS servers.
3. Workload measurement. The real workload is measured in order to obtain the parameter values for each workload component.
4. Exploratory data analysis. Distributions of parameter values of work models are investigated. Patterns in the parameter values are searched for and outliers are carefully considered.
5. Reduction of workload model size (called cluster analysis in [52]). The rationale for this step is to decrease the size of the workload model by collapsing workload components which have similar parameter values. Cluster analysis is one of the most important techniques that is used for this purpose.

Steps 1, 2 and 5 are discussed in the following sections while step 3 is discussed in Chapters 7 and 13 and step 4 is discussed briefly in Chapter 13.

5.3 Selection of Workload Components

The selection of workload components depends on the system under investigation. The level of detail in selecting workload components determines the level of detail of performance estimates produced by contention model evaluations. For instance, if transactions are chosen as the workload components, then the contention model results will be in terms of transactions.
Figure 5.2: Various phases for the construction of static and dynamic workload models (from [27])

*Static* information about a workload is independent of any information about time. The number of times each workload component is used for a specific purpose within a specific period of time, is captured. In addition the sequence of arrivals of workload components is regarded as static information when the actual arrival times are not considered. *Dynamic* information can be the arrival times or rates of use of workload components. The inter-arrival times are often described by a statistical distribution, e.g., the exponential distribution. The mix of workload components should be properly represented in the workload model. Figure 5.2 taken from [27] illustrates that static workload modelling corresponds to characterising the workload components independently of each other whereas dynamic modelling corresponds to characterising temporal dependencies between the workload components.

Temporal dependencies *between* workload components must be sorted out at this stage of workload modelling because these dependencies may influence the selection of workload model components. The mix of concurrent workload components should be preserved in the contention model (Chapter 4) because contention delays occur when competing jobs try to use the same computer resources at the same time. One approach is to select users as workload components instead of transactions [29]. The behaviour of the each user is described by a state-transition diagram. In this way the mix of jobs can be controlled because each user at any time has at most one job being processed. User behaviour graphs [41] have been proposed as the means for such modelling. The behaviour of each user is modelled by a probabilistic graph as in Figure 5.3. Each node in the graph represents an interactive operation which can be invoked by the user. In each graph there is a special node called the "dormant node" which indicates that the user is not using the computer system. The arcs between the nodes are annotated with probabilities. The user behaviour is transformed to transitions between nodes, i.e., operations, according to the probabilities. A user behaviour graph can be modelled as a discrete time Markov chain. The example in Figure 5.3 shows a user behaviour graph for a banking system. The graph refers to three operations which are provided by the system: create_account, read_account
and `update_account`. The edges are annotated by transition probabilities. According to the graph, the user selects `create_account` with probability 0.05 and `read_account` with probability 0.95. 70% of the invocations of `read_account` result in an `update_account`. Another important application of user behaviour graphs is characterisation of how the user-interface of a computer system is used. The resource demands of sophisticated user-interface software such as X-windows, may be significant and therefore there is a need for explicit modelling of user behaviour.

![User Behaviour Graph](image)

**Figure 5.3: An example of a user behavior graph for the client-server example**

Ferrari uses another approach to workload mix representation in [43]: The sequence of the users' interactions with the system are divided into subsequences each of which consists of a fixed number of interactions. Each of these subsequences are then analysed and given a numerical characterisation. The main idea is to model snapshots of the workload mix.

It is possible to select workload components at different levels of detail to obtain a hierarchy of workload components. A multi-level workload model allows different levels of detail to be combined in one model. Hence, the workload model can be used for contention models of different level of detail. Such hierarchical arrangement of workload components has been presented by Haring [49], Calzarossa, Haring, and Serazzi [25] and Beilner [14, 15]. Hughes [56, 58] describes the relationship between a structural system model, concurrency, and a dynamic system model is shown in Figure 5.4. Here the concurrency is separated from the structural system model. An important point is that the dynamic system model has the same external dependencies as the system performance specification. Therefore the division of the structural and dynamic aspects of a performance specification can be applied recursively.
5.4 Selection of Workload Parameters

Only parameters which significantly influence the performance of the computer system should be chosen. In [42], this requirement is stated as follows with respect to the performance-oriented criterion for workload model representativeness (see Section 5.1):

For a given system \( S \), a given set \( \mathcal{P} \) of performance indices, and a given real workload \( W \), those and only those parameters of \( W \) which have a non-negligible influence on some element of \( \mathcal{P} \) should be included in \( W' \), a model of \( W \).

Workload parameters can be found that describe the entire workload without any regard to workload components in order to obtain a compact workload model. However, such a workload model would be system-dependent and inflexible. A more detailed workload model defines workload parameters for each individual workload component. An even finer level of granularity of workload model parameters is obtained by finding workload parameters for each job step for each workload component.

The workload parameters should depend on the computer system's organisation, on the applications, on the operating system, on the types of users, and on the application mix. It must be easy to find the parameter values which describes a particular computer installation. In order to minimise the number of parameters for each workload component, the correlation between the parameters should be examined. Only
one parameter from a set of strongly correlated parameters is needed. Typical types of parameter for workload components are the following:

**Resource Demands** The resource demands of each workload component must be represented as parameters. Normally, the resource usage is described in terms of use of hardware devices such as CPUs, disks, communication networks, and physical memory. The required parameters are a visit count, \( V_k \) for each hardware device \( k \) and the service time, \( S_k \), for each hardware device. The service time may be drawn from a statistical distribution or it can be the mean value. Depending on the properties of the contention model, the product of \( V_k \) and \( S_k \) is sufficient in some cases. Then only the service demand, \( D_k = V_k \cdot S_k \), for each device \( k \) is needed as parameter.

**The Routing Probabilities** In some contention models, the routing probabilities must be specified. They can either be found directly by measurement or derived from the visit counts.

**Intensities** Contention depends on the level of concurrency which in turn depends on the workload intensities of the workload components. In [65] intensities are characterised as either transaction, batch, or terminal:

- A *transaction* workload component has its intensity specified by a parameter \( \lambda \), indicating the rate at which jobs arrive. A transaction workload has a population that varies over time and there is no limit on the population. Jobs that have completed service leave the model.

- A *batch* workload component has its intensity specified by a parameter \( N \), indicating the average number of active jobs. A batch workload has a fixed population. Jobs that have completed service can be thought of as leaving the model and being replaced instantaneously from a backlog of waiting jobs.

- A *terminal* workload component has its intensity specified by two parameters: \( N \), indicating the number of active terminals, and \( Z \), indicating the average length of time that users use terminals ("think") between interactions.

It is often difficult to determine the think time because it depends on human behaviour.

The intensities of the workload components may vary in time [26]. Then time intervals must be defined where the intensities can be regarded as constant and the performance model must be run for each interval. It is also common to focus on intensities that occur during the "busy hour", i.e., at times with maximum usage of the computer system. The time-variation for each workload component may be different. Therefore it may be difficult to find time intervals where the intensities
for all workload components can be regarded as constant for modelling purposes. In [26] it is proposed that the time-varying intensity for each workload component is fitted to a polynomial with 8 coefficients. Then a cluster analysis is performed based on the values of these 8 coefficients for each workload component. The main idea is that workload components with similar time-varying behaviour have approximately the same 8-tuple of coefficients and can therefore be treated together.

5.5 Reduction of Workload Model Size

Workload components are distinguished by the values of their parameters. It is important to reduce the number of different workload components so that the workload model can be applied to a contention model (e.g., queueing network models). Reduction of the size of a workload model is carried out when workload components have been measured and values for the parameters of workload components have been captured. The selected workload components in the reduced workload model are often called workload classes. Two approaches will be presented here: sampling and clustering analysis [30].

The basis for distribution sampling is the distribution of values for each parameter of a workload component. The distributions are determined by analysis of the measurement data. Values of the parameters of a few workload components are drawn from the distribution of each parameter. Thus, the parameter values of a few workload components are obtained. The drawback of this method is that the correlations between the parameters are ignored.

An alternative sampling method is real workload sampling. Workload components are sampled either at fixed points in time or on a “one-out-of-n” basis. Unfortunately, time sampling gives too much weight to workload components which run for a long time while the converse is true when count sampling is used. The advantage is that the correlations between the workload component parameters are preserved because real parameter values from the sampled workload components are used.

Joint probability distribution sampling remedy the correlation problem of simple distribution sampling. An n-dimensional parameter space for workload components is defined. The parameter space is divided into cells into which the measured parameter values are put. The number of workload components in each cell is then reduced proportionally to the number of measured of workload components in the cell. Another approach is to apply clustering techniques in each cell.

Cluster analysis partitions the set of measured workload components into a small number of groups according to some criteria. If the number of measured workload components is very large, it may be necessary to select samples from the measurement data as the input to cluster analysis. The basis for cluster analysis is a metric which represents the closeness between two workload components based on their parameter values. The parameter values must usually be scaled to make them compa-
rable. The clusters are calculated based on a clustering algorithm (e.g., [44]). During the calculation, the optimal number of clusters must be found. When the clusters are established, the representative workload components from each cluster must be selected. Clustering analysis as well as relevant statistical analysis is supported by the "Workload Analyser Tool" (WAT). This tool is presented in Section 8.2.

A problem concerning reduction of workload models is the loss of dynamic information. Information about arrival times, sequence of execution of workload components, and the mix composition is lost. In some cases, this information is not needed as shown by Ferrari in [41]. In other cases, the workload components can be selected so that important dynamic properties are retained in the workload model (as explained in Section 5.3).

5.6 Workload Modelling of Distributed Systems

A paper by Agrawala and Thareja [1] presents some interesting hypotheses that summarise some particular characteristics of the workload in distributed computer systems.

**Hypothesis #1:** The workload as viewed at the central machines serving workstations appears very similar to the workload of batch systems. Typically, the front-end machines will carry out the user-interface processing while the "heavy" jobs are passed to one of the servers in the system.

**Hypothesis #2:** The more friendly the user interface, the more traffic it is likely to generate over the network and more workload will be served by all systems.

**Hypothesis #3:** The classical techniques for mapping application functions to the system resource requirements are not applicable to the distributed system environments because processes in the system is likely to be migrated in order to balance the workload across all servers in the system.

**Hypothesis #4:** As the workload is redistributed by processing some of the workload at the workstation, the resource requirements at the central machine do not decrease by an amount equivalent to the processing capacity of the workstation.

**Hypothesis #5:** The changes in mode of operation always brings about unanticipated increases in the workload.

A consequence of distributed systems consisting of many PCs and/or personal workstations and a few servers is that the interactive part of the software will not cause any performance problems because such processing is done locally. However, more attention is needed on the common parts of the distributed system: the network and the servers.
5.7 Need for Engineering Principles

There is a need for modelling of the relationships between resource demands in order to decrease the long-term modelling costs and to shorten the time it takes to produce updated resource demands in workload models for new performance models. The current approach for capture of service demands corresponds to a one-layer computational work model. This kind of model does not preserve the static relationships between service demands which are described by the software. If some parts of the software are changed, all the service demands must be captured from scratch because the relationships between the demands are not represented in the computational work model. However, in a multi-level work model, such static relationships can be retained. A composite computational work model is used for this purpose. It consists of work model components which must not be confused with workload model components. If several workload components have some software in common, e.g., the remote procedure call library, the composite work model will show that the workload components have a work model component in common. Computational work modelling is the topic of the next chapter.
Chapter 6

Work Modelling

The rationale for work modelling is to provide a *static* representation of the computer resource demands of the software. Static means independent of contention. Sequence is not treated here although it may be considered a static aspect of work models. The motivation for distinguishing work models and workload models is that static aspects of the software are more stable than the dynamic aspects. It is assumed that the static aspects of a software component can be described independently of other software components. It is then possible that the description can be reused when some parts of the software is changed and it can be reused in other *similar* work models. Work modelling in terms of software components in multiple layers is called composite work modelling.

This chapter presents the *Structure and Performance (SP) specification method* [58, 71, 99, 100] which constitutes the basis for this thesis. The SP method defines rules for the formulation of composite computational\(^1\) work models. These rules are assumed to be independent of the computer system architecture and may therefore be used for modelling of any computer system. The practical use of SP is not described in [58, 71] and the existing documentation on practical applications of SP is limited. Guidelines for SP modelling and modelling trade-offs are presented in Part III. An example of the role of SP models is described in [21].

The principles and rules of SP are explained and demonstrated by going through a simplified version of the client-server example (Section 3.1). The example will culminate in the composite work model that is shown in Figure 6.1. The following presentation of SP will explain how this model obeys the SP rules. The operations provided by the various components in this model are listed in Table 6.1.

\(^1\) "Computational work" is frequently referred to as just "work".
Figure 6.1: An composite work model of the client-server application

6.1 Operations in an SP Model

An operation may be non-primitive or primitive. A non-primitive operation is implemented by an information process\(^2\). Such a process executes a set of interconnected operations determined by an algorithm. This definition of operations allows for a hierarchy of operations. An information process needs temporary storage space, or workspace, during execution of the operations of the process. The storage needed for storing data beyond the lifetime of the current process is called memory. Figure 6.2 depicts that the process is run in a work-space, \( W \), and requires access to persistent memory, \( M \).

Figure 6.2: An information processing operation in a work-space \( W \) (from [58])

Hughes [58] has introduced the notion of an abstract virtual machine which is based on a general view of information processing that is independent of any assumptions about computer system architecture or implementation. Figure 6.3 shows that an information process needs operations to manipulate and transform the contents of the

\(^2\)A process in this context means execution of a task in general, not necessarily an operating system process
<table>
<thead>
<tr>
<th>Component</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bank application</td>
<td>cre_acc</td>
<td>create account (client side)</td>
</tr>
<tr>
<td></td>
<td>upd_acc</td>
<td>update account (client side)</td>
</tr>
<tr>
<td></td>
<td>del_acc</td>
<td>delete account (client side)</td>
</tr>
<tr>
<td>User interface</td>
<td>refresh</td>
<td>clear screen</td>
</tr>
<tr>
<td></td>
<td>waddch</td>
<td>put a character on the screen</td>
</tr>
<tr>
<td></td>
<td>winsch</td>
<td>read a character from the keyboard</td>
</tr>
<tr>
<td>File server</td>
<td>fcrc_acc</td>
<td>create account (server side)</td>
</tr>
<tr>
<td></td>
<td>fread_acc</td>
<td>read account info (server side)</td>
</tr>
<tr>
<td></td>
<td>fupd_acc</td>
<td>update account info (server side)</td>
</tr>
<tr>
<td></td>
<td>fdel_acc</td>
<td>delete account (server side)</td>
</tr>
<tr>
<td>RPC library</td>
<td>clnt_call</td>
<td>send remote procedure call</td>
</tr>
<tr>
<td></td>
<td>svc_getargs</td>
<td>get arguments</td>
</tr>
<tr>
<td></td>
<td>svc_sendreply</td>
<td>send reply</td>
</tr>
<tr>
<td>File system</td>
<td>rpage</td>
<td>read file page</td>
</tr>
<tr>
<td></td>
<td>wpage</td>
<td>write file page</td>
</tr>
<tr>
<td>Disk</td>
<td>rblock</td>
<td>read disk block</td>
</tr>
<tr>
<td></td>
<td>wblock</td>
<td>write disk block</td>
</tr>
<tr>
<td>CPU#1/#2</td>
<td>instr</td>
<td>execute normalised instruction</td>
</tr>
<tr>
<td>Network</td>
<td>send_msg</td>
<td>send message</td>
</tr>
<tr>
<td></td>
<td>rec_msg</td>
<td>receive message</td>
</tr>
</tbody>
</table>

Table 6.1: Operations on the components in Figure 6.1

![Diagram](image-url)

Figure 6.3: The abstract virtual machine (from [58]). The small circles denote operations.
inner workspace. It uses certain operations for communication with other workspaces where another information process may be running concurrently. Persistent storage is accessed by means of memory operations. An information process also needs some means for selecting the next operation to be invoked; discrimination. Consequently, the entire repertoire of operations that is needed by an information process can be classified as *communication*, *processing*, *memory*, and *discrimination*.

It is important to understand that the three types of suboperation are distinguished by the persistence of the memory on which it operates:

- a *processing* suboperation operates on data that will disappear when its superior operation is finished.

- a *communication* suboperation operates on data which was passed to its superior operation. Therefore the data may exist after the superior operation has finished.

- a *memory* suboperation retrieves data from and stores data in a permanent store.

According to the SP method, which is based on the abstract virtual machine, all relationships between components and subcomponents must be categorised as either *communication*, *processing*, or *memory*. *Communication* and *memory* relationships between a component and its subcomponents implies that the component delegates computational work. *Processing*, however, is a rather different kind of relationship. A processing subcomponent is internal to the component itself. If it is not a primitive component it is an intermediate representation of processing in order to achieve system independence. It is possible for a subcomponent to offer operations which belong to more than one category, e.g., a component may offer both processing and memory operations.

**Communications**

In distributed systems, operations may execute in disjunct workspaces. That implies that acquisition of the inputs and “publication” of the outputs of an operation require execution of explicit suboperations (such as network commands) in order to perform the actual data transfer. These communication suboperations are not required in non-distributed systems because all the operations in the system receive inputs and return outputs to the same workspace; the stack. As an example, consider the remote procedure call (RPC) library (see Section 2.5). The RPC library is only necessary when an operating system process invokes another operating system process which runs in a different computer, i.e., when a network is involved. Thus, the operations provided by the RPC library are typically *communication* operations according to the SP terminology. Note that the means by which the computer system communicates with its users (human beings) may be regarded as *communication* in the SP sense.
6.1. Operations in an SP Model

The human being has his own "work space", in the brain. The user interface of the computer system is used as the means by which the human and the computer system can communicate.

**Memory Access**

The workspace or the context of an operation is volatile. Some of the data that an operation needs must be fetched from and stored in a persistent storage outside the workspace of the operation. The suboperations which are invoked to fetch inputs and store results permanently\(^3\) are classified as *memory* suboperations.

**Transformation/Processing**

The processing part of a component may be described by *processing* subcomponents. This subcomponent must provide a complete repertoire for processing. If the processing is distributed over several CPUs, one *processing* subcomponent per CPU is required.

![Diagram](image)

**Figure 6.4: Different levels of processing in an SP model**

In order to enable hardware-independent characterisation of processing demands, it is necessary to describe processing in terms of high-level statements of a programming language. Consequently, hardware-dependent characterisations of programming language statements in terms of machine instructions are needed as well. This is represented as a three-layered model as in Figure 6.4. The component at the top describes the execution of an application program in terms of basic statements in a third generation programming language, e.g., assignment of character pointers in C. The next level describes the execution of basic statements of a third generation

\(^3\)Note that "permanent" is also a relative description of duration
programming language in terms of machine instructions. The last level is a primitive component on which the work on the CPU is devolved.

The Control Process

The control process is necessary in order to determine when and in which order the suboperations should be invoked by the information process. Hence it is the control process that connects the suboperations of a component so that the component can deliver results to its superior components. The computational work that is required in order to determine the next suboperation to invoke is called discrimination. The evaluation of the conditions for a certain control flow in the component is not regarded as discrimination but as processing. Only the act of changing the control flow is regarded as discrimination. In practice, discrimination is not treated separately but put together with processing. Note that a complete decomposition of the components is a composite work model will result in discriminations only. Consequently the discriminations constitute the atomic units of information processing. This is analogous to computing which eventually is reduced to logical gate switching in the electronic devices in the hardware; the electronic discriminations.

6.2 The Structure of an SP Model

Graphical Representation of SP Concepts

In a component-subcomponent relationship, the invoking component is termed the superior component and the component being invoked is termed the inferior component. These concepts and their graphical representation are depicted in Figure 6.5. Operations on subcomponents are called suboperations. Thus, a component-subcomponent relationship consists of a number of corresponding operation-suboperation relationships. The letter codes on the edges are defined by the rules of the SP method. P means that a subcomponent is used for processing, C means that a subcomponent enables communication and, consequently, distribution, while M means persistent memory. An operation-suboperation path in an SP model is a continuous chain of operation-suboperation relationships from a higher-level operation to an operation on a lower-level component. Similarly, continuous chains of component-subcomponent relationships are called component-subcomponent paths.

Abstract Data Types

In the previous section, operations were treated as separate entities. The operations may be grouped according to which data they use. A group of operations on a
common data structure forms a structural entity called an abstract data type. Both software and hardware components can be viewed as abstract data types. Abstract data types represent the properties of software and hardware components which are independent of their actual implementation. A software component is represented as an implementation of an abstract data type on a virtual machine provided by subordinate components (Figure 6.6). In SP terms, a module relates an abstract data type to the virtual machine on which it is implemented. A module may have several alternative implementation specifications. An SP module specifies one or more data classes to which operations are attached. The parameters to be used in complexity functions must also be specified for each data class. There are two properties which represent implementation-dependent information: work complexity and compactness. Work complexity specifications describe how many time operations invoke other operations. Compactness specifications describe how the memory or storage units of components at higher software layers map to memory or storage units of components at lower software layers (e.g., records to blocks). Compactness will not be treated further in this thesis because only work complexity was considered during the case studies.
Work Complexity

A complexity function describes how many times a specific operation invokes a specific suboperation. Complexity functions may be parameterised in order to represent data-dependencies. The form of a complexity function describes how coefficients and parameters are related, e.g.,

\[ C(N) = k_1 N \log(k_2 N) + k_3 \]

where \( C(N) \) is the complexity function which depends on parameter \( N \) and where \( k_1, k_2, k_3 \) are coefficients. Ideally, the complexity functions should be independent of the load, i.e., concurrency. However, load-dependent parameters are needed in cases where extra computational work such as paging or swapping, is generated because of concurrent use of computer resources. Thus, load-dependent parameters may have to be introduced in some of the complexity functions.

All the complexity functions which describe the relationship between operations of a superior component and the operations of an inferior component can be arranged in a complexity matrix. Each row in such a matrix represents an operation while each column represents a suboperation. As an example, consider the work model components in Figure 6.1. There are 15 relationships between these components. A complexity function can be defined for each relationship between work model components. If it is assumed that the two RPC library components have identical implementations, there are 13 unique complexity functions. They are listed below. The constants in the complexity matrices in the example are "fakes" in order to keep the example simple. More realistic complexity matrices of the client-server system can be found in Chapter 12. \( C_{\text{component}} \) identifies a complexity matrix for the relationship between a component and one of its subcomponents. The operations are listed in Table 6.1.
6.2. The Structure of an SP Model

\[
\begin{align*}
C_{\text{User Interface}}^{\text{Bank application}} &= \begin{bmatrix}
\text{refresh} & \text{waddch} & \text{winsch} \\
\text{cre.acc} & 3 & 300 & 40 \\
\text{upd.acc} & 3 & 400 & 10 \\
\text{del.acc} & 3 & 300 & 10
\end{bmatrix} \\
C_{\text{CPU}}^{\text{Bank application}} &= \begin{bmatrix}
\text{cre.acc} & 10000 \\
\text{upd.acc} & 9000 \\
\text{del.acc} & 10000
\end{bmatrix} \\
C_{\text{RPC library}}^{\text{Bank application}} &= \begin{bmatrix}
\text{clnt_call} & \text{svc_getargs} & \text{svc_sendreply} \\
\text{cre.acc} & 1 & 0 & 0 \\
\text{upd.acc} & 1 & 0 & 0 \\
\text{del.acc} & 1 & 0 & 0
\end{bmatrix} \\
C_{\text{File server}}^{\text{Bank application}} &= \begin{bmatrix}
\text{fcree acc} & \text{fread acc} & \text{fupd acc} & \text{fdel acc} \\
\text{cre.acc} & 1 & 0 & 0 & 0 \\
\text{upd.acc} & 0 & 1 & 1 & 0 \\
\text{del.acc} & 0 & 0 & 0 & 1
\end{bmatrix} \\
C_{\text{RPC library}}^{\text{File server}} &= \begin{bmatrix}
\text{clnt_call} & \text{svc_getargs} & \text{svc_sendreply} \\
\text{fcree acc} & 1 & 1 \\
\text{fread acc} & 0 & 1 & 1 \\
\text{fupd acc} & 0 & 1 & 1 \\
\text{fdel acc} & 0 & 1 & 1
\end{bmatrix} \\
C_{\text{CPU}}^{\text{File server}} &= \begin{bmatrix}
\text{instr} \\
\text{fcree acc} & 11000 \\
\text{fread acc} & 8000 \\
\text{fupd acc} & 9000 \\
\text{fdel acc} & 10000
\end{bmatrix}
\]
\[
C_{\text{File\_server}} = \begin{bmatrix}
\text{fcre\_acc} & 0 & 1 \\
\text{fread\_acc} & 1 & 0 \\
\text{fupd\_acc} & 0 & 1 \\
\text{fdel\_acc} & 0 & 1 \\
\end{bmatrix}
\]

\[
C_{\text{User\_interface}(P)}_{\text{CPU}} = \begin{bmatrix}
\text{refresh} & \text{instr} \\
\text{waddch} & 100 \\
\text{winsch} & 70 \\
\end{bmatrix}
\]

\[
C_{\text{User\_interface}(M)}_{\text{CPU}} = \begin{bmatrix}
\text{refresh} & \text{instr} \\
\text{waddch} & 50 \\
\text{winsch} & 200 \\
\end{bmatrix}
\]

\[
C_{\text{RPC\_library}}_{\text{CPU}} = \begin{bmatrix}
\text{clnt\_call} & \text{instr} \\
\text{svc\_getargs} & 1000 \\
\text{svc\_sendreply} & 300 \\
\end{bmatrix}
\]

\[
C_{\text{Network}}_{\text{RPC\_library}} = \begin{bmatrix}
\text{clnt\_call} & \text{send\_msg \ rec\_msg} \\
\text{svc\_getargs} & 1 \\
\text{svc\_sendreply} & 0 \\
\end{bmatrix}
\]

\[
C_{\text{File\_system}}_{\text{CPU}} = \begin{bmatrix}
\text{rpage} & \text{instr} \\
\text{wpage} & 100 \\
\end{bmatrix}
\]

\[
C_{\text{Disk}}_{\text{File\_system}} = \begin{bmatrix}
\text{rblock} & \text{wblock} \\
\text{rpage} & 1 \\
\text{wpage} & 1 \\
\end{bmatrix}
\]
6.3 Formulation of SP Models

The SP method defines two rules for formulation of well-formed SP models. These rules are quite general because they are based on the abstract virtual machine. Therefore, the rules of the SP method must be interpreted before it is possible to apply them in practice.

The SP rules force the person who is building the composite work model, to look for communication, processing, and memory relationships between components. Discrimination is normally included in the processing. The rules for formulation of SP models are stated as follows in [58]: In a well-structured system, modules are formed in such a way that

1. for any given non-primitive module, the operations of the associated ADT are all implemented by the same virtual machine, whose suboperations are grouped by type to form new ADTs.

2. the virtual machine is composed of ADTs whose relationships to the superior ADT correspond to the different types of suboperation of the corresponding abstract virtual machine.

The current SP tool [71] defines the following rules:

- Any non-primitive SP component must have at least one processing subcomponent and at least one memory subcomponent.

- If processing in a non-primitive SP component is distributed, there should be one processing subcomponent for each distributed processing unit, e.g., CPUs.

- If persistent storage for a non-primitive SP component is distributed, there should be one memory subcomponent for each distributed storage unit.

- If either processing or persistent memory (or both) of a non-primitive SP component is distributed, a communication subcomponent is required. Only one communication subcomponent for each non-primitive SP component is allowed.

- A non-primitive SP component cannot be a communication component if one of its ancestor components is a communication component.

Illegal SP model constructs should be detected by the tool and rejected. The feasibility of these rules which are enforced by the SP tool had not been properly tested for distributed systems before the three case studies (see Chapter 3) were carried out. These case studies are all about distributed systems. The rules of the SP tool turned out to be too strict for the distributed systems that were examined in the
case studies. There was a need to specify that a non-primitive SP component has a *communication* relationship with more than one subcomponent. In addition, it should be possible to have more than one *communication* link on any path between a top-level component and a primitive component.

![Diagram showing a file server, a sorting package, and CPU connected by links labeled P.]

**Figure 6.7: An illegal composite work model fragment if the links are of the same type**

The rules of the SP tool can in some cases restrict the granularity of composite work models. Figure 6.7 shows an illegal composite work model fragment where the links are of the same type. In SP, if two links of the same type originate from the same component it implies distribution. Hence, a chain of components which are connected by the same type of links cannot be short-circuited by a link of the same type at any stage. This is in conflict with the virtual machine principle where a virtual machine must provide all the operations that implement an abstract data type. It is therefore not possible to extract parts of a work model component and model it as an inferior work model component if the new component cannot offer all the operations within a link type that are invoked from the superior work model component. In the example in Figure 6.7 the sorting operations cannot be modelled explicitly.\(^4\)

Note that component-subcomponent relationships not necessarily define the *direction* of invocations. Normally the application components are defined as the top-level components in a composite work model while user interface components are defined as inferior to the application components. This modelling convention is followed even if it is the user interface components that actually invokes the application components. A mix of the operations on the top level components must correspond to work units (or workload components) in terms of which the performance of the system is quoted, e.g., a transaction. The user-interface components and other components provide services which enable the system to execute its work units, i.e., they are inferior to the application components.

\(^4\)However, it is feasible to provide tool support for such decompositions without breaking the rules of the SP method
Formulation of an SP Model in the Client-Server Example

The purpose of this section is to explain by examples how the SP formulation rules are applied in practice. The SP model in Figure 6.1 will be explained layer by layer.

![Diagram of bank application component](image1)

**Figure 6.8: Subcomponents of the bank application component**

_**First**, the bank application (client) component depends on a virtual machine which provides operations for managing the user-interface, for internal processing, for communication with the file server, and operations for storing data permanently (Figure 6.8). These operations provide the complete functionality which is required in order to run the bank application. The user-interface operations provide the means for communication with the user. The user has his own workspace (the brain) and the user-interface is the means of transfer of information between the workspace of the user and the workspace of the system. Therefore, these operations are classified as _communication_ operations in the SP terminology. The CPU provides the means for execution of the computational work that is not delegated to other components. Thus, these operations must be classified as _processing_ in the SP terminology. The operations of the remote procedure call library (RPC) provide the means for transfer of information between the workspace of the client and the workspace of the server. Hence, these operations must be classified as _communication_ operations. The file server operations provide permanent storage of data, and they are therefore classified as _memory_ in the SP terminology.

![Diagram of file server component](image2)

**Figure 6.9: Subcomponents of the file server component**

_**Secondly**, the file server component depends on a virtual machine which provides operations for communication with the bank application (client), for internal processing, and operations for storing data permanently (Figure 6.9). The RPC library provides operations for transfer of data between the workspace of the server and the workspace of the client. Therefore these operations are classified as _communication_
operations. The CPU provides the means for execution of the non-delegated work of the file server. Hence, it provides processing operations. The file system component provides operations which support permanent storage of data which are classified as memory.

![Diagram](image)

**Figure 6.10: Subcomponents of the user interface component**

Thirdly, the user-interface component depends on a virtual machine which provides execution facilities and operations on the screen buffer\(^5\) (Figure 6.10). Here it was decided that the screen buffer should not be defined as a separate hardware device. Instead it was regarded as a part of the CPU. Consequently, the operations of the CPU component provide all the operations that are required by the user-interface component. The processing operations provide execution in the CPU while the memory operations store the information that currently appears on the user's screen.

![Diagram](image)

**Figure 6.11: Subcomponents of the RPC library component**

Fourth, the RPC library component depends on a virtual machine which can provide operations for execution during processing of network messages and operations which can store and retrieve messages from the "store" (read: network) (Figure 6.11). Thus, the CPU provides processing operations while the network provides memory operations.

Fifth, the file system component depends on a virtual machine that provides operations for execution and data storage (Figure 6.12). The CPU provides processing operations and the disk provides memory operations.

This section has only presented the reasons why the SP model in Figure 6.1 is a legal SP model. The process of defining a composite work model according to the SP

\(^5\)The contents of the screen buffer are shown on the user's screen
rules is described and discussed in Chapter 11. There the method for formulation of composite work models shows clearly that the SP rules only constitute one criterion for a good model.

6.4 Calculation of Devolved Work

The purpose of building an SP model is to calculate the number of invocations of operations on each primitive component which results from a top-level operation. The SP tool\(^6\) takes care of the calculation. To calculate the number of invocations of a hardware (primitive) component because of invocations of a top-level component, multiply the matrices along every possible component-subcomponent path from the top-level component to the primitive component and then add the matrices that are obtained for each path. The matrix calculation is applied for each combination of a top-level component and a primitive component. The results may be used as parameters in a contention model. In some cases, the SP model results must be transformed to obtain contention model parameters. The practical use of SP modelling is described in Chapter 14.

As explained in Sections 4.2 and 4.8, it is sometimes necessary to model invocations of software components as well as invocations of hardware components. Hence, the number of invocations of non-primitive components are also needed. The SP tool is capable of calculating the devolved computational work on any SP component on any level in the SP model.

When the entire composite work model is used, the devolved work from the top-level components onto each primitive component is calculated. Section 6.2 defines the complexity matrices that are referred to here. \(W_{\text{component}_a}^{\text{component}_b}\) means the devolved work from \text{component}_a to \text{component}_b.

\[
W_{\text{Bank_application}}^{\text{CPU}_{\#1}} = C_{\text{Bank_application}} \cdot C_{\text{User_interface}} \cdot C_{\text{CPU}\#1}
\]

\(^6\)The SP method may be implemented by several different SP tools each of which may provide facilities to simplify SP modelling and SP calculations in practice.
Chapter 6. Work Modelling

\[ W_{\text{Bank\_application}} = \begin{bmatrix} \text{cre\_acc} \\ \text{upd\_acc} \\ \text{del\_acc} \end{bmatrix} = \begin{bmatrix} 96650 \\ 119500 \\ 93500 \end{bmatrix} \]

\[ W_{\text{network}} = C_{\text{RPC\_library}} \cdot C_{\text{network}} + C_{\text{RPC\_library}} \cdot C_{\text{RPC\_library}} + C_{\text{File\_server}} \cdot C_{\text{RPC\_library}} \cdot C_{\text{network}} \]

\[ W_{\text{Bank\_application}} = \begin{bmatrix} \text{cre\_acc} \\ \text{upd\_acc} \\ \text{del\_acc} \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \\ 2 \end{bmatrix} \]

\[ W_{\text{CPU\#1}} = C_{\text{Bank\_application}} \cdot C_{\text{User\_interface(M)}} + C_{\text{Bank\_application}} \cdot C_{\text{CPU\#1}} + C_{\text{Bank\_application}} \cdot C_{\text{RPC\_library}} + C_{\text{RPC\_library}} \cdot C_{\text{CPU\#1}} \]

\[ W_{\text{CPU\#2}} = C_{\text{Bank\_application}} \cdot C_{\text{File\_server}} \cdot C_{\text{RPC\_library}} \cdot C_{\text{CPU\#2}} + C_{\text{Bank\_application}} \cdot C_{\text{File\_server}} \cdot C_{\text{CPU\#2}} + C_{\text{Bank\_application}} \cdot C_{\text{File\_server}} \cdot C_{\text{File\_system}} \cdot C_{\text{CPU\#2}} \]

\[ W_{\text{Disk}} = C_{\text{File\_server}} \cdot C_{\text{File\_system}} \cdot C_{\text{Disk}} \]
These results can be used in order to calculate the devolved work of one or more transactions. First, the total number of invocations of each top-level operation must be found and expressed as a vector, \( T \). The devolved work\(^7\) of a set of transactions can now be calculated by multiplying the vector \( T \) with the \( W \)-matrices that were calculated above. For instance, if we want to calculate the devolved work of 2 \( \text{cre.acc} \), 3 \( \text{upd.acc} \), and 1 \( \text{del.acc} \), then:

\[
T = \begin{bmatrix} 2 & 3 & 1 \end{bmatrix}
\]

The devolved work is:

\[
W_{\text{CPU}\#1} = T \cdot W^\text{Bank.application}_{\text{CPU}\#1} = \begin{bmatrix} 645300 \end{bmatrix}
\]

\[
W_{\text{Network}} = T \cdot W^\text{Bank.application}_{\text{Network}} = \begin{bmatrix} 15 & 15 \end{bmatrix}
\]

\[
W_{\text{CPU}\#2} = T \cdot W^\text{Bank.application}_{\text{CPU}\#2} = \begin{bmatrix} 89930 \end{bmatrix}
\]

\[
W_{\text{Disk}} = T \cdot W^\text{Bank.application}_{\text{Disk}} = \begin{bmatrix} 9 & 9 \end{bmatrix}
\]

The devolved work of several transactions is appropriate for calculation of utilisations because there is no need to distinguish the transactions in that case. The devolved work on the primitive components are divided by the corresponding speeds in order to calculate the time it takes to execute the devolved work (assuming no contention).

If the devolved work of operations on intermediate components in the composite work model is needed, it is possible to calculate devolved work for subsystems of the composite work model. For instance, if the devolved work from operations on the \text{File.server} component to hardware devices in Figure 6.1 is required, \( W^\text{File.server}_{\text{CPU}\#1} \), \( W^\text{File.server}_{\text{Network}} \), \( W^\text{File.server}_{\text{CPU}\#2} \), and \( W^\text{File.server}_{\text{Disk}} \) must be calculated.

It may also be necessary to calculate how many times the top-level operations of the subsystem is invoked. In the client-server example this corresponds to calculating \( W^\text{Bank.application}_{\text{File.server}} \) which in this case happens to be equal to \( C^\text{Bank.application}_{\text{File.server}} \).

\(^7\)Calculated hardware resource demands
6.5 Assignment of Parameter Values

Ideally, all parameters in a complexity function should depend on parameters defined by a single superior component in a component-subcomponent relationship. Unfortunately, decomposition of a component-subcomponent relationship may expose new operations to several superior components which require different assignments of parameter values. This problem can be illustrated by the simple example that is shown in Figure 6.13a: The example shows a part of a composite work model which describes how two transactions, trans#1 and trans#2, accumulate the sum of all the account balances in their respective tables. The tables have the same data structure but have different number of accounts (i.e., records). Thus the `sum.accounts` operation of the `Library` component requires different number of invocations of the `add.integers` operations of the `CPU` component depending on whether it was called from `trans#1` or `trans#2`. The problem is that the parameter `n` cannot be given a distinct value since `trans#1` and `trans#2` requires different values of `n`. At least three different solutions are possible:

- Define two data classes for the `Library` component and define the `sum.accounts` operation for both data classes. The data classes correspond to the tables that the corresponding transactions read. The main disadvantage of this solution is that changes in the superior `Appl` component may require changes in its inferior `Library` component, thus the required locality of change is violated.

- The problem can be avoided by collapsing the `Appl` and `Library` components as depicted in Figure 6.13b. Then it is straightforward to let the number of invocations of `add.integers`, `n_1` and `n_2`, depend on the sizes of the tables.

- The problem can also be solved by allowing parameters of an inferior component to be instantiated dynamically depending on the superior operation.

The basic problem is to decide whether work component parameters should be given values at model definition time or at model run time. A composite work modelling tool should allow all the three solutions that were presented above. The main issue in choosing the solution is the expected reusability of the work model components.

Depending on how values are assigned to parameters, different techniques may be required when the devolved work is calculated. If a parameter have just one value throughout a calculation run, matrix multiplications and additions are sufficient. However, if the parameter values are dependent on the superior operation, the calculation must be executed as a depth-first traversal of the directed acyclic graph of the composite work model.
6.6 Benefits of SP Modelling

As a consequence of considerations of the abstract virtual machine formulation rules for SP models have been formulated. Architecture-independent structure and performance models result if these rules for well-structured models are followed. The selection of granularity of such models is a major modelling decision that will be treated in depth in Part III. An SP model may be regarded as a canonical model of the software of a computer system with respect to a specific granularity. It requires a lot of thinking and experimentation to formulate an SP model of a computer system architecture for the first time. However, the structure of SP models can be reused to a great extent. The probability of reuse is high because software systems with similar (fundamental) structure tend to have SP models with similar structure. Another benefit of SP modelling is that the formulation rules guides the identification of software components that should be represented explicitly in the SP model.
Chapter 7

Computer System Measurement

Measurement data are needed to determine the values of workload model parameters and contention model parameters as well as being necessary for validation of a performance model. Some performance measures may be obtained by means of standard measurement tools which use "standard" counters in hardware and in basic software such as the operating system. Other sources of information may be profiler code inserted by means of the compiler. If these sources of measurement data are not sufficient, extra program code must be inserted in the programs to be examined. This is called software instrumentation. Instrumentation of hardware is normally done during the design of the hardware although it may be possible to attach probes to some parts of the hardware circuits while the system is in operation. Since the hardware is becoming smaller and more circuits and electronic components are packaged into chips, it is difficult to get access to the measurement points where the probes should be attached.

![Diagram]

Figure 7.1: Not all the required measurement data can be obtained

The reason why measurement techniques are important in this thesis is that composite work modelling requires detailed measurement data which may be difficult to obtain. Software interactions as well as hardware device usage must be measured. The unfortunate situation that not all the required measurement data can be obtained, is depicted in Figure 7.1. The cost of obtaining high-quality workload
parameters for a contention model by measurement is normally very high because:

- it is difficult to decide on what to measure because detailed knowledge of the internals of the system is required
- doing the necessary measurements in a systematic manner is very time consuming
- some of the required measurement data may be difficult or impossible to obtain
- analysis and transformation of raw measurements may be necessary to obtain contention model parameters
- small but significant changes in the software may invalidate existing measurement data, i.e., new measurement data must be captured

Since measurement experiments are costly, measurement data must be considered as assets and should be reused, if possible, to reduce performance modelling costs.

General presentations of computer system measurement can be found in Ferrari [42], Ferrari et al. [44] and McKerrow [69].

7.1 Suitable Workloads for Measurement

Measurement experiments can only be carried out when the computer system is executing something. Some workload must be applied to the system during the measurement sessions. There are two types of measurement experiment. One type is workload measurement experiments which provide values for workload model parameters. The other type is performance measurement experiments which typically collect data about contention effects such as queueing, response times, and utilizations. Workload measurement experiments must be driven by a natural workload whereas performance measurement experiments must be driven by either a natural workload, a natural workload model, or an artificial executable workload model [52]. Depending on the workload during measurement, the factors that influence the execution are either observable or controllable. Measurement data that are captured while the computer system is run in production, i.e., with a natural workload, may not be reproducible. On the other hand, if an executable workload model is applied to the system during measurement, systematic runs with identical workloads are possible.

Since the focus of this thesis is work modelling, which is a part of workload modelling, workload measurement experiments are given the most attention in this chapter.
7.2 Classification of Measurement Data

Measurement data can be classified in two main groups: Contention-independent measurement data which describe the workload to be executed by the computer system and the speeds of hardware devices. On the other hand, contention-dependent measurement data describe contention effects because of shortage of computer resources.

Contention-Independent Measurement Data

Contention-independent measurement data describe either the computational work that is performed during measurement or the load that is applied to the computer system. Measurement data which represent the load include:

- Multiprogramming level
- Transition arrival rates
- User's think times

The computational work and the corresponding minimum execution time are described by the following measurement data (which may be dependent on the applied load):

- counts of software component interactions
- number of invocations of logical and physical hardware operations
- routing probabilities (if they are contention independent)
- service times at hardware devices (without contention)

If an operation is load-dependent, it means that extra computational work must be carried out to administrate the demand, i.e., load, for a service. For instance, the required amount of administration of execution threads in a multi-threaded software component increases when the number of pseudo-concurrent threads increases.

Contention-Dependent Measurement Data

The purpose of contention-dependent measurement data is to describe how a computer system executes its workload. The measurement data will illuminate any shortage of computer resources. It is important to quote the workload under which the measurement data were captured.
• residence times in hardware devices (include queueing time)
• transaction response times which is the sum of the transaction’s residence times in all software components it has visited
• queue lengths in front of hardware devices
• throughput, i.e., number of completed transactions per unit of time
• utilisation of hardware devices

This classification of measurement data corresponds to the distinction between load, work, and contention models in Section 1.1.

7.3 Measurable Components

Software interactions and hardware resource demands can be measured at different levels of detail. McKerrow classifies measurable components of a computer system in the following hierarchy [69]:

• System level – computer system and users
• Task level – a group of processes which execute to perform a request
• Program level – individual programs, processes, re-entrant routines, and supervisor calls
• Block-structure level – sequential blocks of code, compound statements, procedures, and functions
• High-level language level – individual instructions, interpreter calls
• Machine-code level – assembler instructions, and memory bus cycles
• Microcode level – microcode instructions and ALU cycles

Unfortunately, most measurement tools offer measurement of only a few levels of detail. For a given computer system, components at certain levels may not be measurable and instrumentation is therefore necessary. For instance, the hardware resource consumption of a transaction which utilises several processes can not be measured by state-of-the-art measurement tools. This corresponds to the task level in McKerrow’s hierarchy. For such systems extensive instrumentation and analysis by human experts are necessary.

In addition to measurement of components at some level of detail, it must be possible to measure how these components interact. Moreover, the cause and effect of these
interactions must be captured. For instance, the current best practice of counting interactions between software components is to use profilers to count the number of times component operations are executed. If the same operation is invoked from more than one software component, the measurement data must determine which software component did how many invocations of the operation. Figure 7.2 depicts three software components C1, C2, and C3, where C1 invokes C2 and C3. Each component offers two operations. To characterise the performance properties of C1, eight complexity functions are needed. Measurement problems occur when C2 or C3 are invoked from other components than C1 because it is necessary to distinguish the invocations from C1 and the invocations from other components. A better solution, if possible, would be to count invocations in the software component where the invocations are initiated. A critical issue is whether the operations offered by the software components are data dependent. These dependencies cannot be captured by profilers. Such information must be captured by instrumentation of the software.

Figure 7.2: Interactions between software components

7.4 Measurement Principles

Event Detection and Sampling  An event is a change of the state of a system. A hardware event, e.g., a CPU is interrupted, can be detected as one or more signals in the hardware. A software event, e.g., a transaction is initiated, is associated with the logical behaviour of the software. In most cases the frequency of hardware events is much higher than the frequency of software events. Event-detection is normally used when the sequence and the exact number of events must be collected. Sampling, on the other hand, is a statistical technique which records the state of the system at certain points in time. Data can be collected every time the contents of an event counter reach a certain threshold (count-sampling) or at pre-specified points in time (time-sampling). The measurement data that are collected at the sampling instants are assumed to be independent.

Direct and Indirect Measurement  Direct measurement means that each software component is measured one-by-one in a systematic, controlled manner. Thus,
the component can be thoroughly examined regardless of how it is used in a particular environment. A problem may be that there are too many possible contexts in which an operation of a software component can be executed. Consequently, it is necessary to restrict measurement to contexts which occur in a particular environment. Indirect measurement is performed when measured events and activities are not controlled explicitly. An example of indirect measurement is to run a transaction and at the same time measure how software components are invoked and how the hardware devices are used. Thus, the inputs and contexts are not controlled for each software component. If a production workload is measured, indirect measurements must be performed.

Duration of Measurement Sessions A too short duration of a measurement session may cause serious end-effects if there are jobs in the system that require long time to execute. If measurement data are collected by sampling, there is a minimum number of samples that must be collected before the samples are statistically significant. In addition, the sampling rate must be sufficiently high so that the events or states of interest can be captured. Long measurement sessions produce enormous amounts of measurement data. The amount of measurement data to store may cause practical problems, cost too much, and the process of storing the data on secondary storage media may take so long time that it distorts the captured measurement data.

7.5 Characteristics of Measurement Tools

Ferrari characterises measurement tools according to the following criteria [42]:

- Interference - the measurement tools may require additional execution in hardware and may alter the behaviour of the system in such a way that the performance of the system is degraded and the captured measurement data contain significant errors.

- Accuracy - inaccuracies may be introduced by interference, improper installation of the tool, incorrect operation, limitations in the event detector or inappropriate sampling frequencies.

- Resolution - the maximum frequency with which events can be captured and the maximum sampling frequency

- Scope - the types of measurement data that can be captured by the measurement tool.

- Pre-reduction capabilities - is the measurement tool capable of manipulating or filtering the measured data in order to reduce the volume of the captured data.

- Compatibility - the measurement tool must be compatible with the system to be measured.
7.6 Types of Measurement Tools

- Cost - the price of purchase, rental or lease, and the price of installation and maintenance
- Ease of installation - avoid accuracy problems by proper installation guidelines
- Ease of use - documentation is essential to obtain the required measurement data

7.6 Types of Measurement Tools

Measurement tools are classified as either hardware, software, or hybrid tools.

**Hardware measurement tools** consist of hardware which is added to the system to be measured. The main advantages of such tools are negligible interference and high resolution. The disadvantage is that it is difficult to relate macroscopic and microscopic events, i.e., it is difficult to relate hardware usage to the transaction which uses the hardware.

**Software measurement tools** consist of instructions that are added to the software in order to capture events in the software. The main advantage is that logical events related to processes, software components, and transactions can be captured. The main disadvantage is that the extra instructions that are added to the software influence the performance of the system and require space for buffering of measurement data. The software to be measured is changed by either [44] addition of a program, modification of the software itself, modification of the operating system, or modification of a system program.

**Hybrid measurement tools** The rationale for hybrid measurement tools is to combine the advantages of hardware and software measurement tools. The software tool detects events which can only be detected in the software, and write information about the detected event to a hardware device. The data written to the hardware device are collected by a hardware measurement tool. Thus, the interference caused by extra instructions in software is minimised and the resolution of hardware measurement tools is retained. The hardware events are detected by the hardware part of the measurement tool.
7.7 Requirements on Measurement Tools

Bell and Falk [16] have found that certain requirements on measurement tools must be fulfilled before software performance engineering becomes feasible. These requirements apply for composite work models as well. The requirements are:

1. measurement tools are needed that can collect and log system variables over the period of a test rather than merely giving a series of snapshots
2. measurement data must be available for repeated analysis so that confusing problems can be examined in detail and in different ways
3. measurement data collection should be started at the beginning of the period of interest; the collection should be stopped at the end of the period; and the data collected should be summarised and reported offline
4. a time-stamp should be provided to give the time each set of measurement data is collected
5. measurement data should be collectible on either a sampled basis or on an event-by-event basis as appropriate
6. the resolution of time measurements should be adequate; the required resolution is on the order of 50 microseconds to 100 microseconds
7. the beginning and ending times for collecting data should be specifiable; the sampling period must also be specifiable if data is being collected on a sampled basis
8. collected measurement data should be available to the analyst on an item-by-item basis in order to do detailed system tracing, filtered to present only data of interest, or in a summarised format
9. captured measurement data should be presentable in different forms; as hard-copy, in a form which is readable by statistical software packages
10. it should be possible to correlate captured measurement data with other logged system events

These requirements correspond very well with the experience gained by measurement of distributed computer systems during the case studies.

7.8 Measurement of Distributed Systems

The current best measurement practice for non-distributed systems is to measure hardware demands directly for each software component. The multi-level structure
in most transaction-processing systems are not taken into account. In distributed systems, measurement must take place separately for each hardware platform. Consequently, interactions between software components on different hardware platforms must be measured explicitly because the relationship between the software components must be known. It may also be desirable to measure software components on the same hardware platform separately. In addition, measurement of interactions between software components is a prerequisite for composite work modelling. Consequently, both hardware resource demands and software component interactions are needed to characterise the multi-layered software of distributed computer systems. Some research in measurement of distributed systems is described in [104, 54].

Measurement of the Global State  It is difficult to measure the state of a distributed system because such measurement needs a global clock and the propagation time of measurement data between the computers in the distributed system may be unpredictable.

Storage of Measurement Data  In non-distributed computer systems, both the invoking operation and the operation being invoked are known to the operating system on one computer. Hence, this information can be captured and stored together. In a distributed system, the invoking operation and the operation being invoked may run in different computers. Thus, these two events must either be captured separately on each of the two computers, or details about the invocation must be transferred together with the message to the operation being invoked.

Measurement in terms of Transactions  The software processes need service from hardware devices such as CPUs, memories, disks, and controllers to execute parts of the transactions. The measurement problem is to account for the hardware resource usage of each transaction. The problem arises because software processes and hardware devices are used by several kinds of transaction, possibly with very different resource usage. Therefore it may be difficult to distinguish between transactions, at least if the system under investigation is run in production. Measuring a system in production means that transactions cannot be run one at a time without disturbing the normal use of the system. Measurement data for contention model parameterisation can be collected when transactions are run one at a time because model parameters by definition should be contention independent. As will be explained in Chapter 15, measurement in terms of instances of transactions will simplify the validation of a composite work model.

Heterogeneous Measurement Tools  It may be a problem that different hardware platforms have different kinds of measurement tool. This implies that it is difficult to obtain compatible measurement data from different hardware platforms.
7.9 Design for Measurement

Both the software and hardware of a computer system should be instrumented or prepared for instrumentation as part of the design. Such instrumentation is useful during testing of the functionality of the system, debugging, and for performance measurement experiments. The main problems are to predict the measurement data that will be needed after the system has been implemented, and to avoid excessive instrumentation. The main reason for instrumentation at design time is that the designer have a complete understanding of the system component that he is designing. Therefore the factors which influence each component are known and the component can be instrumented accordingly. In addition, the interference caused by the instrumentation can be minimised by design decisions. It must be possible to activate and deactivate the instrumentation so that there is no interference from the instrumentation when it is deactivated.

The consequences of not designing systems for measurement are excessive interference during measurement, events of interest are not accessible, difficulty in verifying collected data, hardware and software probes may not be placed where they are needed, and finally that the modifications of the system may be difficult, risky, and expensive. The result is a computer system which is difficult to test and the performance cannot be controlled.

7.10 Measurement Method

So far different types of measurement data and measurement tools have been presented. A measurement method is required in order to structure the measurement activities. Ferrari [42] presents the following steps:

1. Decide what to measure
2. Select the measurement tool(s)
3. Design measurement experiments and estimate their cost.

Apparently, these steps are performed in sequence. In practical measurement planning it is necessary to take all steps into account at the same time! It is not possible to decide what to measure before one knows which measurement data that can be captured by the available measurement tools. Some data may be particularly expensive to capture. Are the data worth the cost? Many iterations of the steps in the measurement method may be necessary.

As will be shown in Part III, composite work modelling is closely related to the measurement method for performance characterisation of software components. Step 1
corresponds to selecting the granularity and degree of parameterisation of a com-
posite work model. Step 2 corresponds to finding measurement tools that can capture
information about interactions between software components. There is also a need
for measurement of hardware use in terms of software components and transactions.
Step 3 corresponds to using the composite work model as the guideline for the design
of measurement experiments and as the means for estimation of cost of the mea-
surement experiments. Since composite work modelling is closely integrated with
measurement, the approach could be called model-based measurement.
Chapter 8

Tool Support for Performance Evaluation

There exist a lot of performance measurement, modelling and evaluation tools. The presentation of tools in this chapter is restricted to tools that were explored during the IMSE project and during the doctoral work. Other tools are described only briefly. First, measurement tools are presented. Secondly, stand-alone performance modelling and evaluation tools are described. Finally, a few integrated performance modelling environments are presented.

8.1 Measurement Tools

The characteristics of computer measurement tools were presented in Section 7.5. As examples of measurement tools, a tool provided by Tandem and some tools provided for Unix systems are presented here. All the tools that are presented were used or considered for use during the case studies. Unfortunately, most measurement tools are aimed at capturing the overall hardware usage. The support for capture of software interactions and hardware usage in terms of software components is very limited.

Tandem Measurement Tools

MEASURE is a comprehensive measurement tool designed for the Guardian 90 operating system [37, 96, 95]. One design objective was to minimise the measurement overhead caused by the tool. In most cases, MEASURE consumes less than 1% of the CPU time. In the C20 version of MEASURE there are 17 measurable entities. A measurement run may include any combination of measurable entities. Also concurrent measurement runs are possible. If possible, counters will be shared between
the concurrent measurement runs.

The MEASURE tool consists of a set of software processes. The main processes are:

- **MEASCOM** accepts and processes the user commands that control MEASURE.
- **MEASMON** coordinates the MEASURE system, e.g., it starts and stops measurements and the MEASURE subsystem.
- **MEASCTL** allocate and initialises counters, and they periodically writes counter values to the data file. There is a MEASCTL process running in each CPU that is affected by the measurement experiments.
- **MEASFH** builds counter records from the data in the measurement data file. It also creates and initialises the data file. On demand, the MEASFH process reads data from the measurement file and translates it to an external format which can be interpreted by humans.

As the system runs, various operations within the system code and system processes update the counters. The counters that are updated during measurement of a MEASURE entity can be classified as follows:

- An *incrementing* counter is increased by one every time a specific event occurs. Example: number of messages received by a process.
- An *accumulating* counter adds a value to the counter every time a specific event occurs. Example: Number of bytes received by a process.
- A *busy* counter captures the amount of time a resource is busy. Example: CPU busy time.
- A *queue* counter captures the total amount of time some element spend in a queue. If two elements each spend 10 milliseconds in the queue, the queueing time is 20 milliseconds.
- A *max queue* counter keeps track of the maximum length of a queue.
- A *sampling* counter captures approximate busy times by sampling.

There are other kinds of counter, but they were never used in the case study. MEASURE can capture data about several entities at the same time. The following list of measurable entities were used during the measurement runs in the hospital case study:
8.1. Measurement Tools

<table>
<thead>
<tr>
<th>Entity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Info about the use of a CPU</td>
</tr>
<tr>
<td>Disc</td>
<td>Info about the use of a disk</td>
</tr>
<tr>
<td>Discopen</td>
<td>Info about physical I/O operations on a file</td>
</tr>
<tr>
<td>File</td>
<td>Info about logical I/O operations of a file</td>
</tr>
<tr>
<td>Process</td>
<td>Info about the resource usage of a process</td>
</tr>
</tbody>
</table>

Measurement data can be collected for the entire lifetime of an entity (e.g., a process) or they can be collected at regular intervals.

The following deficiencies of the MEASURE measurement tool were experienced:

- It is impossible to record only a subset of an entity’s counters. This causes unnecessary space consumption when only a few counters in each entity are of interest.

- MEASURE does not support capture of the relationship between incoming messages and outgoing messages. MEASURE only capture the id of the sending process, the properties of the message (e.g., the message length), and the id of the receiving process.

The second deficiency can be remedied to some degree by instrumenting the application programs with calls to MEASURE’s user-defined counters. But since PATHWAY applications (see Chapters 2 and 3) are standardised it should be possible for the MEASURE tool to capture more detailed information about the behaviour of the software components without explicit instrumentation.

These deficiencies makes the tool unsuitable for composite work modelling where measurement-in-the-large and measurement of the internals of software components are necessary.

Unix Measurement Tools

According to [16], the different versions of the UNIX operating system perform very differently. Major performance issues in one version may be minor performance issues in another version. Each version has a set of measurement tools, but the “same” tool may work differently in different versions. The resolution of timing information may be different. In fact, one version of UNIX may offer measurement tools that are not available in other versions.

The UNIX measurement tools are in most cases oriented toward changing system parameters so that a set of unknown applications can execute efficiently (also called tuning). The tools are capable of capturing system-wide performance. However, very few measurement tools are directed at measuring the internals of applications
or directed at measuring interactions between application components. Hence, these measurement tools are not suitable for distributed computer systems (see discussion in Chapter 13).

According to the Performance Management Work Group of UNIX International [80], the measurement of UNIX system suffers from the following problems:

- It is difficult to correlate data from different sources. For example, it is difficult or impossible to correlate process activity with any aspects of system behaviour.

- The granularity of data is often not correct. For example, process accounting records are only written at process termination. There is no mechanism to determine how the process behaved during its lifetime.

- It is not possible to extend the data collected with these tools without the availability of source code.

- The measurement data are typically inflexible and awkward to view.

- There is a lack of standard analysis tools for measurement data collected on UNIX systems.

Unfortunately, there are very few comprehensive measurement tools for UNIX. Therefore, a short survey of the available measurement tools under UNIX includes only small special-purpose tools. The following subset of the tools provide snapshots of the use of the system:

**ps** Display the status of the current UNIX processes. Provides CPU time consumed by or on behalf of process (moving average in seconds), the process' memory usage, paging and swapping information. Warning: Things can change while "ps" is running; the picture it gives is only an approximation to the current state.

**iostat** Reports on disk and terminal activity as well as the percentage of time the system spent in user mode, nice mode (low-priority), system mode, and being idle.

**netstat** Report on network activity between computers.

**nfsstat** Displays statistical information about the Network File System (NFS) and Remote Procedure Calls (RPC). Information about the number of unsuccessful RPC calls, the number of retransmissions of RPC calls, etc., can be captured. Either the client side or the server side, or both, can be examined.

**vmstat** Report on memory usage, paging, swapping and CPU activity in terms of UNIX processes.
8.1. Measurement Tools

**system accounting** Stores information about the activity on a UNIX system. The accounting system is mainly targeted at charging money for the usage of the system. The resolution of timings (in minutes) is too coarse for performance measurements. However, some of the counts may provide useful information. In addition, turning system accounting on enables additional options for "timex".

**sar** (only UNIX System V) This tool outputs "snapshots" of the performance of a UNIX system. For instance, the following performance measures can be reported: CPU utilisation, buffer activity, activity for each block device (e.g., disk), system calls, system swapping, use of file access system routines, run queue of processes in memory, status of processes, message and semaphore activities, paging activity, unused memory and disk space, kernel memory allocation activities, remote file sharing operations, remote file sharing activity, server and request queue status, remote file sharing data caching overhead. The snapshots may be stored on a file to log the performance for longer periods of time. This tool is oriented towards hardware devices, i.e., it does not distinguish between transactions; not even software processes are distinguished. Therefore, if two or more processes are involved in a transaction and they run in the same hardware, it is not possible to distinguish the resource usage of the processes, e.g., when counting disk accesses demanded by the processes.

Two tools provide resource usage and timing of a specific UNIX process (or command):

**time** Returns the following measures concerning execution of a process when it finishes: process CPU time, UNIX kernel CPU on behalf of process, elapsed time, number of inputs and outputs, number of page faults and swaps, and memory usage. The CPU times are accurate to the 1/50th second (1/100th on some systems), while the elapsed time is accurate to the second. If the process being timed is interrupted, the timing values returned may not always be accurate.

**timex** (only UNIX System V) Provides information about processes, e.g., elapsed time in seconds, user CPU time in 1/100 seconds, and system CPU time on behalf of a user process in 1/100 seconds. Other performance-related information about the execution of a process is available provided that the "system accounting" is running.

The following tools are used for generation of profile information. The raw measurement data are collected by instrumentation inserted by means of the compiler:

**prof** Display profile data that were collected during program execution. The C programs to be profiled must be compiled with the "-p" option. The profile output contains counts of function calls and CPU time (approximated by sampling)
spent in each function. By default, the profiler only profile function entry points. Other parts of the code can be profiled by issuing "MARK(arbitrary name)" commands in the source code.

lprof (only UNIX System V) Display line-by-line (in source code) execution count profile data. The program to be profiled with lprof must be compiled with option "-ql".

gprof (only Berkeley UNIX) In addition to the information output by "prof", gprof also produces the call graph. The program to be profiled with gprof must be compiled with option "-pg".

tcov (only Berkeley UNIX) While "prog" and "gprof" only output information about functions, "tcov" provides profile information for each LINE of C code. The program must be compiled with option "-pg".

There are two procedures that can be called from within a program. It is the responsibility of the program to store the collected information in a suitable format:

getrusage (only Berkeley UNIX) A C function in the C library that collect information about the resource usage of a process.

times (only UNIX System V) A C function in the C library that collect information about the resource usage of a process.

In addition the "trace" facility of UNIX is useful when information about the UNIX system calls and network calls are needed.

New measurement tools for UNIX are currently being developed to remedy the shortcomings of the tools that were presented above. The new tools allow measurement of the internal events of software components and the interactions between software components. For instance, the mandate of the Performance Management Work Group of UNIX International [80] is to define requirements and standards for the collection, presentation and distribution of performance data in large-scale to small scale distributed systems. Such performance data include:

- Interval or sampled data describing hardware and software resource usage or times, either globally or by some logical entity.
- Count data representing system or application queue lengths, events, and system resource states.
- Data representing execution traces of processors.
- Data notifying of events occurring at a system, subsystem, or application levels.

There is also a special interest group (SIG) who is working on standardisation of performance instrumentation in the distributed computing environment (DCE) of Open Software Foundation (OSF) [45].
8.2 Stand-alone Software Tools

Workload Analyser Tool – WAT

The Workload Analyser Tool (WAT) [31, 28] was developed at the University of Pavia in Italy. It supports statistical analysis of measured workload components as well as cluster analysis of the workload components. Each measured workload component is described by a tuple of parameters. WAT allows definition of different formats of input files. The formats are stored in a library. Statistical descriptors such as mean, variance, standard deviation, and skewness of each parameter of workload components are calculated. In addition, the correlation matrix for the parameters are calculated. The statistical information is necessary in order to select the parameters to be used in the cluster analysis. The tool offers transformations of the measurement data in order to obtain a good metric for assessing the closeness between workload components during the cluster analysis. Sampling and removal of outliers are also supported. The results of the cluster analysis consists of lists of the workload components which belong to the same cluster. The centroid of each cluster and information about the dispersion within each cluster are reported. WAT is implemented in Fortran77 and can therefore be ported to several kinds of computer. It has been tested on IBM under VM/CMS and MVS/TSO, VAX under UNIX and VMS, and on SUN under UNIX. A graphical user-interface was implemented during the IMSE project.

Structure and Performance Tool – SP

The Structure and Performance Specification Tool was developed by STC Technology Ltd and BNR Europe Ltd during the IMSE project [57, 82]. The tool implements the SP method [58] (described in Chapter 6). The SP tool provides input forms for definition of primitive and non-primitive modules as well as complexity matrices. Several data classes can be defined for each module. Operations are defined for each data class of each module. SP components are obtained when the implementations of SP modules are described by complexity specifications. The tool provides a graphical user interface where the user can combine the SP components in order to obtain an SP model. The components appear as nodes which are connected by links. The line width of a link indicates whether the link is of type communication, processing, or memory. Parameters of SP components can be given values by clicking on an SP component followed by the filling in of the form that pops up. Facilities for calculation of devolved work and devolved storage are available.
Process Interaction Tool – PIT

The Process Interaction Tool (PIT) [10] was first developed at the University of Edinburgh in collaboration with STC Technology Ltd. during the SIMMER project. It was then developed further by STC Technology Ltd (later BNR Europe Ltd.) during the IMSE project. The tool provides a graphical interface where the user can draw activity diagrams [18, 84, 9] (see also Section 4.5) directly, annotate the nodes and links that result, and provide information about nodes and links by clicking on the relevant object and by filling in the form that pops up. When the user has finished drawing and annotation of the activity diagram, the PIT tool is capable of translating the activity diagram to SIMULA [81] code which is then compiled by a SIMULA compiler. The PIT tool assumes that the DEMOS (Discrete Event Modelling on Simula) [18] extension to SIMULA is available. DEMOS provides high-level primitives for simulation programs. It is possible to write SIMULA code directly by introducing source code nodes in the activity diagram. By clicking on source code nodes an editor pops up and source code can be entered. This feature is important because modifications of the generated code invalidates the activity diagram.

Queueing Network Tool – QNAP2

QNAP2 is a commercial software tool which supports formulation and evaluation of queueing network models. It is developed by Bull and Simulog (France). A queueing network model is formulated in a specification language that is specific to the QNAP2. Depending on the properties of the queueing network model, e.g., type of queueing centres, scheduling disciplines, arrival and service time distributions, an appropriate solver is selected for the evaluation of the model. QNAP2 supports analytic modelling, Markov analysis, and simulation. In any case the model is formulated in the same language. QNET [38] is a graphical front-end to the QNAP2 tool. It was developed by Simulog during the IMSE project.

Petri Net Tool – GreatSPN

The GreatSPN package enables Petri net modelling and evaluation. The tool has evolved according to the research results on Generalised Stochastic Petri Nets (GSPN) produced at the University of Turin [33]. The tool has a graphical user interface which enables intuitive drawing and annotation of Petri nets. The analysis of a Petri net results in a list of properties of the net such as deadlocks. A survey of different kinds of Petri net is provided in [72]. The core of the GreatSPN package are written in C which is easily portable. The PNET [8] provides a new user interface to the evaluation algorithms of GreatSPN. It was necessary to adopt a new user interface in order to fit into the IMSE framework.
Other Performance Modelling Tools

**BGS** The performance tools provided by BGS Systems are described here because they are examples of commercial successes. BGS Systems have pioneered the field of capacity management with these tools. The different tools are responsible for different phases of a performance modelling study. The performance tools fit into a three-step methodology: Step 1, “Data collection and workload characterisation”, uses the CAPTURE tool to describe the current workload of a system, the resource consumption, performance, and current growth. Step 2, “Workload forecasting”, uses the INFO/BASE tool to store and manipulate historic data about a system and uses the CRYSTAL tool to determine the impact of adding new software components to the system even before they are developed and implemented. Step 3, “Performance prediction”, uses the BEST/1 tool to determine the consequences of changes of the system.

All the BGS tools are primarily targeted at IBM mainframes and plug-compatible mainframes which run either the MVS or the VM operating system:

**CAPTURE** collects standard measurement data and organises the data so that the performance of the system as well as the reasons for the performance problems can be analysed.

**INFO/BASE** stores, retrieves and manipulates historic measurement data. Based on these historic data the tool can be used for workload forecasting.

**CRYSTAL** characterises and forecasts the performance consequences of future applications based on the system specifications that are available in each phase of the software design. BGS can supply information about hardware characteristics and overhead caused by CICS, IMS and/or DB2.

**BEST/1** can be used for “what-if” analysis based on data from CAPTURE and CRYSTAL. CAPTURE can generate a baseline model for BEST/1.

BGS Systems also provides tools for characterisation and analysis of network components: CAPTURE/SNA and BEST-1/SNA.

8.3 Integration of Software Tools

A performance modelling study may require use of more than one performance modelling and/or evaluation tool. Integrated performance modelling environments enable a smooth transition between the tools. Models and their input and output data are stored in a common store. Hence results from one model run can be used directly as input data in other model runs. Common services such as experiment planning and execution, report generation, and a common user interface, can be provided for all tools.
The Integrated Modelling Support Environment (IMSE) [57, 82] is the result of the IMSE project (ESPRIT II #2143). It is an open-ended environment where new performance tools can be introduced. A workbench is provided in order to enable easy invocation of any tool and to browse objects. All the tools have the same user interface. All tools store models, inputs, outputs and other information in a common format in a common object database (currently implemented on the Portable Common Tool Environment – PCTE). The modelling environment is based on a conceptual model [56] of objects that are involved in performance model construction and evaluation. The facilities for definition and execution of experiments allow runs of several models during an experiment. The concept of free variables in models makes it possible to postpone assignment of parameter values to experiment run time. This is important when the parameterisation of a model is dependent on the results of other model runs.

![Image of the IMSE architecture](image)

Figure 8.1: The IMSE architecture

The IMSE architecture is depicted in Figure 8.1. The shaded rectangles represent components that are present in all IMSE systems. The other rectangles represent tools that have been introduced in the IMSE in order to benefit from the common user interface, object management system, experimenter, etc. The modelling tools
belong to either the *system description level* (WAT and SP) or the *performance modelling level* (QNET, PIT and PNT). The modelling tools were presented separately in the previous section. The following describes the benefits of the IMSE and the achievements of the IMSE project:

**Workbench** The workbench provides a unified view of all the IMSE tool and the models. If the user knows one of the tools it is easy to learn to use another IMSE tool because the screens look familiar and the pop-up menus are used in the same way. This is important because it encourages use of the most suitable tool, not just the tool the user knows best.

**Graphical interfaces** For non-experts in performance evaluation it is easier to use modelling tools with graphical interfaces than to write some kind of program where one must be aware of all kinds of detail. This interface encourages definition of models in a top-down fashion: First the overall structure of the model is defined and then stepwise refinements are carried out until the model is finished.

**Object Management System** Any program, model, and data are stored in OMS objects of certain types. The relationships between objects are defined by the IMSE conceptual model. Dependencies between objects are maintained by the OMS to avoid inconsistencies because of "unfortunate" deletions of objects. In addition, controlled deletions are cascaded as appropriate.

**Experiments** Running performance models using stand-alone tools induces a lot of manual work because the user must type in the input before the model is run and then collect the results. A systematic investigation of a parameter space increases the amount of manual work dramatically. Having defined experiments, it is more likely that model runs will be repeated, e.g., for different sets of measurement data. Experiments also enable cooperation between different modelling tools. Experiments serve as documentation since they describe how results were produced. Experiments also ensure reproducibility of results. For the novice IMSE user experiments are predefined model runs.

**Model parameters** By defining model parameters as *unknown* (or free) the modeller does not have to think about the sources of parameters during model definition. The sources of parameters are defined in the experimenter. Therefore it is possible to change the sources of model parameters without changing the model. The source of parameters can be, e.g., results of other experiment runs, a range of values, or a fixed value.
System description level  The system description tools provide a possible link between the information system tools and performance evaluation tools. They support modelling of software in a structured way. Computer resource demands for performance models are the main outputs of system description models.

Open-endedness  The IMSE environment can host any tool that obeys the conventions imposed by the IMSE user interface and the IMSE object management system. Most important: The IMSE environment is not restricted to performance evaluation tools. This lowers the barrier between information systems tools and performance evaluation tools significantly.

The purpose of presenting the IMSE here was to describe typical properties of integrated modelling environments. Similar properties are or will be adapted by all integrated environments.

Other Integrated Performance Tools

HIT  Beilner et al. [14, 13, 15] at University of Dortmund have developed HIT, an integrated software tool for model-based performance evaluation of computer systems during all phases of their life cycle. Components are defined by partitioning the computer system vertically, into a sequence of layers and levels which communicates via function calls, and horizontally, into independent information-hiding modules. A component is described in an algorithmic high-level language, HI-SLANG. Components can also be described by means of a graphical user interface. Different components can be solved using different evaluation techniques. When some or all of the components have been solved separately, the partial solutions are combined to find the overall solution. The freedom in use of the algorithmic language depends on which evaluation technique that is selected for a component. Analytic models require that only product form properties are specified in the algorithmic language. More freedom is allowed if Markov models are requested. No restrictions apply if simulation is requested. In that case the algorithmic language is just like any third-generation programming language which allows (almost) arbitrary parameterisation of function calls. In summary, HIT allows specification and evaluation of a composite system model. All details about the evaluation techniques are hidden from the user of HIT.
Chapter 9

Sizing of Computer Systems

Sizing is the activity of selecting a software and a hardware configuration which satisfy given functional, performance and capacity requirements for a given workload. The quality of sizing results depends heavily on the quality of component characterisations. Since the component characterisations will be combined in different ways, it is important that they are as independent and robust as possible.

If the accuracy of sizing results are acceptable, sizing can to some extent be a cost-effective alternative to benchmarking. Benchmarking requires that artificial executable workload models (see Chapter 5) are run on different hardware configurations in order to determine the most suitable configuration. Finding an appropriate benchmark which can represent the given workload may be difficult. In addition, it is time-consuming and expensive to put together computer systems in different ways in order to run the benchmarks.

The sizing activity is given special attention here because the sizing procedure and the application of composite work models is quite similar [105].

9.1 The Sizing Procedure

The sizing of computer systems with respect to a particular workload depends heavily on the constructivity of the decomposition of the software. The sizing procedure is quite similar to the fundamental principles of Langefors (see Section 1.2). A sizing model and tool require continuous development and maintenance of the performance characterisations:
Hardware and software components are characterised by experts. The characterisations are based on systematic measurement of each component.

The component characteristics are stored in a library in order to facilitate easy retrieval during a sizing study.

One or more algorithms must be developed in order to combine the component characteristics according to the selected software and hardware configuration.

In some sizing tools the component characteristics are used as parameters in a contention model such as a queueing network model. Thus, contention-dependent results such as transaction throughput rates and response times can be estimated by the sizing tool.

Reporting facilities must be provided so that a non-expert can understand the results of the sizing study.

The performance characteristics of planned hardware and software components should also be included in the sizing library to support performance evaluation of configurations which include components that are being designed.

Since the number of feasible configurations of software and hardware components may be very large, the sizing procedures are often specialised for certain “families” of software and hardware configurations.

A sizing study requires the following inputs:

- Selection of hardware configuration
- Selection of software components
- Description of the workload, i.e., how the users are expected to use the system

The results of the sizing study include the utilisation of hardware devices, response times and throughput rates in terms of transaction classes. Device utilisations can be calculated without any contention model while estimation of throughputs and response times require a contention model. Some sizing tools may also give recommendations concerning the choice of hardware and software components in order to meet the performance requirements provided by the customer.

The results of sizing are subject to the specified workloads. If the real workload turns out to be different from the predicted workload, the sizing may be worthless. This is a serious problem when sizing results are used in contracts to provide guaranteed performance subject to a workload which may have been specified by the customer.
9.2 Sizing Tools

Most computer system vendors have one or more sizing tools for the ranges of configuration alternatives they can provide to their customers. Unfortunately, these tools are normally built for internal use only. However, most sizing tools have most of the functionality in common.

The purpose of a sizing tool is to enable a non-expert to combine the stored performance characteristics of system components in different ways. Quite often it is required that performance characterisations of different computer system configurations can be obtained quickly in response to a customer request. The tool must have a user-friendly interface which is easy to use and it must hide all irrelevant details. The inputs to the sizing tool must be easy to capture.

A graphical user interface enables the user to configure a system by combining icons on the screen. The icons are connected or placed adjacent to each other in order to show a relationship between the components. Necessary input parameters for each component is entered by first "clicking" on the corresponding node and then by filling in the required parameters in a screen form. This mode of interaction with the sizing tool is user driven. Unfortunately, this mode requires expertise in order to do the right things in the right order and to provide complete information to the tool. On the other hand, textual sizing tools are often system driven. This mode of interaction requires less expertise because the user is prompted with the right questions in the right order to provide a complete input description of the system to be sized.

9.3 Users of a Sizing Tool

Three kinds of user can be distinguished; the regular user of the sizing tool, the sizing expert, and system designers.

The regular users are typically persons with a technical background who work in the sales department. They have contact with a lot of (potential) customers and must provide performance predictions of standard configurations of the company's computer systems. For them, the most important feature of a sizing tool is quick sizing results. These users will in most cases benefit from a system-driven sizing tool because of their limited expertise. The reports that are generated by the sizing tool must be understandable to the customer.

The sizing experts are experts on system configuration as well as performance and capacity calculations. The difficult sizing studies are carried out by these experts. Difficult cases include sizing of "one-off" systems and sizing of systems for
big customers where extreme care must be taken. A sizing expert would typically prefer a user-driven user interface. The sizing report can be very detailed to give as much feedback as possible to the expert.

The product designers are developing a new hardware or software component. To ensure that the design can meet the performance requirements, performance characteristics of the other components in the system are needed. As explained in Chapter 2 there is a definite trend towards open computer system architectures where new components can be added or new components can be exchanged with old components. In such systems, just a few components are at the design stage at any time. Thus, the performance characteristics of the existing components can be measured and used as input to the design of the new components. The product designers would benefit from a user-driven user interface. Moreover, the ideal solution would be a sizing tool that was integrated into the design tools (e.g., a computer-assisted software engineering tool) [78, 92].

9.4 Deficiencies of Contemporary Sizing Tools

A major deficiency of current sizing tools is that the multi-layer structure of software of transaction-processing systems is not represented in a general way in the sizing models. Another problem is that sizing tools very often are tailored to specific families of hardware and software components. Thus, commonalities between the sizing systems are not exploited. Also, the use of different sizing tools becomes difficult at least to the inexperienced user. Therefore a coherent approach to sizing should be investigated.

Many sizing tools are programmed in a third generation programming language. Therefore, maintenance becomes very costly and time-consuming.

The current best practice of sizing seems to be rather "ad-hoc". A well-planned approach to sizing would be beneficial. Such an approach would require that hardware and software components are designed for measurement and sizing. Then performance could be controlled thoroughly during sizing and maintenance of computer systems.
Part III

Composite Computational Work Modelling

This part presents how composite work models are formulated, used and assessed. In addition, measurement according to a composite work model is discussed.

Chapter 10 presents a classification of complexity functions and the information needed in order to determine complexity functions. Chapter 11 presents the steps through which the formulation of composite work models proceeds. Chapter 12 shows how the method of Chapter 11 can be applied in practice. Chapter 13 presents measurement experiments that are required for composite work modelling. Chapter 14 recapitulates the contention models that were presented in Chapter 4 and shows how composite work models can provide parameters for these contention models. Chapter 15 discusses quality criteria for completed composite work models.
Chapter 10

Relationships between Work Model Components

The relationships between work model components which correspond to software and hardware components of distributed systems, constitute the basis for formulation of a composite work model. The presentation in this chapter is based on software component interactions and hardware usage that were encountered in the case studies. The main question is how much information about software component interactions is needed to assess alternative composite work model formulations before extensive measurement experiments are undertaken, i.e., how meaningless, useless or impossible measurement experiments can be avoided.

The relationships between work model components are described by complexity functions. Complexity functions may be parameterised in order to make them reusable. Note that operation-suboperation relationships and not operations are classified here because an operation may invoke its suboperations differently, i.e., the corresponding complexity functions may have different forms.

Chapter 11 presents the considerations which underlie the formulation of a composite work model, i.e., how complexity functions are put together.

10.1 Prerequisites

Before composite work modelling can commence, suitable software components must be identified. Preferably, the components should have a high probability of reuse and have a high probability that changes will be local to a few components, i.e., that sound software engineering principles are followed. Software components which conform to a software architecture such as ANSA or PATHWAY (see discussion in Chapter 2) may have a high probability of reuse because many software systems will be structured in the same way. The software components that are identified define
the finest granularity of a composite work model of the system. The granularity of the composite work model is made coarser by collapsing work characterisations of software components. For instance, it may be too costly to measure the interactions between two software components, therefore they are treated and characterised as if they were part of the same software component.

The functionality of each software component must be described and the interactions between the components should be surveyed. The components are arranged in a hierarchy such that operations in superior components invoke operations in inferior components. The purpose of this survey of software component interactions in distributed systems, is to find typical characteristics of complexity functions. Simple scenarios are presented to illustrate how the software components interact.

10.2 Availability of Performance Information

Characteristics of algorithms obtained by theoretical analysis (e.g., [2, Chapter 9]) may be useful when the form of a complexity function is selected. In the literature, algorithms are often categorised by the $O$ notation, e.g., Quicksort is $O(n \log n)$ in the average case and $O(n^2)$ in the worst case. Here, $n$ is related to the size of the problem to be solved. However, the $O$ notation is not precise enough to characterise the work induced by an operation for the purpose of composite work modelling. Moreover, it is not just the size of the problem to be solved that influences the amount of computer resources that are consumed.

There are three main sources of information which are useful for system description with respect to performance: software design specifications, source code, and execution characteristics of software. Existing software can be described by measurement of its behaviour. The granularity of available measurement data depends on the measurement tools and the degree of instrumentation. The source code may be available, but for commercial software it is not generally available. Design specifications may be available, but there is a risk that they are out of date. For source code that is generated based on a software design specification, the design specification will always be up to date. If no design specification is available, the source code must be inspected and the component structure and the interactions between the components of the software must be determined by a reverse engineering procedure. For composite work modelling, the most important part of the software design specification is the specification of the software components' interconnections [86, 7].

The availability of information and knowledge about the software components determines the degree of detail of operation characterisations. Three degrees of availability were experienced during the work on the case studies. The client-server example was programmed by the author. Thus the system was seen from the viewpoint of an implementer where detailed knowledge of the application part, some knowledge about the libraries, and little knowledge about the operating system details, were available. The ICL case study was performed from the viewpoint of a consultant
for a software and/or hardware supplier where overall information about the system and the relationships between the components were available. However, detailed information about the implementation of each component was not readily available. The hospital case study was conducted from the viewpoint of a consultant for a customer where information about the system must be captured by use of standard measurement tools and by reading the documentation of the system.

10.3 Classification of Complexity Functions

The complexity matrix $C_{\text{comp}_a}$ in Figure 10.1 consists of $n \cdot m$ complexity functions $f_{ij}$ (Figure 10.2). It must be decided whether the complexity functions should be characterised

- independently, or
- as vectors that correspond to lines in the complexity matrix

A vector of complexity functions represents the number of times an operation of a superior component invokes each operation of the inferior components. These complexity functions are correlated because these suboperations together implement the operation. This is one argument in favour of classifying vectors of complexity functions. However, this increases the number of dimensions of the classification scheme. Moreover, the purpose of this chapter is to indicate which dependencies that can be represented by a complexity function. Therefore, complexity functions are treated individually.

![Figure 10.1: A complexity matrix which relates two work model components](image)

The execution of an operation provided by a software component may be dependent on data values which are defined externally of the component (see also [92, pp. 189–197]). Examples of such data values are operation arguments, global variables, and the content of databases. In addition, the execution of a function may be dependent
on data structure information, e.g., database organisation: indexes, hash tables, clustering of tuples. In procedural programming languages such as C, the head of a procedure definition contains formal parameters which are instantiated by calls to the procedure. Each formal parameter may have a set of possible values. Each combination of parameter values often results in to different behaviour of the software component. It is also possible that an operation is applied to several different internal data structures (e.g., subsets of a large data structure). In some cases an operation is state-dependent (see Section 2.4) because the execution of the operation is dependent on the state described by the component’s data.

Data-dependent operations are modelled by parameterised complexity functions in order to account for the varying conditions under which the operations are executed. Parameterisation of complexity functions increases the reusability of complexity functions. In addition, parameterisation enables operations of several superior components to invoke the same operation of an inferior component by means of different parameter settings¹ (see discussion in Section 6.5).

The type and degree of data dependency of operations are the main criteria for classification of such operations. Each complexity function must be classified according to:

- $n_p$, the number of parameters in the complexity function, i.e., how many factors influence the execution of the operation.
- the form of the complexity function, i.e., how do the parameters influence the complexity function, e.g., linear or logarithmic.
- $n_c$, the number of coefficients in the complexity function.
- $N_{\text{max}}$, the number of possible combinations of parameter values for all the $n_p$ parameters.
- $N_{\text{use}}$, the number of combinations of parameter values that is encountered in the software to be characterised with respect to performance. $N_{\text{use}} \leq N_{\text{max}}$. $N_{\text{use}}$ is specific to the environment in which a component is used. Its value can be found by program inspection or by measurement.

¹Unfortunately, parameterised invocations inhibit calculation of devoted work by matrix multiplications and additions.
These characteristics must be collected by source code inspection and/or preliminary measurement runs. In particular, these measures determine the cost of measurement experiments and measurement analysis as well as the reusability of the complexity functions.

Some operations cannot be represented properly by a complexity function because the required information is not available. In other cases there are inherent problems which prohibit the determination of a complexity function, even if all required information is available. This is the case for the Quicksort algorithm which is extremely dependent on the data to be sorted. The number of different combinations of data values to be sorted is unlimited for all practical purposes. However, it is feasible to find a complexity function for Quicksort of an instance of a data set if the data set is stable over time, i.e., deletions and additions are rare. Another solution would be to assume the worst case, e.g., that Quicksort is $O(n^2)$. This is similar to other areas of science and engineering where it is not uncommon that a solution of a problem can be outlined but the practical solution is not at all feasible. For instance, it may take a so long time to factorise a very big integer that it cannot be done even though it is straightforward to devise how the problem could be solved if the computers were fast enough.

10.4 Types of Complexity Function Parameters

The following types of dependency can be represented by parameters of complexity functions:

- size of (subset of) data to which the operation is applied
- data values to which the operation is applied
- data structure information, e.g., presortedness, the existence of indexes, the type of index, organisation of database tables
- state of the component whose operation is invoked
- query selectivity
- user behaviour, e.g., menu navigation
- buffering effects which are independent of concurrency. Logical suboperations are usually characterised without taking account of buffering, while the opposite is true for physical suboperations. Buffering effect will have to be represented probabilistically, which may cause problems for small invocation counts.
- concurrency effects (load dependencies), e.g., contention for shared memory or the overhead caused by thread administration in multi-threaded software components.
features/versions of the component

Some of these parameters are numerical while other parameters are categorical parameters which turn on (1) or off (0) a non-numerical effect on the execution of the operation.

10.5 Examples of Complexity Functions

The simplest complexity function is just a constant, i.e., the number of times an operation invokes one of its suboperations is independent of the context in which the invocation takes place.

\[ C = k_1 \]

Another common complexity function is on the form:

\[ C(n) = k_1 \cdot n + k_2 \]

where \( n \) is a parameter. It is used in cases where \( n \) represents the amount of data to be processed, \( k_1 \) represents the demand for computation for each unit of data, and where \( k_2 \) represents the demand for computation caused by overhead, e.g., initialisation of data structures.

A complexity function which describes the "raw" measurement data directly, has the following form:

\[ C(b_1, \ldots, b_n) = b_1 \cdot k_1 + \ldots + b_n \cdot k_n \]

where \( b_1, \ldots, b_n \) are categorical parameters. The conditions for these parameters should be mutually exclusive in the sense that one and only one of \( b_1, \ldots, b_n \) is true for any invocation from an operation to one of its suboperations. This type of complexity is the last resort if no functional relationships can be found between resource demands during executions with different parameter settings. The number of categorical parameters is in this case equal to \( N_{\text{size}} \).

When nothing is known about the complexity function, the following trivial complexity function is used:
10.5. Examples of Complexity Functions

\[ C(p) = p \]

i.e., the complexity function is just a parameter whose value must be measured each time the “complexity function” is used.

![Set owners and set members diagram](image-url)

**Figure 10.3:** Sets with varying number of set members

In [22] complexity functions are derived for set operations in CODASYL databases. Each instance of a set has one and only one owner record and it has zero or more member records. Complexity functions for the average number of records that have to be read in order to find a specific record within a set. The derivation uses \( f(z) \) which is the fraction of the sets which have \( z \) members. The sets in Figure 10.3 have fractions \( f(0) = 0.2, f(1) = 0.2, f(2) = 0.4, \) and \( f(3) = 0.2 \). \( f(z) = 0 \) when \( z = 4, 5, \ldots \) If all the sets are accessed with equal probability the following results are derived. \( b_{\text{exist}} \) is a categorical parameter which is true when the searched record exists, and \( b_{\text{not-exist}} \) is a categorical parameter which is true when the searched record does not exist.

\[
a = b_{\text{exist}} \cdot \frac{1}{2} \sum_{z=1}^{\infty} f(z)(z + 1) + b_{\text{not-exist}} \cdot \sum_{z=1}^{\infty} f(z)z
\]

If all the *non-empty* sets are accessed with the same probability the following results can be derived.

\[
a = b_{\text{exist}} \cdot \frac{\sum_{z=1}^{\infty} f(z)(z + 1)}{2(1 - f(0))} + b_{\text{not-exist}} \cdot \frac{\sum_{z=1}^{\infty} f(z)z}{1 - f(0)}
\]

If all records, regardless of sets, are accessed with the same probability:

\[
a = b_{\text{exist}} \cdot \frac{\sum_{z=1}^{\infty} f(z)(z + 1)z}{2 \sum_{z=1}^{\infty} f(z)z} + b_{\text{not-exist}} \cdot \frac{\sum_{z=1}^{\infty} f(z)z^2}{\sum_{z=1}^{\infty} f(z)z}
\]

These complexity functions are based on statistical treatment. The parameters are the fractions \( f(z) \) (where \( z = 0, 1, 2, \ldots \)) and the categorical parameters \( b_{\text{exist}} \) and \( b_{\text{not-exist}} \). In [17] other complexity functions are formulated based on the complexity functions shown above.
10.6 The Parameter "Curse"

While parameterisation of work model components and operations increases their reusability, it also introduces some problems. The performance characteristics of work model components and operations are not completely specified before the execution subject to all combinations of parameter values, $N_{\text{max}}$, have been examined. If $N_{\text{max}}$ is large, it may not be possible or cost-effective to examine all the combinations of parameter values. In such cases no complete performance specification is obtained and consequently the reusability is limited. In some cases it may be more appropriate to have fewer parameters which can be measured for all combinations of parameter values in order to obtain a complete specification of the operations of a component.

There is an interesting analogy between performance specification and testing. The more features the software or hardware components have, the more test cases must be performed before one can have some confidence in the component. Even if the actual behaviour of the component is not the main interest during testing, it is still necessary to execute the functions with different parameter settings in order to check that the results are correct. In fact, measurements could be carried out while each test case is run with little extra cost. Software testing is treated in, e.g., Pressman [85] and Fenton [40].

10.7 Examples from Case Studies

The following examples illustrate typical data dependencies that must be resolved by parameterisation of complexity functions.

The Client-Server Example If the server is capable of returning more data than can be packaged in one message, multiple remote procedure calls are necessary. The first remote procedure call initiates a query and the first chunk of data is returned. The number of remote procedure calls that are required to fetch the remaining data is dependent on the size of the result of the query. The size of the data selected by the query is dependent on the size of the file or database table, and the selectivity of the query. Hence, complexity functions must at least be parameterised with respect to size of the data and the selectivity of the query.

The server may optimise reading from the disk by assuming sequential read. This is possible by reading a several records in one disk access. This is called look-ahead buffering. Hence, complexity functions which describe the number of disk accesses or the number of operation system calls must be parameterised with respect to the number of records read in each disk access.

2In some cases the correctness of a program can be tested by formal methods which do not require execution of the program
The ICL System  In a DAIS system, all queries to a database is sent to the database server. This server will parse the incoming query and divide it into several subqueries. Each subquery is sent to a different part of the distributed (and possibly heterogeneous) database system. The CPU usage required by the parsing of the query is dependent on the query. Which parameters can describe the parsing process properly? How is the complexity function which represents the invocation of a database system, parameterised?

The Tandem System  For each terminal that is connected to Tandem computer there is a buffer both in the terminal itself and on the computer, which contains the contents on the terminal screen. Furthermore, the contents of several screens can be stored in these buffers. When the screen is updated, only the changes are sent across the terminal communication line. Consequently, the performance of screen updates are improved considerably. This buffering mechanism must be taken into account in the complexity functions which represent the number of messages that are transmitted to the terminal.

10.8 Overall Considerations

There is an important difference between declarative and operational operations. Declarative operations are invoked by issuing a description of the desired result. Hence, just one or a few operations are necessary. On the other hand, an operational operation tends to provide a particular service which is explicitly requested by the superior components. Hence, many operational operations are required. Operational operations lend themselves easily to representation as complexity functions because the need for parameterisation is reduced.

As an example, consider the difference between CODASYL databases and relational databases. CODASYL databases leave the navigation in the database to the application program which invokes operations like find next in set, find set owner, and fetch using index. When relational databases are used, the application programs specify the required result as an SQL query, pass it to the database management system, and collect the results. The application programs do not influence the execution of the query.

There is also a difference between infrastructure components and application-dependent components. Infrastructure components must provide general services since it may be invoked from many different components. In addition, the same component may appear in many different systems. Hence, extensive parameterisation is necessary when the operations of such components are characterised with respect to performance. Application-dependent components are often tailored to a particular system and they are represented by work model components which appear towards the top of composite work models. Hence, the need for parameterisation of complexity functions is reduced.
Chapter 11

Formulation of Composite Work Models

Many aspects of a composite work model must be taken into account when its granularity and degree of parameterisation are chosen. Composite work models should be formulated according to the rules of the SP method (Chapter 6). This method structures the composite work model according to fundamental functionality which must be present in any information-processing system [58]. When a composite work model has been proposed, all relationships between work model components must be assessed with respect to the feasibility of doing the necessary measurements. If some of the required measurement data cannot be obtained, another composite work model must be formulated. The composite work model must also be compatible with the contention model. Even if all the required measurement data can be captured, the cost of performing the measurement experiments may be prohibitive. By assessment of the granularity and degree of parameterisation of the proposed composite work model, it is possible to assess cost, accuracy, and reusability before extensive measurement experiments are performed. The rationale for these preliminary assessments is to avoid superfluous measurement experiments. The final choice of granularity and degree of parameterisation is based on a trade-off between reusability, cost and accuracy of the composite work model and the corresponding measurement experiments. By now it should be evident that composite work modelling and measurement experiments are closely related:

| Formulation of a composite work model is at the same time planning of measurement experiments |

The feasibility and cost of measurement experiments limits composite work modelling while composite work modelling aim at capturing reusable and accurate measurement data.

The planning that is necessary to obtain an appropriate formulation of a composite work model proceeds as follows:
1. Propose a composite work model of the system according to the rules of the SP method.

2. Assess the feasibility of doing the necessary measurements. If not feasible, formulate another composite work model in step 1.

3. Assess the compatibility with the contention model. If not compatible, formulate another composite work model in step 1.

4. Determine the form of complexity functions required to complete the proposed composite work model.

5. Assess the reusability of operations and components in the proposed composite work model.

6. Assess the risk of error propagation in the proposed composite work model.

7. Assess the measurement cost according to the proposed composite work model.

8. Repeat from step 1 until enough alternative formulations are evaluated.

9. Find a trade-off between the accuracy, cost, and reusability of the alternative formulations of the composite work model. Select the most appropriate formulation according to this trade-off.

The rest of this chapter is structured according to these steps.

11.1 Propose a Composite Work Model

The relationships between the resource demands of the software components are modelled in a composite computational work model. The most probable changes to the work model must be taken into account when the work model components are defined. An important requirement is that:

| Locality of change in software components should imply locality of change in work model components |

The composite work models should be formulated according to the rules of the SP method (Chapter 6). As mentioned earlier, composite work models obtained by enforcing the SP rules tend to be similar to composite work models of other systems which have the same architecture. This increases the probability of reusing work model components.
11.1. Propose a Composite Work Model

The first step in composite work modelling is to regard the software as one big component and determine which operations (or services) that the software provides to the user. Then the hardware devices and their operations are defined. In the simplest possible model that is obtained, the monolithic software component is connected to all the hardware devices in the work model. As an example, the initial work model of the client-server system is shown in Figure 11.1. In the next steps of composite work modelling the top-level operations and the hardware components and their operations will remain fixed while the number of layers and operations at intermediate components will be different in different formulations.

![Figure 11.1: The initial work model of the client-server example](image)

The next step is to decompose component-subcomponent relationships according to the SP method in order to introduce additional layers in the composite work model. As an example, the relationships between the bank application component and the CPU#2 and Disk subcomponents are decomposed. The result is shown in Figure 11.2, where the file server component is introduced. Note that several component-subcomponent relationships have been changed. The result of one more decomposition of the work model in Figure 11.2 is shown in Figure 11.3.

![Figure 11.2: The result of decomposition in the client-server example](image)

The decomposition step may be applied recursively until no information about additional software components is available. The granularity of a composite work model is made finer as additional layers of components are introduced in the composite work model. The reason is that the new intermediate components introduce new operations, and consequently new operation-suboperation relationships, into the model. If the number of operations on the new intermediate components is large, the performance characteristics of (some of) the operations of a software component may have to be aggregated so that similar operations appear as one operation in the composite work model. If an operation behaves very differently depending on factor
settings, several performance characterisations of the operation may be necessary in the composite work model. Operations which initialise, initiate, or terminate an activity which lasts longer than a top-level operation, e.g., a transaction, are not included in a composite work model.

Practical composite work modelling is treated in detail in Chapter 12.

Decomposition of Component-Subcomponent Relationships

The rationale for decomposition of component-subcomponent relationships in composite work models is to characterise an operation in terms of (new) suboperations. This decomposition of operations should result in suboperations that are common to several operations. Then it is possible that suboperations can be measured and characterised for the benefit of several operations in the model. Another benefit is that reuse of performance characterisations of operations in several composite work models becomes feasible. A consequence of the decomposition of operations is that new components are introduced in the composite work model. There are two main problems concerning decomposition of component-subcomponent relationships:

- The number of operations which is provided by new work model components may be large.
- Each operation may have to be generalised to be usable for more than one superior operation. Generalisation is carried out by parameterisation as described in Chapter 10. The complexity functions which describe how the operation invokes its suboperations may refer to these parameters.

A strict relationship between complexity matrices must be preserved when a component-subcomponent relationship is decomposed. In Figure 11.4 the relationship between Comp.#1 and Comp.#2 is decomposed into a relationship between Comp.#1.1 and
Figure 11.4: The relationship between complexity matrices after a decomposition of a component-subcomponent relationship

Comp. #1.2, and a relationship between Comp. #1.2 and Comp. #2. Consequently, the complexity matrix $C$ must be decomposed into $C_1$ and $C_2$. It is required that $C = C_1 \cdot C_2$. In the client-server example the $C_{\text{CPU#2}}$ complexity matrix in Figure 11.2 must be equal to $C_{\text{file-server}} + C_{\text{file-server}} \cdot C_{\text{file-system}}$ in Figure 11.3.

Splitting of Operations of a Work Model Component

In some cases it may be desirable to split operations of a work model component. Such cases include:

- The initialisation and termination phase of an operation may be represented as operations on their own
- An operation may be split into one operation for each data set on which it operates
- An operation whose behaviour depends on information about the state of a software component may be split into one operation for each possible state that can be described by the component’s data.

The advantage of splitting of operations is that parameterisation can be avoided. However, the disadvantage is that more operations are defined for the work model component.
Aggregation of Operations of a Work Model Component

The rationale for aggregation of operations on work model components is to reduce the number of operations that have to be characterised explicitly by complexity functions.

Aggregation of operations on work model components requires that they invoke the same suboperations, and the corresponding complexity functions must have the same form and must be dependent on the same parameters. However, the coefficients may be different. Load dependent complexity functions cannot be aggregated unless equal load on all the aggregated operations is guaranteed. The complexity functions that represent the aggregated operations may be determined by sampling from a (preferably large) collection of measured measurement data about the operations to be aggregated. However, the reusability of the complexity functions will be poor. Alternatively, either averaging, weighted averaging, or numerical clustering could be applied. The aggregation techniques must be applied with care because the operations may not form suitable groups. Erroneous application of aggregation may be detrimental to the accuracy of the composite work model.

Aggregated operations must be given a description of their functionality. Without such a description it is impossible to relate operations of the superior components to the aggregated operations.

The need for aggregation is greatest for components which provide a lot of operations where the operations can be grouped according to their resource demands. This is often the case for software components that are represented by top-level work model components. An example of aggregation of top-level operations is presented in Section 12.3.

11.2 Assess the Feasibility of Measurements

It is important to examine as early as possible whether the required measurement data can be measured at all. If not, a new composite work model must be formulated.

Measurable Components

It must be possible to collect measurement data in terms of the work model com-
ponents. Unfortunately, there is frequently a mismatch between work model compo-
ponents and measurable components. Typical relationships between work model compo-
ponents and measurable components are shown in Figure 11.5. Measurable compo-
nents were surveyed in Section 7.3. For instance, if a work model component rep-
resents a process and a process is a measurable component as in UNIX, it is straig-
tforward to capture the measurement data to characterise the work compo-
11.2. Assess the Feasibility of Measurements

This situation corresponds to a perfect match as depicted in Figure 11.5a. If there is a mismatch as in Figure 11.5b–d, one of the following actions must be taken:

- Change the composite work model so that work model components correspond to measurable components.
- The mismatch in Figure 11.5b requires that the measurable component must be instrumented in order to measure software interactions and hardware device usage of its different parts separately.
- The mismatch in Figure 11.5c requires aggregation of the measurement data from each measurable component.
- The mismatch in Figure 11.5d requires both instrumentation and aggregation.

![Diagram](image)

**Legend:**

- a work model component
- a measurable component

**Figure 11.5: Relationships between work model components and measurable components**

For instance, the software in Tandem computers is distributed among hundreds of processes (see Section 3.3). Even if a decision is made to collapse software components to form work model components, measurement of resource demands and software component interaction must be performed in terms of processes. It is not possible to hide complex interactions by encapsulating them in a few components.
This “feature” or flexibility of Tandem computers is detrimental to composite work modelling of Tandem software. The Tandem case corresponds to the case in Figure 11.5c.

**Measurable Operations on Measurable Components**

Invocations of software components may be measured at different levels of detail. Six possible levels of detail are depicted in Figure 11.6. Composite work modelling requires that the operation which invoke a suboperation and the suboperation itself can be measured. This corresponds to the situation in Figure 11.6f. The case in Figure 11.6a is that only the total number of invocations of a measurable component can be captured. The origins of the invocations are unknown. In Figure 11.6b the total number of invocations in terms of invoked operations can be captured. Still, the origins of the invocations are unknown. In Figure 11.6c interactions between components can be counted, but neither the invoking operation nor the operation being invoked can be captured. In Figure 11.6d invocations can be measured in terms of operations being invoked and the origin of the invocations. In Figure 11.6e the total number of invocations of the inferior component caused by invocations of an operation of the superior component can be captured.

![Diagram](image)

**Legend:**
- Invocation
- Operation on measurable component
- Measureable component

*Figure 11.6: Details in measurement of invocations between components*

The case in Figure 11.6c corresponds to the level of detail that can be measured by the MEASURE tool provided by Tandem (see description of measurement tool in Section 8.1).
11.3 Assess Compatibility with Contention Models

A composite work model calculates parameter values that are required by contention models in order to describe the software which is executed by the computer system. A prerequisite for successful composite work modelling is that the work model results, the work devolved on each computer resource, are compatible with the parameter needs of the contention model. The results must be usable as parameter values, either directly or after a suitable derivation.

Each stage in the execution of a transaction must correspond to a linear combination of operations that are explicitly defined as operations in the composite work model. As described in Section 4.2, transactions may compete for both hardware devices and software resources such as software servers and database locks. The contention model will focus on contention points, i.e., bottlenecks. Thus, the resource demands for different transactions will be needed in terms of critical hardware devices and critical software resources. The implications for composite work modelling are that all operations on hardware devices and software servers, for which there can be contention, must be represented explicitly in the corresponding composite work model.

The more levels in the composite work model, the more operations are modelled explicitly, and therefore the more flexible the parameter capture. A detailed discussion of compatibility between composite work models and contention models can be found in Chapter 14.

11.4 Establish the Form of Complexity Functions

The parameterisation of complexity functions influences the reusability of the complexity functions, the risk of propagation of inaccuracies, and the cost of doing the necessary measurement experiments. These effects will be discussed in the following sections. In Chapter 10 several types of parameter were presented. Some descriptors of complexity functions such as $\mathcal{N}_{\text{max}}$ and $\mathcal{N}_{\text{asr}}$, were also presented.

At this stage of the formulation of a composite work model, the form or at least the parameterisation of all the complexity functions in the proposed composite work model should be selected. This selection of parameters is a modelling decision. In addition, $\mathcal{N}_{\text{max}}$ and $\mathcal{N}_{\text{asr}}$ should be estimated. The values of $\mathcal{N}_{\text{max}}$ and $\mathcal{N}_{\text{asr}}$, however, are not defined by a modelling decision. Their values are a consequence of the selection of parameters.

It may be difficult to specify the form of a complexity function before measurement, in particular if the source code is either unavailable or if it is too complex to analyse. Therefore a special notation is needed to specify the parameters and to avoid extensive indexing of names of constants and complexity functions. In addition, complexity functions that are not data dependent should be clearly marked because
Chapter 11. Formulation of Composite Work Models

ey prevent propagation of inaccuracies in a composite work model. In this thesis, data-independent complexity functions will be surrounded by a \text{frame}. The notation is as follows:

\begin{align*}
&k & \text{a constant to be determined by measurement} \\
&f(N) & \text{a complexity function which depends on a parameter } N
\end{align*}

If the constant or the complexity function is known (e.g., by code inspection), it is included verbatim in the complexity matrices. Note that $f(N) \cdot f(N) = f(N)$, $f(A) \cdot f(B) = f(A, B)$ and $f(A) + f(B) = f(A, B)$ where $f$ denotes some complexity function, and that $A$, $B$, and $N$ are variables. The scope of names of constants and complexity functions is the complexity function itself. However, parameter names are global. Hence, $f(N)$ and $f(N)$ may be different complexity functions but they are dependent on the same parameter, $N$.

The complexity functions which describe all the relationships between a superior and an inferior work model component are represented as a complexity matrix. For instance, the matrix:

\begin{equation*}
C_{\text{superior}}^{\text{in inferior}} = \begin{bmatrix}
\text{op\#1} & \text{subop\#1} & \text{subop\#2} \\
\text{op\#2} & \begin{bmatrix} 2 \\ 100 \end{bmatrix} & \begin{bmatrix} 3 \cdot N + 10 \\ N^2 + 3 \end{bmatrix}
\end{bmatrix}
\end{equation*}

may be represented as the matrix

\begin{equation*}
C_{\text{superior}}^{\text{in inferior}} = \begin{bmatrix}
\text{op\#1} & \text{subop\#1} & \text{subop\#2} \\
\text{op\#2} & \begin{bmatrix} 2 \\ k \end{bmatrix} & \begin{bmatrix} f(N) \\ f(N) \end{bmatrix}
\end{bmatrix}
\end{equation*}

according to the notation that was introduced above. A composite work model consists of several complexity matrices. For instance, the composite work model in Figure 11.3 needs 8 complexity matrices because there are 8 component-subcomponent relationships. The complexity matrix which represents the relationship between the File server and the File system components may be as follows:\footnote{The functionality of these components is outlined in Appendix C}:

\begin{equation*}
C_{\text{superior}}^{\text{in inferior}} = \begin{bmatrix}
\text{op\#1} & \text{subop\#1} & \text{subop\#2} \\
\text{op\#2} & \begin{bmatrix} 2 \\ f(N) \end{bmatrix} & \begin{bmatrix} f(N) \\ k \end{bmatrix}
\end{bmatrix}
\end{equation*}
11.5 Assess Reusability

Reuse of complexity functions that are derived from measurement data is the means with which the cost of performance characterisations of software components is reduced. An important advantage of reuse of complexity functions, in addition to the potential cost savings, is that the time needed for parameter capture for contention models can be reduced considerably.

Composite work modelling requires explicit measurement of interactions between software components. This involves capture of lots of measurement data that will be useful only if the model can be changed easily and used in several performance modelling studies. This section presents the main factors that influence the reusability of complexity functions and measurement data for software components.

Software Component Reusability

The first criterion for reuse of measurement data and complexity functions is of course that the software components themselves are reusable. There are several trends that indicate that reuse of software components will be feasible. It is becoming common to build software by putting together existing components. They are called reusable components if they can be reused separately in different environments. This implies that reusable components tend to be independent, i.e. that one such component may be replaced by another component which provide the same functionality. Another important implication of this trend is that changes in the software normally will concern one or a few components ("locality of change"). Another trend is the principle of "evolution not revolution" [68]. Existing information systems are not rebuilt from scratch but integrated to form distributed heterogeneous computer systems. Yet another trend is open systems [47], which provide the
system developers with a choice between system components with standardised interfaces. This trend also enable an exchange of a system component with a new one with identical interface specification. The main issue for reusability in this thesis is to formulate a composite work model where the components represent reusable software components that have a high probability of reuse [24].

Reusability of Performance Characteristics

The second criterion concerns the reusability of measurement data for reusable software components [75]. It should be emphasised that reuse of performance characteristics of components requires more than software reuse. Even a minor change in a software component which does not affect the component's functionality may have dramatic effects on the performance characterisation of the component. Reusability is limited when an operation cannot be characterised properly by a complexity function. The robustness is better when the complexity functions are based on inspection of source code than when the complexity functions are based on black-box measurement experiments because it is easier to determine the form of complexity functions when source code is available.

Reuse of complexity functions requires proper parameterisation so that the complexity functions can be adjusted to new environments when they are reused. It is also important to consider the ease of reuse. Some parameters are easy to determine, e.g., block lengths, while other parameters are difficult to determine, e.g., query selectivity in databases and presortedness of data to be sorted. If the complexity functions of a work model component depend on many different parameters, the reusability of the work model component is limited. Barnes and Bollinger argue in [11] that extensive parameterisation of software components can be regarded as a mini-language. Thus the human effort required to reuse the software component using this mini-language may be just as big as the effort required to re-implement the component. This is analogous to reuse of parameterised complexity functions, which must be re-measured if reuse fails. If some of the operations are split (see Section 11.1) to account for data-dependencies, the number of operations as a result of the split may be different in the new context of the component. This requires modification and, unfortunately, new measurement experiments. Even worse, reuse of a work model component may require changes in other components in the work model. In summary, given that reuse of complexity functions and measurement data is possible, the reusability of a work model component is assessed by the number of extra measurement experiments and changes of other work model components that have to be undertaken when it is reused.

The ideal operations with respect to reusability of the corresponding complexity functions have the following properties:

- The operations are applied to the same data or data of the same size each time the operations are invoked
• The operations are not applied to different subsets of data if the corresponding resource demands vary significantly

• The data on which the operations are applied do not contain a description of a state which influences the execution of the operation

Unfortunately, the preferred types of operation are not frequently encountered because software infrastructure operations typically offer the same service to the entire application and therefore must be very general. Thus, these operations are applied to various input data and various sizes of input data.

11.6 Assess Risk of Error Propagation

The main issue here is to describe the consequences for the accuracy of devolved work when the granularity and the degree of parameterisation are varied. Calculation of the actual accuracy of a composite work model must be postponed until the measurement experiments have been performed.

A composite work model is a static representation of the number of times software components invoke each other and the number of times that software components invoke operations on hardware devices. Ideally, no modelling approximations or simplifications should be necessary and models with perfect accuracy should be obtainable. However, as pointed out in [55], there are some inherent problems of workload modelling which necessitate introduction of approximate complexity functions based on measurement data:

• Functional variety requires aggregation of the performance characteristics of operations.

• Data-dependencies require introduction of parameters. Thus, functional relationships between the parameter settings and the computer resource demands must be found, i.e., complexity functions are needed. The coefficients of complexity functions may be inaccurate as well as the parameter values themselves.

• State-dependence may require that the performance characteristics must be found by an average (aggregate) over sets of states which have similar influence on the devolved work of the software components.

• Sequence-dependence must often be ignored at least when long-run effects are modelled.

Thus, complexity functions are approximate in practice. The main question is how the inaccuracies of complexity functions propagates in a multi-layer model. The risk of error propagation is related to accuracy in the worst case. In other words, if the worst case accuracy is acceptable, there is no risk of significant error propagation.
Figure 11.7: Two composite work models. The model in b) is a more
detailed version of a). All components have just one operation.

If the effect of parameterisation on the accuracy of a complexity function is not
represented explicitly, the accuracy of a complexity function is represented as a
value and a *deviation coefficient*:

\[
c \pm c \cdot p
\]

where \( c \) is the value of the complexity function and \( p \) represents the deviation coeffi-
cient, i.e., the deviation as a fraction of \( c \). This corresponds to the simple work model
in Figure 11.7a. If this complexity function is replaced by two complexity functions
which occur on two adjacent layers in a composite work model (Figure 11.7b), the
accuracies of the new complexity functions are as follows:

\[
c_1 \pm c_1 \cdot p_1
\]

and

\[
c_2 \pm c_2 \cdot p_2
\]

The values of the two complexity functions must be multiplied in order to calculate
the devolved work. Ideally, \( c = c_1 \cdot c_2 \) but if the inaccuracies are taken into account:

\[
(c_1 + c_1 \cdot p_1) \cdot (c_2 + c_2 \cdot p_2) \\
= c_1 c_2 + c_1 c_2 (p_1 + p_2) + c_1 c_2 p_1 p_2 \\
\approx c_1 c_2 + c_1 c_2 (p_1 + p_2) \\
= c + c \cdot (p_1 + p_2)
\]

The approximation is valid when the product \( p_1 \cdot p_2 \) is small. If \( p_1 + p_2 > p \) then
accuracy is lost because of the extra layer in the composite work model. Similar
calculations can be performed for the lower bound. These calculations are trivial, but they lead to an interesting observation: The accuracy of the devolved work along one operation-suboperation path in a composite work model is approximately equal to the sum of the deviation coefficients in the worst case (regardless of the values they deviate from). Hence, the more layers in the composite work model, the more accurate the complexity functions must be in order to avoid serious propagation of errors.

There may be several operation-suboperation paths between a top-level operation and a primitive operation. What is the accuracy of the devolved work when the devolved work for each path is added? If the devolved work along two paths are:

$$\Pi_1 \pm \Pi_1 \cdot \Sigma_1$$

and

$$\Pi_2 \pm \Pi_2 \cdot \Sigma_2$$

and their values are added, the sum becomes

$$\Pi_1 + \Pi_2 \pm (\Pi_1 + \Pi_2) \cdot \frac{\Pi_1 \cdot \Sigma_1 + \Pi_2 \cdot \Sigma_2}{\Pi_1 + \Pi_2}$$

where $$\Pi_1$$ and $$\Pi_2$$ are the devolved work for path #1 and path #2 in the ideal case, $$\Sigma_1$$ and $$\Sigma_2$$ are the added deviation coefficients along each path, and where $$\frac{\Pi_1 \cdot \Sigma_1 + \Pi_2 \cdot \Sigma_2}{\Pi_1 + \Pi_2}$$ is the new deviation coefficient. It can be shown that the new deviation coefficient will always be between $$\Sigma_1$$ and $$\Sigma_2$$ (including the endpoints). If $$\Pi_1 \gg \Pi_2$$, then the deviation coefficient is approximately $$\Sigma_1$$ and vice versa. If $$\Pi_1 \approx \Pi_2$$, then the deviation coefficient of $$\Pi_1 + \Pi_2$$ is approximately the average of the deviation coefficients of $$\Pi_1$$ and $$\Pi_2$$.

In some cases a complexity function is just a constant which can be determined with absolute certainty. For instance, it may be trivial to find out that an application component invokes the init.screen operation of the user interface component exactly once during the execution of a transaction. Such complexity functions does not degrade the accuracy of a composite work model.

These observations can be extended to paths which contains more than two layers and to more than two paths. Therefore:

- The worst-case accuracy degradation along a path is approximately additive and it is independent of the actual values of the complexity functions.
The accuracy of added paths is defined by the path with the smallest added error and by the path with the greatest added error. The actual accuracy is dependent on the actual values of the complexity functions, not only the deviation coefficients.

The "rule-of-thumb" seems to be that introduction of one extra layer of work model components in a composite work model may degrade the accuracy of the model significantly. One the other hand, the introduction of new operations on work model components is not critical with respect to accuracy.

So far, only the influence of the granularity (number of layers and number of paths) on the propagation of errors has been discussed. The influence of parameterisation of complexity functions should also be taken into account. This is because the parameter values may be error prone. It may be easy to find the exact message length that are used in communication, but finding parameter values for query selectivity [59] and presortedness [67] of a database table with respect to a sort key may be difficult, costly, and error prone. Moreover, the accuracy of parameter values influences the accuracy of the coefficients because the coefficient values are determined based on analysis of the parameters values (e.g., regression analysis).

A statistical treatment is necessary to determine the effects of the parameters with respect to the accuracy of a complexity function. Such treatment requires the statistical distributions of the possible values of the parameters. Such information was not captured during the case studies and therefore the influence of parameter accuracy on the accuracy of complexity functions are considered as further work.

### 11.7 Assess Measurement Cost

The cost of measuring software interactions is significant because instrumentation, recompilation, profiling, administration of large volumes of measurement data, extraction and analysis of measurement data take relatively long time and expertise is needed. It may be difficult to decide on which measurement data to collect, and it takes a long time to execute transactions or parts of transactions in varying contexts during the measurement experiments. It is evident that the extra cost of measuring software interactions is not acceptable if the measurement data cannot be reused several times. Note that extensive tool support for these activities is feasible. However, each measurement run and the analysis of the measurement data still need considerable human intervention.

Consequently, the cost of measurement for composite work modelling is primarily dependent on the degree of reuse, the number of measurement runs, and the number of operation-suboperation relationships that has to be analysed.
11.7. Assess Measurement Cost

The Number of Measurement Runs

The number of measurement runs is an important factor of measurement cost because each measurement run must be prepared, performed, and the captured measurement data must be prepared for analysis. Measurement runs can be performed under a production workload or under an artificial workload. The number of required measurement runs is dependent on which software and hardware events that can be captured by a measurement tool or by instrumentation. Also the amount of information that can be captured for each event influences the number of measurement runs. The requirements on measurement data will be discussed in Chapter 13. Individual transactions must be distinguished somehow; either the measurement tool can distinguish between transactions when software interactions and hardware usage are measured, or the transactions must be run one-by-one on an otherwise idle machine. In addition, the measurement tool must be able to provide measurement data that are sufficiently detailed to obtain complexity functions.

There are three categories of measurement data availability which influence the number of measurement runs:

1. Transactions can be distinguished and a complexity function for each operation-suboperation relationship can be derived from the measurement data. In this case only a few measurement runs of the production workload are necessary.

2. Transactions cannot be distinguished but complexity functions can be derived from the measurement data. In this case the number of measurement runs is dependent on the number of different transactions that are offered by the system.

3. Complexity functions cannot be accurately derived from the measurement data. In this case each operation of each component must be invoked one-by-one in a systematic manner. The number of measurement runs is dependent on the total number of operations on components in the system.

In order to obtain a complete performance characterisation of a component, a measurement run is necessary for each possible parameter combination. If some parameter combinations do not occur in the production workload, direct invocations of the component’s operations under these parameter combinations are necessary. However, it may be difficult to establish the correct parameter settings before measurement is carried out.

Reuse of measurement data has little influence on the number of measurement runs when measurements are carried out under production workload or when transactions are applied to the system one at a time. The reason is that transactions involve all or most of the components in the software. On the other hand, reuse of measurement data is beneficial when software components are measured one at a time.
The Number of Operation-Suboperation Relationships

The importance of taking into account the number of operation-suboperation relationships in composite work models is illustrated in Figure 11.8. In a) there are two components: A with 5 operations and B with 2 operations. There are \(5 \cdot 2 = 10\) relationships between an A operation and a B operation. If one assumes that the cost of measurement analysis is proportional to the number of relationships that must be analysed, 10 is an indication of the cost of analysis in a). In b) there are 5 component-subcomponent relationships. The total number of operation-suboperation relationships in b) is \(5 \cdot 3 + 3 \cdot 2 + 5 \cdot 2 + 5 \cdot 4 + 4 \cdot 2 = 59\). These numbers show clearly that the model in b) is much more expensive to analyse than the model in a). If the workload model is to be used once and then discarded then alternative a) must be chosen from the cost point-of-view. If model b) was built with the intention to use it several times, several changes can be done in the model. For instance, if the implementation of component C is changed, the \(3 \cdot 2 = 6\) operation-suboperation relationships between C and B must be re-measured. This change can not be done in model a). Instead all the \(5 \cdot 2 = 10\) operation-suboperation relationships must be measured, i.e., the measurements are done from scratch. Note that the number of top-level operations (=5) is the same in both a) and b). Also the number of bottom-level operations (=2) is the same in a) and b).

![Figure 11.8: The composite work model in b) is a more detailed version of a).](image)

When the number of operation-suboperation relationships are counted, all relationships are given equal weight. The number of factors involved in an operation-suboperation relationship may vary and the difficulty of the analysis may vary accordingly. Therefore a more elaborate estimation of analysis cost may be necessary. However, this is not pursued further here.

Obviously, there is no need for an analysis if performance characteristics for an operation-suboperation relationship can be reused. As the library of reusable measurement data grows, the need for analysis should decrease.
11.8 Finding a Trade-off

The dependencies between the cost, accuracy, and reusability of a composite work model should also be assessed because these properties may conflict. A trade-off between these properties must be found. The trade-off is of course not necessary if the constraints on the modelling study are violated, i.e., the cost limit is exceeded or the model is too inaccurate.

The only reason why a composite work model should be used is that the work model may be maintained over a long period of time, and that work model components may be reused in several different composite work models. Thus, the long-term cost of parameter capture for performance modelling studies can be reduced. Consequently, the degree of reusability of the components of a composite work model is a crucial issue when the feasibility of composite work modelling is discussed. The primary rule should be to avoid spending money on measurement experiments which do not contribute to reusability of measurement data.

The worst-case accuracy of devolved work tends to increase when the number of layers in the model increases. As an example, assume that errors are evenly distributed among the layers in a composite work model and that the acceptable error of devolved work is 15% deviation. Then the allowed error in each layer is $0.15/n$ where $n$ is the number of layers. Hence, the allowed error on each layer may be disturbingly small. However, it is likely that the errors will cancel out\(^2\) to some degree. In addition, some complexity functions may be exact. Exact complexity functions make it possible to distribute the error of devolved work over fewer layers, i.e., greater error is allowed for the layers which do not contain exact complexity functions.

A major source of error is aggregated operations. Since the propagation of errors is not influenced by the number of operations on work model components, it may be advisable to do less aggregation of operations for the sake of accuracy. Unfortunately, this will increase the cost of composite work modelling.

The degree of reusability may be limited by the required accuracy. If the reusability of a composite work model demands several layers and the risk of error propagation increases with the number of layers, there is a conflict that must be decided by the trade-off between cost/reusability and accuracy.

---
\(^2\)This cancelling effect can not be investigated before the measurements have been captured
Chapter 12

Practical Composite Work Modelling

The composite work models can be classified in two dimensions: the granularity and the number of parameters that are introduced in the model. The demonstrations of practical work modelling in this chapter explains the properties of composite work models for selected combinations of granularity and number of parameters. The formulations of composite work models are discussed according to the steps of Chapter 11. Within each case study, all formulations of a composite work model have the same primitive (hardware) components and the operations on these components are the same. In addition, the operations on the top-level component(s) are the same in all formulations. The main difference between two formulations is the number of software layers that are modelled explicitly.

The structure of this chapter is as follows: For each of the three case studies a few alternative formulations of increasing level of detail are discussed. Emphasis is put on the difference between one formulation and the next formulation. Typographical conventions are described in the Preface and the conventions for representation of complexity matrices are explained in Section 11.4.

12.1 The Client-Server Example

The client-server example is described in Section 3.1. The UNIX operating system is assumed. The operations that are provided by the components are listed in Appendix C. The parameters that are introduced here are:
<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Description of parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>The number of accounts in the accounts file</td>
</tr>
<tr>
<td>$R$</td>
<td>The length of an account record</td>
</tr>
<tr>
<td>$M$</td>
<td>Maximum message length in RPC calls</td>
</tr>
<tr>
<td>$B_r$</td>
<td>Size of buffer that is used during read</td>
</tr>
<tr>
<td>$B_w$</td>
<td>Size of buffer that is used during write</td>
</tr>
<tr>
<td>$D_r$</td>
<td>Size of the disk cache for diskread</td>
</tr>
<tr>
<td>$D_w$</td>
<td>Size of the disk cache for diskwrite</td>
</tr>
<tr>
<td>$b_c$</td>
<td>Does open create a file?</td>
</tr>
<tr>
<td>$b_s$</td>
<td>Is writing synchronous?</td>
</tr>
</tbody>
</table>

The last two parameters are categorical parameters. If these parameters are used in all formulations where they are applicable, many complexity matrices will not change when the level of detail is increased.

Complexity matrices are named as $CV_{I}^{S}$ where $V$ indicates the number of the formulation to which the complexity matrix belongs, $S$ indicates the name of the superior component, and $I$ indicates the name of the inferior component in a component-subcomponent relationship.

**Formulation #1**

![Diagram of a minimal composite work model of the client-server example](image)

**Figure 12.1:** A minimal composite work model of the client-server example

**Formulation of the Model**  Given that CPU#1, CPU#2, the network and the disk are the hardware resources to be modelled, the composite work model in Figure 12.1 is the simplest possible model. There is just one non-primitive component which represents the entire software. Each primitive component represents a hardware device. According to the SP rules (Chapter 6), the Bank application component have processing relationships with CPU#1 and CPU#2. The fact that there is more than one processing relationship implies that processing is distributed. The Network component enables the distribution of processing between CPU#1 and CPU#2. Hence the Bank application component and the Network component have a communication relationship. The Disk provides persistent data storage. Consequently, there is a memory relationship between the Bank application and the Disk.
Feasibility of Necessary Measurements  The CPU usage can be measured by starting the client and the server with "time" ("time Bank_application" and "time file_server"). Then the CPU usage is output when the client and the server are stopped in a controlled way. The deficiencies of the "time" tool are that the process must be stopped for each measurement run, and that the overhead in starting and stopping processes is included in the measurement data. If it is possible to ensure that the two computers are not executing anything but the bank application, UNIX provides adequate measurement tools for measurement of disk and network activity. The tool "iostat" can provide CPU utilisation and the number of disk transfers while the tool "netstat" can provide information about the number of messages that was sent to and received from the network. If the resolution of the "time" tool is insufficient, it is possible to instrument the code (in this case both the client and the server code) to obtain the CPU time by means of the getrusage library function. Beware that some implementations of "getrusage" does not provide better CPU time resolution than "time". The "getrusage" library can also provide information about disk accesses and memory usage if it difficult to get hold of idle UNIX systems. The UNIX "trace" facility can be used to infer the number of disk accesses and the number of messages that was sent to or received from the network. This relationship between work model components and measurable components corresponds to case c) in Figure 11.5.

Compatibility with Contention Model  The model in Figure 12.1 is not compatible with a contention model that explicitly models contention for the File server as well as for the hardware devices as depicted in Figure 3.1 on page 32. The reason is that the composite work model cannot provide separate resource demand parameters for the client process and the server process.

Complexity Functions and Matrices  In the following, the complexity functions and the complexity matrices for the composite work model in Figure 12.1 are outlined.

\[ C_{Bank\_application}^{CPU\#1} = \begin{bmatrix}
\text{create} & f(R) \\
\text{read} & f(R) \\
\text{update} & f(R) \\
\text{delete} & f(R) \\
\text{custnam} & f(N, R, M) \\
\text{custbal} & f(N, R, M)
\end{bmatrix} \]

Assumptions concerning complexity matrix \( C_{Bank\_application}^{CPU\#1} \): Since the operations create, read, update, and delete on the "client side" only operate on one specific account, the CPU demand of each operation is approximately dependent on the data volume they process, i.e., the length of an account record, \( R \). However, custnam and
custbal list information about all \(^1\) \(N\) accounts. Therefore their CPU demand is dependent on the data volume which in this case is \(N \cdot R\). The implicit communication processing is dependent on how many messages that are sent to and received from the network. The maximum number of account records in one message is defined by \(R\) and \(M\).

\[
C_{1}^{\text{Bank\_application\_Network}} = \begin{bmatrix}
\text{create} & 2 \\
\text{read} & 2 \\
\text{update} & 2 \\
\text{delete} & 2 \\
\text{custnam} & f(N, R, M) \\
\text{custbal} & f(N, R, M)
\end{bmatrix}
\begin{bmatrix}
\text{send\_msg} \\
\text{rec\_msg}
\end{bmatrix}
\begin{bmatrix}
2 \\
2 \\
2 \\
2 \\
2 \\
2
\end{bmatrix}
\]

Assumptions concerning complexity matrix \(C_{1}^{\text{Bank\_application\_Network}}\): The “one-account” operations need to execute one \text{send\_msg} and one \text{rec\_msg} for the client process and ditto for the server process. It is assumed that the data to be transferred for the “one-account” operations fits into one message. The amount of network use of the two list operations is dependent on \(N\), \(R\) and \(M\). Many RPC calls may be necessary in order to transfer all the information from the file server to the client (Bank application).

\[
C_{1}^{\text{Bank\_application\_CPU\#2}} = \begin{bmatrix}
\text{create} & f(R) \\
\text{read} & f(R) \\
\text{update} & f(R) \\
\text{delete} & f(R) \\
\text{custnam} & f(N, R, M) \\
\text{custbal} & f(N, R, M)
\end{bmatrix}
\begin{bmatrix}
\text{instr}
\end{bmatrix}
\]

Assumptions concerning complexity matrix \(C_{1}^{\text{Bank\_application\_CPU\#2}}\): Only the file server runs in CPU\#2, hence this complexity matrix represents the resource demands of the file server. The create, read, update and delete operations are independent of \(N\) because they operate on only one account. It is assumed that the file server uses appropriate indexing of its data so that data can be localised with approximately fixed resource usage. It is also assumed that their CPU demand only depends on the volume of the data they have to process \((R)\). The two list operations are obviously dependent on \(N\). The CPU demand caused by sorting is mainly dependent on \(N\), the demand caused by reading of the account file and writing to the temporary file is mainly dependent on the data volume \((N \cdot R)\) and the demand caused by

\(^1\)If customers or accounts could be selected according to some criteria, another parameter would be necessary: the selectivity of the query
communication is mainly dependent on the data volume and the maximum number of records that can be transferred in one message \((N, R, \text{and } M)\).

\[
\begin{bmatrix}
\text{Create} & \text{DiskWrite} \\
\text{read} & f(R, D_w, b_s) \\
\text{update} & f(R, D_w, b_s) \\
\text{delete} & f(R, D_w, b_s) \\
\text{custnam} & f(N, R, D_r) \\
\text{custbal} & f(N, R, D_r)
\end{bmatrix}
\]

Assumptions concerning complexity matrix \(C_{Disk}^{Bank\_application}\): The create and read operations do random accesses which will not benefit from read-ahead. The custnam and custbal operations reads the accounts file sequentially and may therefore take advantage of the disk buffer for reads. Therefore their need for disk accesses is dependent on the data volume to be read and the size of the read-ahead buffer, \(D_r\). The update and delete operations do not need disk reads. All operations except read need to write on the disk. The number of physical disk accesses depends on how many records (of length \(R\)) can be stored in the disk buffer for writes (size \(D_w\)) before a physical disk access is necessary. If synchronous writes are used (specified by \(b_s\)) there will be no buffering.

Reusability of Performance Characterisations Since the complexity functions are parameterised with respect to \(N\), the number of accounts, the composite work model can be used for varying sizes of the account file. The other parameters are less likely to change. However, any change in any part of the software may invalidate the complexity functions because all the software components are represented by only one work model component.

Risk of Error Propagation There is no risk of error propagation in this formulation because there are just two layers. Error propagation becomes an issue when the composite work model consists of several layers.

Measurement Cost The number of operation-suboperation relationships is 33. The number of required measurement runs is 6 times the number of combinations of parameter values. In this degenerate case measurement of one transaction at a time is the same as measurement of one component at a time. It is not feasible to do the measurements under production workload because the parameters will probably remain the same as long as the system is running.
Formulation #2

![Diagram](image)

**Figure 12.2: A composite work model of the client-server example where the client and the server are modelled explicitly**

**Formulation of the Model** Since the client and server software can run in different computers, it is desirable to represent this fact explicitly in the model. Therefore the server part is represented by a work model component that is inferior to the Bank application component (see Figure 12.2). Informally, one could say that the Bank application component have now delegated the server work to a specific inferior component while the client part of the application is not delegated. The File server component provides persistent storage to the Bank application component. Therefore there is a memory relationship between these two components. Both of these components use the Network component in order to communicate with each other by means of remote procedure calls. Thus, communication relationships are needed in the composite work model. It is evident in the model that the client part is using only CPU#1 for processing and that the server part is using only CPU#2 for processing. The Disk provides persistent data storage to the file server. Thus, a memory relationship is required.

**Feasibility of Necessary Measurements** The measurement data must be captured in exactly the same way as for formulation#1 but in this case it is not necessary to aggregate the measurement data. The relationship between measurable components and work model components corresponds to case a) in Figure 11.5 if the client and the server run on different computers.

**Compatibility with Contention Model** The composite work model in Figure 12.2 is compatible with a contention model which takes contention for the File server into account (as described in Section 4.8).

**Complexity Functions and Matrices** The relationship between the Bank application component and the CPU#1 has not changed:
When it comes to the relationship between the Bank application component and the File server component, a problem arises. The "one-account" operations are ok. For instance, when the function update of the bank application component is invoked, fupdate is invoked once. However, for the operations fcustnam and fcustbal there are no such simple relationships. The first invocation reads all the account data, sorts the data according to some criterion, writes the sorted data to a temporary file, and returns information about the first account in the temporary file. The next invocations of the operation will return information about the next account in the temporary file. Hence, the first invocation behaves totally different from the following invocations. These operations are state-dependent. Therefore the fcustnam operation must be splitted. In the following fcustnam represents the behaviour during the first invocation. The fcustbal must also be splitted in the same way. Since fetching information about the next account from a temporary file is the same regardless of the sorting criterion, a new operation, fnext, is defined. This operation is common to fcustnam and fcustbal.

\[
C^2_{\text{Bank\_application}} = C^1_{\text{Bank\_application}}
\]

\[
C^2_{\text{CPU\#1}} = C^1_{\text{CPU\#1}}
\]

\[
C^2_{\text{File\_server}} = \begin{bmatrix}
\text{create} & \text{read} & \text{update} & \text{delete} & \text{fcustnam} & \text{fcustbal} & \text{fnext}
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & f(N, R, M) \\
0 & 0 & 0 & 0 & 0 & 1 & f(N, R, M)
\end{bmatrix}
\]

Assumptions concerning complexity matrix \(C^2_{\text{File\_server}}\): The number of invocations of fnext is dependent on the number of account descriptions \(N\) that must be transported to the client as well as on the number of account descriptions that can be packed in one network message (parameters \(R\) and \(M\)). If only one account record is returned per invocation of fnext, then \(f(N, R, M) = N - 1\).

\[
C^2_{\text{Disk}} = \begin{bmatrix}
\text{diskread} & \text{diskwrite}
\end{bmatrix}
\]

\[
\begin{bmatrix}
\text{fccreate} & \text{fread} & \text{fupdate} & \text{fdelete} & \text{fcustnam} & \text{fcustbal} & \text{fnext}
\end{bmatrix}
\]

\[
\begin{bmatrix}
k & k & 0 \\
0 & 0 & 0 \\
f(N, R, D_r) & f(N, R, D_w, b_s) \\
f(N, R, D_r) & f(N, R, D_w, b_s) \\
f(R, D_r) & 0
\end{bmatrix}
\]
Assumptions concerning complexity matrix $C_{\text{Bank_application}}^{2}$: The $f\text{create}$ and $f\text{read}$ do random reads by means of appropriate indexing, therefore it is assumed that they will not benefit from read-ahead (i.e., not depend on $D_r$). However, the $f\text{custnam}$, $f\text{custbal}$ and $f\text{next}$ operations do sequential reading and may therefore benefit from read-ahead. Hence the data volume to be read ($N \cdot R$ or $R$) and the size of the disk buffer for reads (read-ahead), $D_r$, determine the number of physical disk accesses. On the other hand, the number of required $\text{diskwrite}$ is dependent on the data volume to write ($N \cdot R$ or just $R$), whether write is synchronous, and the size of the disk buffer for writes (if writes are not synchronous).

$$
C_{\text{Bank_application}}^{2} = \begin{bmatrix}
\text{send}\_\text{msg} & \text{rec}\_\text{msg} \\
\text{create} & 1 & 1 \\
\text{read} & 1 & 1 \\
\text{update} & 1 & 1 \\
\text{delete} & 1 & 1 \\
\text{custnam} & f(N, R, M) & f(N, R, M) \\
\text{custbal} & f(N, R, M) & f(N, R, M)
\end{bmatrix}
$$

Assumptions concerning complexity matrix $C_{\text{Network}}^{2}$: Each “one-account” operation needs only one $\text{send}\_\text{msg}$ to send the query and one $\text{rec}\_\text{msg}$ to receive the result of the query. For $f\text{custnam}$ and $f\text{custbal}$ the number of messages is determined by the data volume ($N \cdot R$) and the maximum length of an RPC message ($M$).

$$
C_{\text{Network}}^{2} = \begin{bmatrix}
\text{send}\_\text{msg} & \text{rec}\_\text{msg} \\
\text{fcreate} & 1 & 1 \\
f\text{read} & 1 & 1 \\
\text{fupdate} & 1 & 1 \\
f\text{delete} & 1 & 1 \\
\text{fcustnam} & 1 & 1 \\
\text{fcustbal} & 1 & 1 \\
f\text{next} & 1 & 1
\end{bmatrix}
$$

Assumptions concerning complexity matrix $C_{\text{Network}}^{2}$: Each operation requires only one $\text{rec}\_\text{msg}$ to receive the query and one $\text{send}\_\text{msg}$ to return the result.
12.1. The Client-Server Example

\[ C^{File\_server}_{CPU\#2} = \begin{bmatrix} fcreate & instr \\ fread & f(R) \\ fupdate & f(R) \\ fdelete & f(R) \\ fcustnam & f(N, R) \\ fcustbal & f(N, R) \\ fnext & f(R) \end{bmatrix} \]

Assumptions concerning complexity matrix \( C^{File\_server}_{CPU\#2} \): It is assumed that the CPU demand of each of these operations is only dependent on the data volume that is processed.

**Reusability of Performance Characterisations** No standard software components are represented in the composite work model in Figure 12.2. That implies that it is unlikely that some of the work model components will be reused in a composite work model of another similar client-server system. However, changes in the implementation of either the client or the server will only have effect on the corresponding work model component if the interfaces remain unchanged.

**Risk of Error Propagation** There is no propagation of errors in the formulation in Figure 12.2. The reason is that all the complexity functions in \( C^{Bank\_application}_{File\_server} \) can be found with 100% certainty.

**Measurement Cost** The measurement cost are approximately the same as in the previous formulation (Figure 12.1). The reason is that many of the complexity function are just a constant which can be found with 100% certainty.

**Formulation #3**

**Formulation of the Model** In this formulation a third layer of software components is introduced. The file system part of the File server component in Figure 12.2 is extracted and modelled as a separate software component in the model in Figure 12.3. The File system component executes in CPU\#2 (a processing relationship in SP terms) because it is a part of the file server. Now the operations of the File server can be characterised with respect to File system operations (a memory relationship in SP terms) instead of operations on the Disk component. Hence, the performance characterisation of the File server component is less dependent
Figure 12.3: A composite work model of the client-server example
where also the file system is modelled explicitly

on disk operations. The File system component has a memory relationship to the Disk component.

Feasibility of Necessary Measurements The operations provided by the File system component are regarded as "system" calls in the UNIX operating system [19]. Invocations of system calls in UNIX can be captured by the "trace" command. In Unix it is difficult to measure the CPU demand of parts of the software which runs in a process. This corresponds to the case b) in Figure 11.5 where several work model components make up a measurable component. Therefore, instrumentation of the file server software is necessary in order to measure the CPU demands of the File system component and the File server component separately.

Compatibility with Contention Model If there can be contention for operations or data (locking) of the file system and that these contention effects are modelled explicitly in the contention model, then the formulation in Figure 12.3 is more compatible with the contention model than the formulation in Figure 12.2.

Complexity Functions and Matrices Four of the complexity matrices of the model in Figure 12.2 remain the same in the model in Figure 12.3:

\[
C_{3\text{CPU#1}}^{\text{Bank-application}} = C_{2\text{CPU#1}}^{\text{Bank-application}}
\]

\[
C_{3\text{Network}}^{\text{Bank-application}} = C_{2\text{Network}}^{\text{Bank-application}}
\]

\[
C_{3\text{File_server}}^{\text{Bank-application}} = C_{2\text{File_server}}^{\text{Bank-application}}
\]

\[
C_{3\text{Network}}^{\text{File_server}} = C_{2\text{Network}}^{\text{File_server}}
\]

The new component-subcomponent relationships can be characterised as follows:
12.1. The Client-Server Example

\[
C_{\text{File\_server}}^{\text{File\_system}} = \begin{bmatrix}
1 & 1 & 1 & 1 & 0 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 \\
2 & f(N) & f(N) & 2 & 1 \\
2 & f(N) & f(N) & 2 & 1 \\
0 & 0 & 1 & 0 & 0
\end{bmatrix}
\]

Assumptions concerning complexity matrix \(C_{\text{File\_server}}^{\text{File\_system}}\): It is assumed that all operations except \(\text{fnext}\) perform \(\text{open}\) and \(\text{close}\) for every call in order to ensure exclusive write. This is not necessary every time the \(\text{fnext}\) operation is invoked because it operates on a temporary files which is private for the transaction. The \(\text{fcustnam}\) and \(\text{fcustbal}\) operations must \(\text{open}\) and \(\text{close}\) two files: The accounts file that is shared among all the transactions and the temporary file which is private to the transaction.

\[
C_{\text{Disk}}^{\text{File\_system}} = \begin{bmatrix}
\text{open} & \text{diskread} & \text{diskwrite} \\
& f(b_c) & f(b_c) \\
& 0 & f(B_w, D_w, b_s) \\
\text{write} & f(B_r, D_r) & 0 \\
\text{read} & 0 & f(D_w, b_s) \\
\text{close} & 0 & f(N, R) \\
\text{unlink} & & \\
\end{bmatrix}
\]

Assumptions concerning complexity matrix \(C_{\text{Disk}}^{\text{File\_system}}\): \(\text{open}\) writes to disk only if a new file is created during the \(\text{open}\). The categorical parameter \(b_c\) indicates whether a create is needed. The number of \(\text{diskwrite}\) during a \(\text{write}\) is dependent on whether writing is synchronous (categorical parameter \(b_s\)) the size of the diskbuffer for writes, and the size of the buffer that is transferred. The number of invocations of \(\text{diskread}\) during a \(\text{read}\) depends on \(B_r\), the requested amount of text and the size of the diskbuffer for reading, \(D_r\). If the disk buffer have space for more than one account record, it is possible to do read-ahead. If the accounts are read sequentially, the number of diskreads can be reduced. When a \(\text{close}\) is invoked the data in the disk buffer for writes must be written to disk. The maximum amount of data in the disk buffer is \(D_w\). If writing is synchronous, the diskbuffer for writes will always be empty. The number of \(\text{diskwrite}\) during an \(\text{unlink}\) is dependent on how much data that are stored in the file to be deleted.
\[ C_{CPU}^{File\_server} = \begin{bmatrix}
  fcreate & k \\
  fread & k \\
  fupdate & k \\
  fdelete & k \\
  fcustnam & f(N, R) \\
  fcustbal & f(N, R) \\
  fnext & k 
\end{bmatrix} \]

Assumptions concerning complexity matrix \( C_{CPU}^{File\_server} \): The \( fcreate, fread, fupdate, fdelete \) and \( fnext \) operations process information about only one account. Their CPU demand is assumed to be fixed. The other operations, \( fcustnam \) and \( fcustbal \), process information about several accounts. Their CPU demands are dependent on the data volume \( (N \cdot R) \) to be read from the accounts file and the data volume to be written to a temporary file, and dependent on \( N \) because of the sorting.

\[ C_{CPU}^{File\_system} = \begin{bmatrix}
  open & k \\
  write & f(B_w) \\
  read & f(B_r) \\
  close & k \\
  unlink & f(N, R) 
\end{bmatrix} \]

Assumptions concerning complexity matrix \( C_{CPU}^{File\_system} \): The operations \( open \) and \( close \) are assumed to demand a fixed amount of CPU. The CPU demand of \( write \) depends on the size of the write buffer, \( B_w \), while the CPU demand of \( read \) depends on the size of the read buffer, \( B_r \). The CPU demand of \( unlink \) depends on the size of the file \( (N \cdot R) \) to be deleted.

**Reusability of Performance Characterisations**  In this formulation a standard component is modelled explicitly; the \( File \_system \) component. It is very likely that this work model component can be reused.

**Risk of Error Propagation**  To assess the risk of error propagation it is necessary to find the operation-suboperation paths in the composite work model which result in the largest devolved work on a primitive component. These paths are then investigated to see if they contain some trivial\(^2\) complexity functions. If so, the probability of significant error propagation is low. Non-trivial complexity functions

---

\(^2\) Trivial complexity functions, e.g., 1, are marked with a \[\text{frame}\] in the complexity matrices.
are required for aggregated operations and for complexity functions that relate a software component to a hardware component.

Formulation #4

![Diagram of a composite work model of the client-server example where also the remote procedure call library is modelled explicitly](image)

**Figure 12.4:** A composite work model of the client-server example where also the remote procedure call library is modelled explicitly

**Formulation of the Model** The remote procedure call library is modelled explicitly in the formulation in Figure 12.4. The implementations of the RPC library components may be different if, for instance, the CPU#1 and CPU#2 are of different types. The RPC library components have a processing relationship with CPU#1 and CPU#2 respectively. The network itself is regarded as a low capacity store, hence the memory relationships.

Formulation #5

**Formulation of the Model** Here the communication library provided by the operating system (UNIX in this case) is characterised separately from the characterisation of the RPC library. The reasons are that the implementation of the RPC library is independent of the communication library of the operating system and that several implementations of the communication library are available. The same communication library, e.g., the socket library, may be implemented differently on different types of computer and different communication libraries, e.g., the socket and stream libraries, may be available on the same type of computer.

**Feasibility of Necessary Measurements** The operations provided by the UNIX socket library are also regarded as “system” calls in UNIX. Therefore, they can be traced by the “trace” command.
Figure 12.5: A composite work model of the client-server example where also the UNIX communication library (e.g., socket) is modelled explicitly

Other Formulations

The five preceding formulations of composite work models for the client–server example show the result of recursive application of the SP method when both the application-dependent components, Bank application and File server, are considered to be distributed. During the course of the case studies, composite work models were built where only the File server was considered as distributed. In Figure 12.6 the distributed File server devolves work onto the same RPC component for both the client and the server side. This does not imply that both the client and the server runs the same implementation of the remote procedure call library. It is perfectly possible to specify several alternative implementations of a work model component. In addition, the RPC library and the Operating system comms components are considered as distributed because they demand processing in both the CPUs. Note the communication relationships which are required by the SP rules when the processing and/or memory of a component is distributed. The composite work model in Figure 12.7 is similar to the previous model, but it represents the communication software separately for the client side and the server side.
12.1. The Client-Server Example

Figure 12.6: The same component is used for representing the communication between the client and the server

Figure 12.7: Different RPC components are represented for the client-side and the server-side
12.2 International Computers Limited

The ICL example system, DAIS, is described briefly in Section 3.2. Considerable time was spent on identifying appropriate software components and identifying the operations on these components. The dependencies between the components turned out to be quite strong as will be demonstrated in the following. Note that the software for transaction management is not included in the composite work models. The reason is that the DAIS prototype did not support transaction management at the time it was examined. The actual number of CPUs and number of disks that are modelled in these examples are reduced in order to simplify the examples. All the disks do not necessarily have to be included in the model. Some disks may contain the source code of the system, others may be used for development and debugging only, etc.

Formulation #1

![Diagram]

Figure 12.8: A minimal composite work model of the ICL system

**Formulation of the Model** This formulation regards DAIS as a monolithic piece of software which uses a local area network for communication between the four CPUs in the system. Therefore the DAIS component has a communication relationship with the LAN component (in SP terms) and each CPU has a processing relationship with the DAIS component. Two disks are represented in the model. The disk#1 and disk#2 components have a memory relationship with the DAIS component.

**Feasibility of Necessary Measurements** Since the software consists of a lot of processes which are spread among the computers in the system, it is not possible to measure the software as a monolithic piece, i.e., the entire DAIS is not a measurable component.

**Compatibility with Contention Model** The formulation in Figure 12.8 is compatible with a contention model which only models contention for hardware devices.
Complexity Functions and Matrices  The operations that are provided to the users (Appendix B) fall into three groups:

1. Fetch information about one customer according to a unique identification of the account, i.e., the account number.

2. Fetch list of customers according to some criterion, e.g., the customers whose last name begins with a “V”.

3. Perform some operation on one specific account.

Likely parameters of the complexity functions are:

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Description</th>
<th>Applies to</th>
</tr>
</thead>
<tbody>
<tr>
<td>(N)</td>
<td>The number of accounts</td>
<td>1 2</td>
</tr>
<tr>
<td>(I)</td>
<td>Number of levels in the index</td>
<td>1</td>
</tr>
<tr>
<td>(S)</td>
<td>Query selectivity</td>
<td>2</td>
</tr>
<tr>
<td>(R)</td>
<td>Record length of account information</td>
<td>1 2 3</td>
</tr>
</tbody>
</table>

Note that the software components of the database management system are hidden within the DAIS software component in this composite work model. That means that it is not necessary to characterise the performance of other queries than those (typically few) called from the user operations. However, the query optimisation strategy of the database management systems may change suddenly when some threshold is exceeded. Such a change in the optimisation may invalidate all the performance characteristics that have been obtained.

Reusability of Performance Characterisations  The composite work model formulation in Figure 12.8 yields no reuse benefits. It provides no support for reuse of measurements when the configuration of the system is changed.

Formulation #2

Formulation of the Model  The formulation in Figure 12.9 reveals the distributed database management system. The Database server component has been extracted from the DAIS component. The remains of DAIS are assumed to run exclusively (as one process) in CPU#1 while the Database server component is distributed over CPU#2, CPU#3 and CPU#4. The DAIS component uses the Database server for persistent storage, hence the memory relationship.
Feasibility of Necessary Measurements  If it is assumed that the DAIS component runs in one process, it can be measured without any need for aggregation of measurement data. However, the Database server still consists of many measurable components (as in Figure 11.5c).

Compatibility with Contention Model  The Database server may be a critical component in DAIS because it ramifies the queries to the database management systems which constitute the distributed database. The Database server may be replicated or multi-threaded or both, but since all requests to the databases must go through this/these component(s) there is a risk of saturation. The formulation in Figure 12.9 is compatible with a contention model which takes contention for the Database server into account.

Complexity Functions and Matrices  The functionality of the database server component (see also Appendix B) is represented by only three operations:

- **LIS_CONNECT** – Prepare for query
- **LIS_SQL** – Execute an SQL statement which is given as a textual argument to this operation.
- **LIS_FETCH** – Fetch the next unread part of the result of a previous SQL call.

In particular the resource demands of the LIS_SQL operation varies greatly depending on the query to be executed. Hence, this operation must be split (Section 11.1). One LIS_SQL operation must be defined for each typical SQL query. In order to reduce the number of operations that are defined at this stage, clustering may be beneficial. First, different kinds of query are grouped according to functionality and then numerical cluster analysis may be applied to find representative operations for each kind of query.

---

3The startup and disconnect operations are used in order to start and stop the database server process. These operations are not called from a transaction.
Reusability of Performance Characterisations  The software of the Database server is independent of the applications, but the performance characteristics of this component is dependent on the selection of queries that are characterised by splitting and possibly by clustering. In addition, the Database server is dependent on the actual configuration of the distributed database. This configuration is not likely to be reused.

Formulation #3

![Diagram of ICL system components]

Figure 12.10: A composite work model of the ICL system where the application, the database server, and the local database management systems are modelled explicitly

Formulation of the Model  In the formulation in Figure 12.10, two different database management systems, Ingres DBMS and Oracle DBMS, are extracted from the Database server component. The two new components each have a memory relationship to the Database server component. Each of the new components have a communication relationship with the LAN component, a processing relationship with one CPU, and a memory relationship with one Disk. Note that each of the non-primitive components runs in only one CPU. That means that this formulation depicts the actual distribution of processing in the system.

Feasibility of Necessary Measurements  Now measurement of one work model component is confined to measurement on one computer. If UNIX is assumed and each work model component runs in one process, the work model components are measurable components in the real system.

Compatibility with Contention Model  Now two more software components for which there can be contention, are exposed.
Complexity Functions and Matrices There are two main operations (Appendix B) on each DBMS component: Operation and Fetch. The former operation are similar to the LIS_SQL operation of the Database server component while the latter operation is similar to the LIS_FETCH operation. Hence, the problems concerning performance characterisation of the Database server manifests themselves for the DBMS components as well. Splitting of the operations and possibly clustering are necessary.

Reusability of Performance Characterisations The components of the database software are application independent, but this is not the case for the performance specifications of these components. The reason is that the splitting of operations possibly followed by clustering are application dependent. Reuse is difficult. There is a possibility that the splitting of the operations may be reusable within some narrow application domain if the splitting is done carefully.

Possible Extension #1

Figure 12.11: The communication software for each capsule in the system is modelled explicitly in the composite work model. In a) only one layer is introduced whereas in b) two layers are introduced.

Formulation of the Model Each non-primitive work model component in Figure 12.10 may have a subcomponent which represents the communication subsytem which enable the distribution. The communication link from each of the non-primitive components can be intercepted by a Comms component as shown in Figure 12.11a.
Feasibility of Necessary Measurements  DAIS is based on the ANSA architecture which provides a well-defined interface to the communication subsystem. The corresponding communication library is linked to the application programs in each UNIX process (other operating systems are possible). The communication library provides a transparent way for inter-process communication regardless of which computer the communicating processes run on. A well-defined interface to the communication software also implies well-defined measurement points.

Reusability of Performance Characterisations  The interface to the communication library is the same in all ANSA systems (by definition). The implementation of the communication library may be different in different systems. Since the number of implementations is limited, it is likely that the performance characteristics of the communication component can be reused.

Possible Extension #2

Formulation of the Model  In Figure 12.11b the Comms component is split into two components: REX (Remote execution protocol) and MPS (Message passing system). The REX have a communication relationship to its superior component while it uses the MPS component as memory and some CPU for processing.

Feasibility of Necessary Measurements  Both the REX and MPS have interfaces which are defined by the ANSA software architecture. Thus, well-defined measurement points exists and measurement is feasible. However, by code inspection it was found that functions were often used as arguments in calls within the REX and MPS software components. Consequently, the interfaces are blurred and a thorough examination is necessary in order to determine the measurement points.

Reusability of Performance Characterisations  Since the REX and MPS components have well-defined interfaces and since these components are reused in all ANSA systems, reuse of performance characterisation of such work model components is feasible. Since both components can have different implementations, a library of performance characterisations for the alternative implementations must be established.

Possible Extension #3

Formulation of the Model  For several third generation programming languages there are standards (e.g., C, Fortran, Cobol) which make it possible to run a pro-
Figure 12.12: The processing of each capsule in the system may be characterised in terms of high-level program statements

gram on a different hardware platform by just recompiling the code. Recompilation is necessary because the mapping from high-level language statements to machine instructions is system-dependent. Consequently, each non-primitive work model component in Figure 12.10 may have a subcomponent which represents the mapping from source code written in a third generation language such as "C", to machine-dependent machine code. So the processing link from each of the non-primitive components to a CPU can be intercepted by a High-level language component as shown in Figure 12.12 (see also the presentation of system independent workload models on page 73).

Other Formulations

The final SP model from the case study at ICL is depicted in Figure 12.13. This model is drawn by means of the SP tool. Dashed lines represent communication relationships, thin lines represent processing relationships, and thick lines represent memory. The SP tool allows lines in the model to be broken in order to avoid too many crossing lines. On paper it is difficult to see which line segments that belong to the same line. However, in the SP tool both ends will be high-lighted when the pointer is close to one of the ends.

The components client, dbserver, ingdb and oracdb corresponds to the non-primitive components in Figure 12.10. The ingdb components provides a standard interface for the ingdbms component and the oracdb component provides a standard interface to the oracdbms component. In addition two other main components are introduced: the lisadmin and the trader components\(^4\). These components are explained in Section 3.2. Each main component have an obj_code subcomponent which maps the source code (e.g., written in C) into machine-dependent instructions. This is to ease the work modelling when software components are moved to different hardware platforms. Each main component is associated with a "local"

\(^4\)Some text close to the broken lines indicates where the line ends.
Figure 12.13: The "final" SP model at ICL
LAN component, a "local" CPU component and a "local" Memory component. Some main components also have a "local" Disk component. The work devolved onto these local components enables calculation of the devolved work in terms of each main component in the system. In turn, these local components are mapped onto (or related to) the corresponding nodes which represents the real hardware devices. Each computer is represented by its CPU, memory, disk, and network connection. In this example, work is only devolved onto two computers, pluto and noah, which correspond to the configuration used during the experimentation. It would be very easy to map the work onto several other hardware platforms.
12.3 The Hospital Case

The Hospital system is described briefly in Section 3.3. The formulation of composite work models of the Hospital system was difficult because the operations provided by each software component were unknown. Note that the composite work models of the hospital system (and Tandem systems in general) appear to be messy. The reason is that most work model components represent several processes which may be allocated to any of the CPUs in the system. Each process can access any of the disks in the system.

Formulation #1

![Diagram of Hospital System](image)

Figure 12.14: A minimal composite work model of the hospital system

Formulation of the Model The model that is shown in Figure 12.14 is the basic formulation of a composite work model of the Tandem system. Only one layer of software components in addition to the hardware devices are represented in the model. For convenience, only two of the CPUs and two of the disks are represented in the model. The inter-processor bus which enables processes which run in different CPUs to communicate, and the terminal communication lines are represented as primitive components in the model. The terminal lines enables communication with the user; therefore there is a communication relationship (in SP terms) between the application components and the terminal line component. The inter-processor bus enables distributed processing; therefore the application components have a communication relationship to the inter-processor bus component. The CPUs provide processing to the applications while the disks provide the means for persistent storage of data for the applications. Thus there exist processing and memory relationships, respectively. Note the inter-processor bus. Tandem claims that it has a very high capacity so it will never become a bottleneck in the system. One might argue that this device should not be part of the model then. It is included in the model because the SP rules (Chapter 6) require that the means of distributed processing is included in the model.

Feasibility of Necessary Measurements There are two approaches to measurement according to the model in Figure 12.14. The first approach is to run
transactions one-by-one on an otherwise idle Tandem computer. Hence, measurement data can be collected in terms of hardware devices. The other approach is to measure hardware usage in terms of processes that are involved in the execution of transactions. Unfortunately, transaction instances cannot be distinguished during measurement. Therefore the transactions have to be run one-by-one. In order to obtain the complete resource demands of a transaction, also overhead must be taken into account. It is a major problem that the resource demands of overhead does not necessarily increase linearly with the number of concurrent transactions. Consequently, the overhead is difficult to predict when transactions are run one-by-one. Tandem systems can also run batch jobs. Each instance of a batch job run in a process of its own. They are initiated from a terminal and put into a batch queue where they will remain inactive until their scheduled start time.

Compatibility with Contention Model The model in Figure 12.14 is compatible with any contention model which only models competition for hardware devices. Since applications in PATHWAY systems are made up of requester-server relationships, it is possible that a software server can become the bottleneck of the system. Tandem tries to avoid queueing for software processes by replicating software servers for which there are contention. To determine whether this is a problem, it is possible to measure the time that messages to a process spend waiting in the process's message input queue (i.e., a queue counter provided by the MEASURE tool as described in Section 8.1). The "RECV-QTIME" in the PROCESS entities collected by MEASURE provide this information. During one week in October 1990, 12338 different processes were measured. The process with the longest average queue of messages/requests was the "PATHMON" process which monitors PATHWAY. Its average queue length was 4.2. The distribution of average queue lengths was as follows:

<table>
<thead>
<tr>
<th>Range of average queue lengths</th>
<th>number of processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q &gt; 1$</td>
<td>5</td>
</tr>
<tr>
<td>$1 \geq q &gt; 0.1$</td>
<td>1997</td>
</tr>
<tr>
<td>$0.1 \geq q &gt; 0.01$</td>
<td>2019</td>
</tr>
<tr>
<td>$0.01 \geq q &gt; 0.001$</td>
<td>2965</td>
</tr>
<tr>
<td>$0.001 \geq q$</td>
<td>5352</td>
</tr>
</tbody>
</table>

Note that some of the averages represent up to 7 days of execution. Hence, any perturbations are lost. The processes which offer sorting services typically have a (short) queue. The other processes with queues are almost exclusively administration processes and processes which provide a command interface to the operating system for the system manager. It seems that processes that are involved in transactions and batch jobs do not have significant queues.

Reusability of Performance Characterisations The main problem concerning the model formulation in Figure 12.14 is the lack of reusability. Changes within one
application may invalidate the performance characterisations of the entire application. The reason is that components of the application are not modelled explicitly. For instance, if one process is moved from one CPU to another it is difficult to predict the demand for the affected CPUs after the move. If the process was shared by several application, also the performance specification of these applications are invalidated. To achieve reusable performance specifications it is necessary to limit the consequences of changes of the software and of changes in the mapping of processes to CPUs. In addition, it is necessary to model explicitly the software components that are shared between different applications. Only then it is possible to know when changes in one application influence other applications.

Formulation #2

Figure 12.15: A composite work model of the hospital system where the most significant components have been identified and modelled explicitly

Formulation of the Model  To remedy the problem concerning the reusability of the model in Figure 12.14, the composite work model shown in Figure 12.15 is proposed. This composite work model represents terminal handling, terminal communication and the disk process explicitly. Each of these three components represents a collection of processes which provides the same services, i.e., they are replications which may be distributed in any manner between the CPUs.

It is difficult to decide which components of the Tandem software that should be represented explicitly in the composite work model. The approach in identifying important components was as follows. A measurement run in February 1990 gave the following distribution of CPU usage over different classes of processes.
Communication 6%
Terminal Control Program (TCP) 6%
Patient Administration Application
   - Online 2%
   - Batch 24%
Laboratory system (just testing) 2%
Disk processing 39%
System administration 21%

The overall utilisation of the 5 CPUs in the system was 16%. All software components shown in the distribution of CPU usage, except system administration and the system being tested, are part of the model in Figure 12.15. The system administration software is difficult to model because its resource usage may be based on ad-hoc activities that are not related directly to execution of the application. Another question is whether the PATHMON process which had this relatively long queue, should be represented explicitly in the model. It was assumed that the service times in PATHMON were so short that the queuing time was negligible. In summary, it is difficult and time-consuming to obtain a complete composite work model because of all the processes that must be examined.

**Feasibility of Necessary Measurements** It is not possible to measure the TCP, communication, and diskhandler processes with respect to a transaction. Neither it is possible to measure how the functionality of these components are used by other software components in the system.

**Complexity Functions and Matrices** A major problem in the hospital case study was that measurement data could only be captured in terms of processes, not in terms of operations provided by the processes. Therefore it was assumed that each online application process offers a set of related operations with approximately similar resource demands. In fact, it was assumed that each such process only has one operation. Its resource demand was found by measuring the total resource demands of the process and then dividing these demands by the number of incoming messages (= number of requests, by assumption) that had been serviced during the measurement session. On the other hand, each batch job runs in a process of its own and the name of the process indicates which batch job the process is running.

Even after these severe simplifications there were still too many operations to be represented as operations of a work model component. Totally 180 different jobs were detected by analysis of the measurement data. It is unrealistic to model 180 operations on application components in a composite work model. Therefore the number of operations had to be reduced. This was achieved by means of the WAT tool (Section 8.2). The operation characterisations were grouped according to application subsystem (PAS, NSK, or NSM) and according to job type (batch or online). Thus 6 groups of operation characterisations were obtained. The distribution over the six groups of application processes was as follows:
12.3. The Hospital Case

<table>
<thead>
<tr>
<th>Service</th>
<th>Code</th>
<th>CPU Time</th>
<th>Memory Use</th>
<th>Disk I/O</th>
<th>Network I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>Patient administration</td>
<td>61</td>
<td>766.484</td>
<td>502.201</td>
<td>1263.242</td>
<td>0.000</td>
</tr>
<tr>
<td>Patient administration</td>
<td>23</td>
<td>3380.000</td>
<td>711.358</td>
<td>4203.000</td>
<td>20.000</td>
</tr>
<tr>
<td>Clinic laboratory</td>
<td>43</td>
<td>24394.880</td>
<td>274.793</td>
<td>87879.160</td>
<td>81.120</td>
</tr>
<tr>
<td>Clinic laboratory</td>
<td>14</td>
<td>4562.000</td>
<td>196.528</td>
<td>9428.000</td>
<td>0.000</td>
</tr>
<tr>
<td>Micro-biology</td>
<td>36</td>
<td>46245.000</td>
<td>760.484</td>
<td>93250.000</td>
<td>0.000</td>
</tr>
<tr>
<td>Micro-biology</td>
<td>3</td>
<td>150181.500</td>
<td>175.140</td>
<td>162825.000</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>307976.000</td>
<td>462.755</td>
<td>358179.500</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>155963.500</td>
<td>186.812</td>
<td>169726.500</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>153102.500</td>
<td>179.674</td>
<td>167990.000</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>245293.500</td>
<td>257.796</td>
<td>453643.000</td>
<td>0.000</td>
</tr>
</tbody>
</table>

Table 12.1: Example of inputs to WAT. The columns in the list represent: 1) numerical identification of operation, 2) CPU consumption in milliseconds, 3) number of bytes sent to terminal control processes (0 for batch jobs), 4) number of messages sent to some disk process, 5) average block-size for communication with disk processes, 6) number of block reads from disk, and 7) number of block writes to disk.

The results of the 6 WAT runs are shown in Table 12.2. The table\(^5\) shows the calculated resource demands of the 12 representative application operations. Rows 1 to 4 constitute the complexity matrix of the NSK application work model component. Rows 5 to 8 represent the NSM application while rows 9 to 12 represent the PAS applications.

The characterisation of application operations did not take into account any data-dependencies. Therefore the reusability of these measurement results are very limited. The appropriateness of the clusters was not examined either. But the case serves as an example of the need for aggregation of operations (Section 11.1) on work model components.

\(^5\)The measurement data show that \texttt{nsmbatch1} and \texttt{nsmbatch2} do not use the disk at all. This is probably not correct.
Table 12.2: Average resource demands of the application operations obtained by cluster analysis

<table>
<thead>
<tr>
<th>Operation</th>
<th>cpu</th>
<th>tcp bytes</th>
<th>disk msg</th>
<th>read disk block</th>
<th>write disk block</th>
</tr>
</thead>
<tbody>
<tr>
<td>nskonline1</td>
<td>343.098</td>
<td>1253.669</td>
<td>27.247</td>
<td>1.849</td>
<td>0.058</td>
</tr>
<tr>
<td>nskonline2</td>
<td>2013.087</td>
<td>2611.065</td>
<td>85.101</td>
<td>24.960</td>
<td>2.059</td>
</tr>
<tr>
<td>nskbatch1</td>
<td>242702.781</td>
<td>0.000</td>
<td>6441.210</td>
<td>793.481</td>
<td>0.001</td>
</tr>
<tr>
<td>nskbatch2</td>
<td>706840.062</td>
<td>0.000</td>
<td>53859.586</td>
<td>26420.414</td>
<td>3287.042</td>
</tr>
<tr>
<td>nsmonline1</td>
<td>158.037</td>
<td>3175.948</td>
<td>21.437</td>
<td>3.588</td>
<td>0.265</td>
</tr>
<tr>
<td>nsmonline2</td>
<td>1423.000</td>
<td>5001.241</td>
<td>180.439</td>
<td>79.838</td>
<td>24.077</td>
</tr>
<tr>
<td>nsmbatch1</td>
<td>6280.667</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>nsmbatch2</td>
<td>58253.039</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>pasonline1</td>
<td>53.153</td>
<td>735.601</td>
<td>15.043</td>
<td>4.865</td>
<td>0.115</td>
</tr>
<tr>
<td>pasonline2</td>
<td>2040.718</td>
<td>695.328</td>
<td>636.333</td>
<td>61.561</td>
<td>2.453</td>
</tr>
<tr>
<td>pasbatch1</td>
<td>193601.094</td>
<td>0.000</td>
<td>125974.773</td>
<td>152433.828</td>
<td>4.596</td>
</tr>
<tr>
<td>pasbatch2</td>
<td>449136.594</td>
<td>0.000</td>
<td>251136.031</td>
<td>415363.000</td>
<td>5061.000</td>
</tr>
</tbody>
</table>

Reusability of Performance Characterisations  As explained earlier, the TCP, Terminal communication and Diskhandler components have significant hardware resource demands. Therefore it would be beneficial if performance characterisations of these components could be reused in several studies. The probability of reuse of these components is high because they appear in every PATHWAY system.

Other Formulations

As can be seen clearly in the composite work models in Figures 12.14 and 12.15, the unrestricted distribution of processes among CPUs leads to "messy" models. From the fault-tolerance and workload-balancing points of view such distribution of processes is necessary. To obtain "cleaner" models and to make the work model components less dependent on the actual distribution of processes, artificial work model components have been introduced in the model as "distributors" according to some statistical distribution (Figure 12.16). However, this model is not a legal SP model because the Disk distr. component neither have a processing subcomponent nor a communication subcomponent. This can be remedied because the disk can have a disk controller which is capable of doing the processing that is required during disk accesses. Also the CPU distr. component is illegal according to the SP rules. It lacks one or more memory components. It is possible to assume that the physical memory of the CPUs is the memory that is needed to satisfy the SP rules.

The SP model that was used during the IMSE project [21] is shown in Figure 12.17 where the distr work model component represents an aggregated CPU which distributes the computation by means of the bus. The disk component represents all the disks in the system. The figure shows how the SP model appears on the screen.
when it is used in the SP tool.

The model that is depicted in Figure 12.18 takes the model in Figure 12.17 one step further. The CPUs and the disks in the system are represented by one primitive work model component for all the CPUs and ditto for the disks. With this model, the distribution of the devolved work on the CPU component and the Disk component must be done "by hand".

Figure 12.16: Composite work model of the hospital system where artificial components are included in the model in order to tidy up. Note that this model does not obey the SP rules.
Figure 12.17: Composite work model of the hospital system from the IMSE project

Figure 12.18: Composite work model of the hospital system where the distribution is hidden within the CPU and the Disk components.
Chapter 13

Measurement of Computational Work

This chapter presents a discussion of measurement according to a composite work model. The required measurement data cannot be captured by most state-of-the-art measurement tools. Instrumentation of the software components is therefore necessary. The process of capturing the raw measurement data and the derivation of complexity functions are very demanding tasks both in terms of time and cost. This is the reason why a thorough procedure for formulation and assessment of composite work models was developed (Chapter 11).

When a composite work model has been formulated, the extensive measurement experiments can be carried out according to the composite work model. The costly measurement experiments are therefore undertaken only after careful planning, that is, measurement capture is driven by overall considerations, not only by considerations for each component. This aspect makes the composite work modelling approach somewhat different from traditional sizing studies (Chapter 9).

A measurement tool which minimises the need for human intervention is needed in order to reduce the cost of measurement capture (see Section 11.7). Preferably all measurement data should be collected when the production workload is run. However, it is not likely that all parameter combinations for all components occur in the production workload during the measurement experiments. Therefore it is difficult to obtain complete performance characterisations based on the production workload.

The resource demands on the hardware devices can be measured by most state-of-the-art measurement tools although the repertoire of measurable components may be limited. The hardware measurement data are very sensitive to interference from measurement tools which capture software events. Measurement of software component interactions requires extensive instrumentation of the software so that the invoking software component and the software component which provides the service are identified for each invocation. Also the parameter settings and other dependen-
cies between the two components must be captured. The good news is that this kind of measurement is insensitive to interference from measurement tools although it may slow down the system\footnote{This is not true for real-time systems where the timing of the software component interactions may influence the execution of the software.}.

![Diagram of hardware and software interactions](image)

**Figure 13.1: A software component running on hardware platform A invokes a software component running on hardware platform B**

The interactions between software components determine the total hardware usage of the components. In Figure 13.1, the hardware resource usage of the software component running on hardware platform B depends on how many times it is invoked by the software component which runs on hardware platform A. Information about the software interactions is necessary in order to find out which software components that are using the hardware. Thus, two measurement steps must be performed either in sequence or simultaneously:

1. Measure interactions between software components
2. Measure hardware resource demands for each software component for each kind of invocation

Preferably, these steps should be performed simultaneously. The problem concerning measurement of software interactions is that instrumentation is needed which may interfere significantly on the hardware resource demands. The state of the art of distributed (hybrid) measurement tools (e.g., [48, 70]), is that the interference caused by software instrumentation can be reduced considerably when descriptions of the software events are written onto a logging device in hardware. Each hardware platform in the distributed system has such a logging device and these logging devices are connected by an additional network which is only used for transportation of measurement data. If such a measurement tool is not available, it is proposed that measurement of software interactions is performed before measurement of hardware resource demands to obtain a “map” of what is going on in the system. Then hardware resource measurement experiments are performed systematically, guided by the “map”, either software component by software component, or transaction by transaction. This sequencing of measurement experiments requires that the software interactions are reproducible.
13.1 Necessary Measurement Data

In the context of composite work models, the purpose of measurement is to determine the values of coefficients in the complexity functions. Measurement according to a composite work model requires that each work model component must be a measurable component (as explained in Section 11.2). Every time an operation of a component is invoked by one of the operations of one of its superior components, the context (parameter settings) of the invocation and the identifications of the involved operation and suboperation must be recorded.

Ideally, the following information should be available for each invocation of an operation:

- id of operation and its component
- parameter settings when the operation was invoked
- transaction id
- start time of execution of operation
- finish time of execution of operation
- suboperations which are invoked during the execution of the operation (one record for each combination of suboperation and parameter setting)
  - id of invoked suboperation and its component
  - parameter settings
  - number of invocations with identical parameter settings
- CPU usage (instruction count or CPU time) during execution of the operation
- Number of disk accesses during execution of the operation
- Number of messages sent to/received from the network during execution of the operation
- Memory demand during execution of the operation
- Buffer/cache statistics during execution of the operation

The viewpoint of the measurement data is from within each software process because it is necessary to capture the relationship between incoming invocations and outgoing invocations. It is not sufficient to regard the invocations from between processes. This is depicted in Figure 13.2.

This information about invocations of operations correspond the level of detail of Figure 11.6f.
Figure 13.2: Different viewpoints during measurement (note the two "eyes" in the drawing). In a) the viewpoint is from within a process whereas in b) the viewpoint is between processes.

Instrumentation of each software component is necessary to capture the operation argument values and other data dependencies which influence the execution of the operation. This instrumentation is preferably done during design since the dependencies should be well-known at design time. In addition, compilers may provide some instrumentation support. It must be easy to activate and deactivate the instrumentation because it will only be used during measurement of software component interactions.

Figure 13.3: Transaction execution paths

Tracing of transactions can be achieved if all invocations of software component operations carry a unique identification of the transaction instance which causes the invocation. Transaction tracing is illustrated in Figure 13.3. If the instrumentation cannot capture the id of the transaction that is requesting an invocation, it is not possible to relate execution of processes to transactions which constitute the work-
load model components. Without instrumentation at all, the available information is limited to which processes that interact. This is illustrated in Figure 13.4.

Collection of start time and finish time of an operation has at least three benefits. The difference between the finish time and the start time yields the response time for the execution of the operation. Concurrency effects can be investigated since it is possible to find which transactions that were executing concurrently at a certain point in time, i.e., the transaction mix. Finally, it is possible to reconstruct the sequence of invocations of operations of a transaction.

13.2 Three Modes of Measurement

There are three main modes of measurement: Measurement of production workload, measurement of individual transactions, and measurement of individual components. These modes correspond to the three categories of measurement availability that were presented in Section 11.7. The two first modes correspond to indirect measurement (Section 7.4) while the last mode corresponds to direct measurement. Each mode of measurement is assessed according to the following criteria:

- Need for instrumentation
- Availability of information
- Required number of measurement runs
- Completeness of measurement data
- Expertise required in measurement experiments
Measurement of Production Workload

Measurement data must be captured in terms of transactions, software components, and hardware devices when a distributed computer system are measured under production workload.

Need for instrumentation In order to distinguish transactions during measurement, a unique transaction id must be part of the message that is sent when an operation is invoked. Instrumentation is necessary in order to collect information about software interactions. Instrumentation is also necessary to capture hardware usage if the components in the composite work model does not correspond to measurable software components. The required instrumentation may require human intervention in the software.

Availability of information All information listed in Section 13.1 can be made available if the interference caused by the instrumentation is not prohibitive. In addition, since several transactions are running concurrently in the production workload, it is possible to measure the effects of concurrency. This information is necessary in order to obtain load-dependent complexity functions (if such functions are required).

Required number of measurement runs The main advantage of identifying each instance of a transaction is that the production workload can be characterised with respect to software component interactions without human intervention. This approach places more burden on the data analysis phase where the characteristics of each transaction type are extracted. However, this extraction can be supported extensively by analysis tools.

Completeness of measurement data Measurement data from production workloads can normally not be used as the basis for complete specifications of performance properties of software components. There is no guarantee that all combinations of parameter settings occurred in the production workload. In particular, the parameters which describe the system configuration will not change when the system is run in production.

Expertise required in measurement experiments No expertise is required during measurement experiments.
Measurement of Individual Transactions

Another measurement approach is to run the transactions one-by-one and measure the hardware resource usage in terms of all software components during the execution of each transaction. It is possible to measure transactions one at a time if only contention-independent measures are needed to characterise the computational work demand (Section 7.2).

Need for instrumentation There is no need for a unique transaction id because only one transaction is active at any time. instrumentation is still necessary in order to capture parameter settings. If the computer system is idle except for the transaction, the hardware usage can be measured for each of the hardware devices. If the same hardware devices are used by several different software components, instrumentation may be necessary in order to find the hardware device usage in terms of software components.

Availability of information All the information required in Section 13.1 can be obtained if suitable instrumentation is in place. Load dependencies are (by definition) difficult to capture.

Required number of measurement runs The number of measurement runs is dependent on the number of different transactions. Several measurement runs may be necessary for each kind of transaction if the transactions behaves differently when different input data are given.

Completeness of measurement data This mode of measurement can normally not provide sufficient measurement data for complete performance specification of the performance properties of software components. The reason is that each transaction can be applied systematically to the system while a software component cannot be systematically invoked by execution of transactions.

Expertise required in measurement experiments No technical knowledge of the computer system is necessary, but some knowledge about the applications is necessary in order to run the transactions one at a time.

Measurement of Individual Components

This is the “brute-force” measurement mode. Each component is placed on the “testbench” for detailed measurement. The hardware resource usage for each operation for each combination of parameter settings must be measured. A problem
which arises when components are measured one-by-one is that invocations and responses from invocations of operations of other components must be simulated and the context (not just the parameter settings) must be established explicitly.

**Need for instrumentation** The need for instrumentation is limited here because operations are invoked directly and the parameters are set explicitly. There is no need to measure which operations that are invoked. The “responses” are collected by a suitable measurement tool.

**Availability of information** The transaction concept is invisible when components are measured individually. The parameter settings are set systematically in order to measure which operations are invoked and how hardware devices are used. It is difficult to capture load dependencies.

**Required number of measurement runs** The number of measurement runs is dependent on the number of combinations of parameter settings for each software component.

**Completeness of measurement data** This approach allows measurement of invocations of operations in contexts which do not occur in the production workload. Hence, more general and more reusable performance characterisations of software components can be obtained.

**Expertise required in measurement experiments** Unfortunately, this mode of measurement requires expert knowledge about the component that is measured.

**Combination of Measurement Modes**

The three measurement modes may be combined. For instance, the production workload may be measured and then supplemented by extra measurement runs of individual components to obtain complete performance specifications of the software components.
13.3 Derivation of Complexity Functions

Before the measurement experiments started it was assumed that the resource demands of each software component was dependent on a set of parameters. The measurement experiments were carried out for all or a subset of the possible combinations of the selected parameters. One or more of the measurement modes in the previous section was used.

When the measurement data have been captured according to a composite work model, the complexity functions must be derived from the data. Preferably the measurement data should contain data sets which show the correspondence between invocations of an operation and invocations of its suboperations. Then the forms of the complexity functions must be found by iterated assumption and testing against the data. In the following example it is assumed that the operations of the component is dependent on only one parameter, \( N \), which denotes the size of the problem to be solved by the operations. The component has two operations which invoke two suboperations. Linear complexity functions are assumed. Data set #1:

\[
C_{\text{component}}^{\text{subcomponent}}(N = 10) = \begin{bmatrix}
\text{operation}\#1 \\
\text{operation}\#2
\end{bmatrix}
\begin{bmatrix}
430 & 120 \\
2230 & 2250
\end{bmatrix}
\]

Data set #2:

\[
C_{\text{component}}^{\text{subcomponent}}(N = 1000) = \begin{bmatrix}
\text{operation}\#1 \\
\text{operation}\#2
\end{bmatrix}
\begin{bmatrix}
13300 & 2100 \\
124000 & 150750
\end{bmatrix}
\]

The linear complexity functions that result of analysis of the measurement data are as follows:

\[
C_{\text{component}}^{\text{subcomponent}}(N) = \begin{bmatrix}
\text{operation}\#1 \\
\text{operation}\#2
\end{bmatrix}
\begin{bmatrix}
13 \cdot N + 300 & 2 \cdot N + 100 \\
123 \cdot N + 1000 & 150 \cdot N + 750
\end{bmatrix}
\]

The parameter values are the independent variables while the measured number of invocations is the dependent variable. Then the coefficient values are determined by linear regression, curvilinear regression, or any other curve-fitting analysis technique.

In addition to being the basis for derivation of complexity functions, the measurement data can be combined to obtain
• transaction traces, i.e., the sequence of invocations of operations of software components caused by each type of transaction. However, the elapsed times of the transactions are most likely inaccurate because of excessive overhead.

• total resource demands per transaction. They are useful for validation of a composite work model because they short-circuit all intermediate levels (see also work-oriented validation in Chapter 15).

• typical contexts in which each operation of each software component is invoked. In this way \( N_{\text{use}} \) is defined for each operation according to the classification schema in Chapter 10. This information can be used to assess the completeness of the performance characterisations, i.e., is \( N_{\text{use}} \) close to \( N_{\text{max}} \)?

In Chapter 10 it was explained that the form of each complexity function should be determined by code walk-through or possibly by inspection of the design documentation for the software components. Thus, white-box measurement experiments are considered necessary. Black-box measurement experiments where also the form of complexity functions are sought, are very difficult and time-consuming to undertake.
Chapter 14

The Use of Composite Work Models

This chapter demonstrates how a composite work model can generate parameter values for the different kinds of contention model that were presented in Chapter 4.

14.1 The Composite Work Model

A composite work model for the client-server application is shown in Figure 14.1. The model describes an application where the file server, RPC library, and operating system comm. are distributed between CPU#1 and CPU#2. The user-interface part of the application, also called the client, runs exclusively in CPU#1. Some of the components in Figure 14.1 are annotated with the names of some of their operations. Each component-subcomponent relationship in the composite work model is represented by a complexity matrix. In Figure 14.1 there are 20 edges and therefore 20 complexity matrices (denoted by $C_1, \ldots, C_{20}$) are needed.

The service demand of a non-primitive operation on a hardware device is calculated in two steps:

1. The operation is located in the composite work model. The work devolved on the hardware component by a single invocation of the operation is calculated by multiplication and addition of intervening complexity matrices. For instance, the work devolved by operations on the RPC library component to the CPU#1 component is $C_6 + C_7 \cdot C_8$ (see Figure 14.1). Similarly, the work devolved to the Network component is $C_7 \cdot C_9$. The devolved work of a specific operation can be found in the corresponding row in the matrix that results.
Figure 14.1: An SP model of the client-server system

2. The minimum execution time required by the operation at each hardware device is calculated by dividing the devolved work on the device by the speed of the device.

Simultaneous resource possession is common in distributed computer systems. Both hardware and software resources may be held simultaneously. Any overlapping use of hardware devices must be taken into account in order to calculate the minimum execution time (i.e., without contention) of the operation. Note that the overlap problem must be dealt with whether or not a composite work model is used.

The number of times a non-primitive operation in software is invoked during a transaction can also be calculated based on the composite work model, by computing the work devolved from the relevant top-level operations to the level of the operation in question. For instance, the work devolved by operations on the Application component to the leftmost RPC library component is \( C_3 \cdot C_5 \) (see Figure 14.1).

14.2 Static and Dynamic Aspects

If it is assumed that the need for computer resources does not change during the execution of an operation, a static description of the operation is sufficient. On the other hand, operations that acquire new resources and release resources that
are no longer needed, cannot be modelled statically because the changing need for resources may have to be represented explicitly in the contention model. Therefore the borderline between static and dynamic modelling is defined by choosing "static" operations for which the resource demands should be calculated. In Figure 14.2 the borderline goes along the following operations:

- handle_user_request
- present_result_to_user
- send_rpc_call
- receive_rpc_call
- send_rpc_response
- receive_rpc_response
- process_query

These operations are regarded as static operations and their service demands are calculated by means of the composite work model. On the other hand, the components above the borderline and their operations are modelled by a (dynamic) contention model. Operations which involve contention for software resources cannot be characterised statically with respect to performance.

Alternative borderlines between static and dynamic operations could have been chosen. For instance, a borderline just above all the primitive (hardware) components in the composite work model means that the basic operations provided by hardware devices are modelled statically while all the software components are modelled dynamically. Hence, the corresponding contention model would be parameterised in terms of basic operations on hardware devices. A borderline above the top-level component (application) indicates that a transaction is modelled statically, i.e., a transaction acquires all the resources that will be needed, starts and completes the transaction, and then releases all the acquired resources.

14.3 Utilisation of Hardware Devices

The most straightforward use of a composite work model is to calculate the utilisations of the hardware devices caused by a number of transactions in a time interval. The devolved work on each operation of a hardware device is multiplied by the service time. The utilisation of a hardware device is the total time the operations of the device provided service, divided by the time interval that the number of transactions was counted.
Figure 14.2: An example of a borderline between static and dynamic modelling

Calculation of utilisation is often the first step in a performance study. The utilisation can be used for detection of saturated hardware devices under a given workload. In addition, the bottleneck device can be detected. Residence times for the services provided by devices can be calculated by means of the following simple relationship (for open systems in the queueing modelling sense):

\[ R_k = \frac{S_k}{1 - U_k} \]

where \( R_k \) is the residence time in device \( k \), \( S_k \) is the service time, and \( U_k \) is the utilisation of the device. The transaction response time can then be estimated by summing the transaction's residence times in each hardware device. If a more detailed performance model is required, a contention model must be defined. Fortunately, the same composite work model can probably be used for providing inputs to several contention models as explained in the following sections.

14.4 Parameters in terms of Transaction Steps

Composite work models can calculate the work devoted on each computer resource for each transaction step. Three different contention models which model transaction
steps explicitly, were presented in Chapter 4. These models are presented again here for easy reference. A simulation model is shown in Figure 14.3 [100]. A queueing network model of the same transaction is presented in Figure 14.4. Yet another contention model of the same system, a Petri net, is shown in Figure 14.5. The calculation of service time for each step in the transaction is the same for all the three contention models.

![Activity diagram of a transaction](image)

**Figure 14.3: Activity diagram of a transaction**

The transactions in these models are divided into a number of steps, represented by "hold" nodes (simulation model), delay centres (queueing network), or timed transitions (Petri net). Each of these steps corresponds to one or more operations which should be represented in an accompanying composite work model. Some of the work model components in Figure 14.1 are annotated with names of operations which are referenced in the contention models.

The duration of a transaction step (i.e., not including waits for resources) is calculated as explained in Section 14.1. As an example, consider the operation `send rpc call` on the client side. The service time of this transaction step is determined by
Chapter 14. The Use of Composite Work Models

Figure 14.4: A contention model of the client-server application which explicitly takes simultaneous resource possession into account

Figure 14.5: A partial Petri net which represents a transaction
the time that \texttt{CPU#1} is used. First, the number of CPU instructions to be executed for each operation provided by the left-most \texttt{RPC library} component in Figure 14.1 is calculated by the following matrix calculation: $C_6 + C_7 \cdot C_8$. The invocation count for the operation on \texttt{CPU#1} which is caused by \texttt{send rpc call} is picked from the result matrix and multiplied by the appropriate instruction execution time. Note that in general there will be a vector of invocation counts which correspond to separate CPU instruction types distinguished by the model. In the simplest case only one instruction type is necessary.

### 14.5 Parameters in terms of Entire Transactions

In the previous section it was demonstrated how a composite work model could provide devolved work in terms of transaction steps. In this section it is shown how the same composite work model devolves work in terms of entire transactions, i.e., the contention model does not take into account each step in the transaction. Only one contention model, a queuing network model, is used as example (Figure 14.6). Note that this model does not represent overlapping service provided by the hardware devices.

![Figure 14.6: A coarse contention model of the client-server application](image)

Each of the primitive components in the composite work model in Figure 14.1 corresponds to one queueing centre in Figure 14.6. Assume that all the machine instructions that are offered by the CPUs, can be executed during one clock cycle. The clock cycle time may be regarded as the service times of the CPUs: $S_{cpu1}$ and $S_{cpu2}$. The number of machine instructions to be executed are generated by the composite work model in terms of each top-level operation (in this case transaction). The numbers are called visit counts: $V_{cpu1}$ and $V_{cpu2}$. The service time of the disk, $S_{disk}$,

\footnote{It is assumed that the CPU does not use the network continuously in this phase because data must be packed into messages before each message is sent}
is the disk access time. The number of disk accesses, $V_{\text{disk}}$, is also generated by
the composite work model in terms of each top-level operation. The service time of
the network, $S_{\text{network}}$, is the time it takes to send a message from the origin to the
destination in the network. The number of network messages, $V_{\text{network}}$, is generated
by the SP model in terms of each top-level operation.

In general, contention models cannot use the devolved work demands directly as
parameter values because they represent a transaction's total work demands at each
hardware device. Most transactions are routed between the hardware devices so
that a hardware device may be visited more than once. Therefore, the devolved
work of a transaction must be transformed into work demand per visit and visit
counts. The product of these two measures should equal devolved work. I/O work
demand correspond to I/O visit count but that is not the case for CPU work demand.
However, CPU visit counts can be inferred from I/O counts. The feedback loops
over CPU#1 and CPU#2 in Figure 14.6 make it possible to characterise CPU work
demand directly in terms of visit counts. Some analytic queueing network models are
in product form (see Section 4.6). As a consequence of the product-form property,
only the product of service time and visit count is significant. Service demand, $D$, is
defined as $D = V \cdot S$. That means that work demand need not be transformed into
work demand per visit and visit counts, i.e., devolved work multiplied by appropriate
service times can be used directly as parameters in the contention model.

14.6 Parameters in terms of Software Processes

In the two previous sections it was shown that a composite work model can pro-
vide resource demand parameters for an entire transaction and for each step of the
transaction.

![Diagram](image)

**Figure 14.7:** A software contention model and a device contention
model for a client-server system (see Chapter 3.1)

When competition for software processes is significant, also the devolved work on
non-primitive work model components must be calculated.

For instance, Figure 14.7 shows a software contention model and a device contention
model for the client-server example. The software contention model indicates that the visit count from the client part to the server part is needed in order to use the Method of Layers (Section 4.8) to solve the models in Figure 14.7. The composite work model must therefore calculate the number of invocations of the File server component (= $C_3$ in Figure 14.1) as well as the number of invocations on each of the hardware devices. The latter invocation counts are needed by the device contention model. They are calculated as follows ($W^H_L$ means the work devolved from component $H$ to component $L$ and $C_i$ refers to a complexity matrix in Figure 14.1):

$$W^{Appl}_{CPU \#1} = C_1 \cdot C_2 + C_3(C_4 + C_5(C_6 + C_7 \cdot C_8))$$

$$W^{Appl}_{Network} = C_3 \cdot C_5 \cdot C_7 \cdot C_9$$

$$W^{Fileserver}_{Network} = C_{10} \cdot C_{11} \cdot C_{12}$$

$$W^{Fileserver}_{CPU \#2} = C_{10}(C_{11} \cdot C_{13} + C_{14}) + C_{15} + C_{16}(C_{17} + C_{18} \cdot C_{19})$$

$$W^{Fileserver}_{Disk} = C_{16} \cdot C_{18} \cdot C_{20}$$
Chapter 15

Assessment of Completed Composite Work Models

When the measurement experiments have been carried out and the measurement data have been analysed and coefficients in complexity functions have been assigned values, the final assessment of the completed composite work model can be carried out. The long-term cost of composite work modelling should be assessed. Failure to validate a completed SP model may be caused by incomplete coverage of software or inaccurate complexity functions. The multiple layers in a composite work model may propagate and magnify errors in the complexity functions in each layer. Both qualitative and quantitative aspects must be validated. This chapter describes how a completed SP model is assessed.

15.1 Long-Term Cost of Parameter Capture

The rationale for using composite work models is to reduce the cost of building performance models. This is achieved by capture of reusable measurement data which can be combined in different ways to generate parameter values for performance models. The key question is whether the long-term cost of maintaining a composite work model is less than the cost of building a new work model from scratch every time it is needed.

The granularity of reusable measurement data is normally finer than measurement data which are captured from scratch every time they are needed. The cost of measurement and analysis of measurement data increases when the granularity of measurement data is made finer. Consequently, the extra cost of capturing reusable measurement data must be regarded as an investment which involves a risk of not being profitable. The profitability of the composite work modelling approach cannot be assessed after a few completed composite work models because of the initial investments.
The investment in reusable measurement data must be justified by the cost savings each time a work model component can be reused. Thus, there are two conditions for reduced long-term cost of composite work modelling:

1. The cost of reuse must be less than the cost of measurement from scratch.

   The cost of measurement reuse is mainly dependent on the cost of retrieving the relevant complexity functions and the cost of capturing the parameter values for the complexity functions. Measurement from scratch every time the measurement data are needed, does not need parameterised complexity functions. Hence, to improve the long-term cost of composite work modelling, it is important to minimise the number of parameters and to minimise the need for measurement to find parameter values.

2. The performance characteristics must be reused enough times to justify the investments.

   The less the gain in cost every time a work model component is reused, the higher the required number of reuse. Hence, it is important to focus on the reusability of the work model components when the granularity of a composite work model is made finer.

In summary, the criteria for cost-efficient composite work modelling are:

- Minimise the number of parameters in complexity functions
- Select parameters whose values are easy to find
- Avoid work model components that are less likely to be reused and contribute little to cost savings when they are reused.

The accuracy of the composite work model is a limiting factor for reduction of cost by reuse. As discussed in Section 11.6, many layers in a composite work model tend to increase the risk of error propagation and error magnification. Therefore the composite work models must be properly validated with respect to both qualitative and quantitative aspects.

15.2 Qualitative Validation

Ideally, completeness, or software coverage, should have been one of the criteria for selection of composite work model granularity. This is difficult because the significance of software that is left out is best tested during the validation of the model! It is normal that surprisingly large portions of the software are executed in extraordinary situations only. It must be decided whether such software should be included in the composite work model at all. One might argue that nothing behaves
as normal in a computer system in an extraordinary state, so most models will be wrong unless they are built specifically for extraordinary conditions.

Complete coverage of software is possible if the entire software can be partitioned into components each of which can be measured as one entity. For instance, this is difficult for Tandem software because the software is distributed over hundreds of processes. The resource usage of a group of processes cannot be measured as one entity. That implies that all processes which consume significant amounts of computer resources must be measured explicitly and the interactions between the processes must be measured explicitly. The finer the granularity of composite work models, the greater the risk of incomplete coverage of the software. For instance, the risk of incomplete coverage is very high when functions, e.g., in C programs, are invoked with functions as parameters. Communication software, e.g., the remote procedure call library, often contains lots of functions passed as parameters to other functions. In addition, particular care must be taken in order to include all overhead work that is done on behalf of the applications. Typical overhead work is the administration carried out by the operating system. Moreover, such overhead work is typically load-dependent.

With respect to a contention model the following criterion applies for completeness and flexibility of a composite work model:

Composite work model completeness requires that all resource demands in a contention model represent the execution of one or more operations that are defined at some level in the composite work model.

This implies that the greater the number of levels of operations that are defined in the composite work model, the more flexible the parameter capture for a contention model. Conversely, the fewer the service demands required by the contention model, the lower the requirements on composite work modelling [100].

The completeness of a composite work model requires that any horizontal interception of the model describes a set of operations which can implement any top-level operation that is modelled by the composite work model. An interception divides a composite work model in an upper part and a lower part such that no component in the upper part is inferior to a component in the lower part. The lower part of the composite work model constitutes a virtual machine which by definition must provide the full functionality for any operation that is implemented by the virtual machine. In Figure 15.1 the operations (shown as annotations) along each of the three interception lines must constitute a complete repertoire for the implementation of the top-level operations. The first alternative interception is shown as a horizontal medium thick line just above the Application component. In this case the virtual machine provides only one top-level operation, the transaction. The second interception line is dashed and thick. Here the operations of the User interface, RPC library and Services components constitute the virtual machine. The third and last interception line is continuous and thick. It is drawn just above the hardware components. In this case the virtual machine corresponds to the real machine!
In addition to representing the required operations in a composite work model, the complexity functions of these operations must be parameterised so that buffering effects and concurrency effects are represented properly. It is also important that the modelling assumptions for all the complexity functions are compatible, i.e., that the model is constructive (Section 1.2). Constructivity requires that the dependencies between the work model components are properly represented so that the properties of the entire software is obtained when the model components are put together.

15.3 Quantitative Validation

A composite work model represents the number of interactions between software components and the number of times hardware operations are invoked. The model is static. The main sources of inaccuracies are incompleteness, functional variety, data dependencies, state dependencies and sequence dependencies. In the ideal case it should be feasible to obtain accurate work models, but in practice only approximate work models can be found.

The accuracy of composite work models can be assessed according to at least two different criteria. One which involves only the composite work model, work-oriented validation, and one which involves both the composite work model and the corresponding contention model, performance-oriented validation.
15.3. Quantitative Validation

![Diagram](image)

Figure 15.2: Work-oriented validation

![Diagram](image)

Figure 15.3: Performance-oriented validation
Work-oriented validation The devolved work provided by a composite work model should be compared with the devolved work calculated by a corresponding monolithic work model (see Figure 15.2). After all, we want to make sure that the division in multiple layers in a composite work model does not degrade the accuracy of the devolved work significantly. Validation measurement of the workload proceeds by measurement of the software as a whole. The most accurate validation measurement data can be captured when the entire software is a measurable component (Section 11.2). Unfortunately, this is not the case for distributed systems. The physically disjoint software components in a distributed system must be measured separately and the interactions between the components must be taken into account. This means that it is difficult to validate composite work model of distributed systems! Work-oriented validation of distributed systems is feasible if suitable instrumentation and suitable measurement tools can characterise the computer resource demands in terms of the top-level operations (e.g., transactions) of the composite work model to be validated. Then it is possible to short-circuit all the intervening layers to check that the propagation of errors is within acceptable limits.

Performance-oriented validation Performance indices are used as the basis for comparison in this approach to validation of a composite work model [41] (see Figure 15.3). The rationale for the performance-oriented criterion is that different workloads (models) may have the same effect on the performance of the computer system. Thus, a simple and compact workload model may be just as good as a large and complex workload model. First, a load model and a composite work model are used for calculation of parameter values to the contention model (see also Section 1.1). Then the contention model is used for calculation of performance indices. Secondly, the real system is run under the real workload while the same performance indices are captured. If the performance indices match each other one can conclude that the load model, work model and the contention model are correct for one specific workload. Several different workloads must be applied to the real system and the contention model in order to increase the confidence in the models. An advantage of the performance oriented validation is that performance indices such as utilisation, throughput, response times, and queue lengths may be easier to capture with contemporary measurement tools then information about computational work demands. A disadvantage is that it is difficult to find the source of inaccuracies when the models (load, work and contention) are not validated separately.

Discussion Work-oriented validation should be preferred if suitable measurement data can be obtained. It is then possible to validate the composite work model in isolation. Performance-oriented validation may be preferable if the composite work model contains load-dependent complexity functions. A disadvantage of this type of validation is that the correctness of a composite work model is closely related to specific load and contention models. This is the very opposite of the rationale of composite work modelling, namely the reusability of the work model components.
Part IV

Concluding Discussions and Remarks
Chapter 16

Concluding Remarks

A method for static performance characterisation of software components in distributed systems has been presented. Composite work modelling has been examined as the means for planning of measurements and practical use of reusable measurement data.

16.1 Recapitulation of Motivation and Objectives

The motivation for composite work modelling is twofold. Such modelling has the potential of reducing the cost of building performance models of existing systems. In addition, composite work models can be used, at a reasonable cost, in order to ensure that new systems can meet their performance requirements. Computer systems are normally built by a combination of reuse of existing software components and development of new components. If the system is built from existing components, it is important to know the performance characteristics of each of the components. If the system is developed (partly) from scratch, it is important to design the components with the performance properties in mind.

The objective has been to find how to capture and reuse performance characteristics of software components in order to reduce the cost of obtaining sufficiently accurate parameter values for contention models of the computer system.

The approach has been to capture measurement data in such a way that reusable complexity functions can be derived from them. Composite work models have been used as the means for choosing the granularity and parameterisation of complexity functions. The properties of composite work models were examined with respect to degree of parameterisation and granularity. The granularity of a composite work model must be chosen such that:
• its components correspond to the size of the software components that are most likely to be changed or exchanged.

• the model is robust with respect to likely modifications.

• serious accuracy degradation is avoided. This problem occurs when the composite work model has too many levels.

• it represents a measurement plan which can be completed without exceeding the cost limit of the measurement experiments.

• it is easy to calculate resource demands in terms of service demands.

• it is possible to extract resource demands which are compatible with the parameter needs of the contention model.

16.2 Major Results

The Framework and its Limitations

The framework for composite work modelling has been elaborated on and sharpened by applying it to real distributed systems. The framework provides the means for reuse of measurement data. Common performance modelling and measurement activities have been related in new ways. In particular, the close integration of work modelling and planning of measurement is believed to be original. Composite work modelling provides a coherent framework for sizing studies. The composite work modelling approach has been pushed to its limits, and therefore the areas which must be improved have been illuminated. The limitations of the approach fall into three categories: The inherent problems (method), the practical problems (tool support), and the availability of information about structure and performance.

The inherent problems of workload modelling (functional variety, data dependence, state dependence, and sequence dependence) manifested themselves at each layer of composite work models. The case studies showed that the "functional variety" problem was most severe for higher-level work components, while the data-dependence problem was most severe for lower-level work components. Operations which pass very general arguments such as queries, present special difficulties for parameterisation of complexity functions in a composite work model. It was revealed that the work-complexity of software components in client-server architectures frequently is state-dependent, in particular because the communication infrastructure does not allow that all the results are returned as one message. The data-dependencies call for parameterisation of complexity functions, the functional variety calls for clustering analysis, and the state-dependencies call for splitting of operation in the work model. These techniques were presented as part of the step-by-step method for composite work modelling.
The practical problems can usually be solved by extensive software tool support for measurement and modelling. Any modelling activity requires a lot of tedious (human) work and some creative work. The software tools should minimise the need for the tedious work so the focus is moved to the creative part. During the work on the case studies there was a need for tool support for storage and analysis of measurement data and for experimentation with SP models, e.g., to put components together in different ways. There was also a problem which concerned the assignment of values to parameters in SP models when there were conflicting assignments.

The availability of information Tools are of no help if the required information itself is not available. The main problem in the case studies was to capture information about the co-operation between software components. It was also experienced that black-box experiments are difficult to perform on complex software components. Inspection of the source code of the software components or inspection of the design specification is necessary. Instrumentation of the source code and/or development of new measurement techniques may be necessary to make the required information available.

The Trade-off within the Limits

The three categories of problems that were presented above, limit the applicability of composite work modelling. Within these limits, the cost of measurement must be traded against the loss of accuracy. The basis for this trade-off is the reusability of work model components, the accuracy of the devolved work, and the cost of the corresponding measurement experiments.

Reusability of work model components requires that the systems to be represented by a composite work model must themselves consist of reusable components. In addition, the reusability depends on how many of the possible number of combinations of parameter values that have been measured. Composite work modelling is appropriate for reconfigurable computer systems which consist of (relatively) independent components. Distributed, heterogeneous and open systems are designed to be reconfigurable and are therefore well suited for composite work modelling. These properties enable software development by composition of software components.

Accuracy The actual accuracy cannot be determined before the measurement experiments have been carried out. However, some considerations of accuracy can be carried out by regarding the structure of the composite work model: In the worst case the errors introduced by operations along an operation-suboperation path in the model are approximately additive. That implies that the introduction of additional layers in the composite work model requires that the new measurement data must be more accurate than the measurement data for a coarser work model. When the number of layers in a composite work
model increases, there tend to be more paths in the work model. The inaccuracy along the different paths in the composite work model may cancel out, but this effect cannot be estimated before the complexity functions are known.

Cost The cost of measurement according to a composite work model is dependent on the number of measurement runs that require human intervention and the number of operation-suboperation relationships that must be analysed. Reuse of measurement data via complexity functions has great impact on the cost of measurement analysis, and it may influence the number of measurement runs that are required.

The Trade-off The cost of composite work modelling is mainly dependent on the degree of reuse. The reusability of work model components are dependent on the granularity of work model components and the degree of parameterisation. In general, the accuracy of the calculated devolved work degrades when the number of layers in the composite work model increases. Consequently, there is a conflict if reusability requires several layers in the work model.

16.3 Discussion of Results

The results show that there are several problems that have to be solved before the objective of this thesis is fulfilled. The practical problems and the unavailability of information can be solved by improving the software tools. On the other hand, the inherent problems limit the applicability of composite work modelling. A step-by-step method for composite work modelling was proposed in order to investigate the inherent problems before a distributed computer system is modelled. The closer one gets to the limits of composite work modelling, the higher the risk of accuracy degradation and cost overruns.

The main benefit of the results is the elucidation of concerns during composite work modelling. Many examples from real computer systems have been presented. Since distributed (and possibly heterogeneous) systems are very complex, the potential user of composite work modelling will get useful information about the benefits and problems of modelling of a particular (distributed) system. In particular, data dependencies may cause problems that should be known before expensive measurement runs are carried out. When the potential problems are assessed in advance of the modelling study it is possible to decide whether a composite work model is worth the cost and if it is feasible to obtain a reasonable accuracy of the devolved work.

It appeared that the flexibility of parameter generation for contention models increases as the number of layers in the composite work model increases. Hence, one single composite work model can provide parameter values for contention models of different level of detail. In particular, this applies to contention models which entail both software processes and hardware devices.
The best conditions for cost reduction by reuse of measurement data requires that the software is designed for reuse or at least designed for exchange of components. In addition, the software should be designed for measurement so that the instrumentation can count the number of invocations of component operations. Also the dependencies of an operation being invoked and a unique identification of the current transaction should be recorded properly.

### 16.4 Experience Gained from the Case Studies

**Tandem** The software components of the hospital system could not be properly characterised with respect to computer resource demands because measurement data could not be captured in terms of specific transactions and not in terms of the functionality of the software components. Hence, it was very difficult to establish cause and effect when software interactions were measured. This calls for extensive software instrumentation. Another problem was that the software is spread over hundreds of processes each of which must be measured with respect to resource usage and interaction with other processes. Thus, the possible choices of granularity of measurement data were very limited. In addition, the large number of processes increases the cost of measurement data analysis. This case study demonstrates clearly the extreme requirements that composite work modelling presents to the measurement tools (see discussion in Section 11.2). It also demonstrates that black-box measurement experiments are not very useful. To define the factors that influence the execution of a function it is necessary to look at the source code of the function. Tandem systems provide the MEASURE tool which is an appropriate measurement tool for tuning and system administration. However, it is not suitable for composite work modelling.

**ICL** The ICL case study showed that a software component can quite easily be exchanged with another component with identical interface. It was also demonstrated that it is feasible to move components around in a distributed heterogeneous computer system. However, the components that are involved in data storage typically have very general functionality that accepts queries in textual form as arguments. This kind of function argument cannot be characterised in general by a complexity function because a large number of different queries can be given as argument and because of query planning and optimisation. The software components which take care of communication, i.e., the distribution infrastructure, can be characterised more easily (with respect to performance). The functions are typically dependent on message lengths, how much data that are transferred, and the average number of retransmissions. Another experience from the ICL study is the need for system-independent characterisation of processing demands because it is very likely that a program is moved to a completely different hardware platform. In UNIX systems there is a definite need for a comprehensive approach to performance measurement. The current plethora of measurement tools makes measurement-in-the-large both time-consuming and difficult.
16.5 Experience with SP

The Structure and Performance Specification (SP) method was applied extensively during my work on composite work modelling. The corresponding SP tool was used only occasionally because the then current version of the tool did not provide the needed functionality.

The experience gained during this work indicates that the fundamental principles behind the SP rules are difficult to apply in practice on complex computer systems. Considerable initial effort is required because it is necessary to "look behind" the structure of the computer system. SP models seems to be a canonical representation of computer systems. Consequently, by using existing SP models as examples, the SP modelling process is greatly simplified. The observation that similar work model components tend to appear in composite work models of different systems of the same architecture strongly supports the idea of reuse of work model components.

During the work on SP modelling of distributed systems it was found that one of the SP rules concerning distribution was too restrictive. Thus, either an appropriate SP model of the system could not be built or an awkward SP model had to be built in order to avoid the faulty restriction.

The SP method tends to require that infrastructure software components are modelled explicitly. The problem is that operations provided by the software infrastructure are very general. Therefore strong data-dependencies are difficult to avoid.

Several shortcomings of the SP tool were experienced. The following requirements or suggestions for improvement materialised during practical work on the case studies:

- Distinct phases in the use of the tool. There is a need for top-down modelling where the components are defined first and then the implementation details are added afterwards.

  1. Draw composite work model ("draw boxes and connect them"). No modelling rules apply.
  2. Rudimentary characterisation of components
  3. Rudimentary characterisation of composite work model
  4. Complete characterisation of components with detailed information
  5. Use the completed SP model for calculation of devolved work on any component in the SP model.

The current tool provides only bottom-up modelling.

- Representation of simultaneous resource possession. There should be a way to indicate simultaneous usage of hardware devices, e.g., it is common that a CPU and a disk is utilised simultaneously. If information about such usage is available during composite work modelling, this information should be put into
the composite work model. As was explained in Chapter 14, this information will be very useful when the composite model is used for capture of parameter values for contention models.

- Representation of possible synchronisations each of which may act as a bottleneck in the computer system. Operations which are synchronised with other operations must be modelled explicitly in the composite work model so that they can be properly represented in a contention model (Chapter 14).

- Support for change. The effects of a change in an SP model may propagate so that a lot of human intervention is required to establish a new consistent SP model after the change.

Note that a new version of the SP tool has been implemented, partly influenced by the experiences from this work. Hence, some of these complaints do no longer apply.

16.6 Commercial Exploitation

The benefits of structuring software performance measurement according to a composite work model are most evident when it is applied to large-scale, “open” software running on distributed, heterogeneous hardware. In a typical commercial case there would be many alternative software and hardware configurations. The performance properties of software components would be measured by their developers or by other experts. The performance properties would be stored and maintained in a library where they could be retrieved easily. The amount of fine-grained measurement data would be very large so that a tool like SP would be beneficial in order to manage the complexity and enable timely generation of performance properties of different software and hardware configurations. The performance characterisation of the software components that is necessary for composite work modelling, is a multi-company endeavour rather than a one-person endeavour.

16.7 Critique

No validated composite work model has been presented in this thesis. Composite work modelling is still in an infant stage where all kinds of problem are encountered. Although it would have been feasible to build a composite work model for the client-server example, most of the work was done on real distributed computer systems. The advantage of this approach is that the problems of large-scale work modelling were experienced early in the doctoral work. Hence, the discussions could take account of these problems. The disadvantage is of course that the results carry less weight than if a validated composite work model existed. In any case it would have required too much work for one person to provide a validated model for a large-scale system.
The use of physical memory and data storage at different levels in the memory hierarchy have not been dealt with properly. A proper statistical treatment of propagation of inaccuracies was not performed partly because of the lack of numbers to analyse. These two issues should be pursued in the further work on composite work modelling.

Although this thesis was targeted at transaction-oriented systems, no special attention was given to the transaction management software.

### 16.8 Further Work

Little practical experience on composite work modelling existed before this doctoral work started. Some more experience have been gained, in particular on distributed systems. Obviously, much more experience is needed. Composite work modelling entails many complex modelling tasks which could only be described briefly in this thesis.

**Aspects of Composite Work Modelling** Several commercial software components should be characterised with respect to performance to gain thorough experience. In particular database management systems are critical. Such performance characterisations would also allow exploitation of statistical techniques. Physical memory is a very common bottleneck in contemporary systems which deserves attention. In particular, user interfaces such as X-windows need large contexts which demand a lot of memory to avoid excessive paging and swapping. In addition, buffers in, e.g., database systems tend to be very large in order to optimise performance. This issue is treated in [79, 89, 88].

**Improvement of Measurement Tools** Composite work modelling poses requirements on which information that must be captured as well as requirements on the degree of automation of instrumentation, measurement setup and analysis of measurement data. In particular, information about invocations of operations of software components is required. The automation of measurement makes it possible to capture more measurement data and the focus is moved from "how to measure" to "what to measure".

**Integration with Performance Modelling of Software Processes** Woodside [102, 73, 103] and Rolia [87] are working on performance evaluation techniques which take into account contention for both software processes and hardware devices. It should be investigated how composite work modelling can be integrated with these performance evaluation techniques.
Integration with Software Performance Engineering  Opdahl [78], Brataas [20] and Smith [92] elaborate on frameworks for software performance engineering with emphasis on performance considerations during software design. It should be investigated how measurement-based composite work modelling can improve software performance engineering.

Parameterisation with respect to time  It may be of interest to make complexity functions dependent on time in order to represent, e.g., expected growth of data volumes. In this way time is introduced in the multi-dimensional parameter space which is explored during what-if analysis.

OSF/DCE – Distributed Computing Environment  During the doctoral work only two software architectures, PATHWAY and ANSA, were examined. The experience from these kinds of system should be applied in the standardisation of Open Software Foundation’s (OSF) Distributed Computing Environment (DCE). OSF/DCE provides services and tools that support the creation, use, and maintenance of distributed applications in a heterogeneous computing environment [93].

16.9  Summing Up

The “measurement data dilemma” turned up several times during the work on the thesis. It may seem strange that the presentation of a measurement-based method hardly contains any measurement data at all. The main practical experience from this work is that the contemporary measurement tools do not support well the capture of software interactions. It is my belief that this work has illuminated the current limits of available measurement data and measurement tools for distributed computer systems. Unfortunately, the problems concerning information availability overshadowed the discussion and analysis of complexity functions which were supposed to be the most prominent issues in the thesis. To proceed with practical application of composite work modelling, the supply of performance information must be improved.

I hope that the thesis has shed some light on the relationships between different modelling activities that are necessary in performance engineering and convinced the reader that model-driven measurement in terms of software interactions as well as hardware device usage plays or should play an important role in such activities.
Part V

Appendices and Bibliography
## Appendix A

### Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADT</td>
<td>Abstract Data Type</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DEMOS</td>
<td>Discrete Event Modelling on Simula</td>
</tr>
<tr>
<td>DBMS</td>
<td>Database Management System</td>
</tr>
<tr>
<td>FESC</td>
<td>Flow Equivalent Service Centre</td>
</tr>
<tr>
<td>HIT</td>
<td>Hierarchical Evaluation Tool</td>
</tr>
<tr>
<td>ICL</td>
<td>International Computers Limited, (a computer company based in the UK)</td>
</tr>
<tr>
<td>MOL</td>
<td>Method of Layers</td>
</tr>
<tr>
<td>PIT</td>
<td>Process Interaction Tool</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computer</td>
</tr>
<tr>
<td>Rit</td>
<td>The Regional Hospital in Trondheim, Norway</td>
</tr>
<tr>
<td>SP</td>
<td>The Structure and Performance Specification Method/Tool</td>
</tr>
<tr>
<td>SPE</td>
<td>Software Performance Engineering</td>
</tr>
<tr>
<td>UBG</td>
<td>User Behaviour Graph</td>
</tr>
<tr>
<td>WAT</td>
<td>Workload Analyser Tool</td>
</tr>
</tbody>
</table>
Appendix B

International Computers Limited

This appendix presents the operations that are provided by the DAIS prototype at ICL/Manchester, UK. The components are described in Chapters 2 and 3.

Application Component – Bank/teller

- get_customer key = account_number
- get_customers key=surname
- get_all_customers nokey
- get_company key = account_number
- get_all_companies nokey
- withdraw
- deposit
- statement
- balance

Terminal Handling Component

The terminal (screen) handling functions provided by the Ingres library were used.
Database Server Component

- Startup
- LIS.CONNECT
- LIS.SQL
- LIS.FETCH
- Disconnect

LISADMIN Component

- Startup
- Start.LIS
- Get.id

LIS Components

- Startup
- Operation
- Fetch
- Disconnect

Trader

- Register
- Lookup
- Delete
- Purge
- Search
- Shutdown
- Update
Remote Execution Protocol – REX

- startup
- send
- receive
- reply
- cast
- reject
- disconnect
- cleanup

Message Passing Services – MPS

- startup
- send
- receive
- tick
- cleanup
Appendix C

The Client-Server Example

Only the functions that are used in the example system are listed here.

The Bank Application Program (Client)

The short form of each operation is shown in brackets.

- `create_account` (create)
- `read_account` (read)
- `update_account` (update)
- `delete_account` (delete)
- `list_customers_by_name` (custnam)
- `list_customers_by_balance` (custbal)

The File Server

- `file_create_account.1` (fcreate)
- `file_read_account.1` (fread)
- `file_update_account.1` (fupdate)
- `file_delete_account.1` (fdelete)
- `file_list_accounts_by_name.1` (fcustnam)
- `file_list_accounts_by_balance.1` (fcustbal)
The Remote Procedure Call Library

The remote procedure call (RPC) library offers the following functions:

- `clnt_create` (init RPC session)
- `clnt_destroy` (quit RPC session)
- `clnt_call` (for every RPC call) (`ccall`)
- `pmap_unset` (init RPC session)
- `svcuda_create` (init RPC session)
- `svctcp_create` (init RPC session)
- `svc_register` (init RPC session)
- `svc_run` (init RPC session)
- `svc_getargs` (for every RPC call) (`sgarg`)
- `svc_sendreply` (for every RPC call) (`srepl`)
- `svc_freeargs` (for every RPC call) (`sfrarg`)
- `bzero` (for every RPC call) (`bzero`)
- pack and unpack (for every RPC call)

Some of the functions are invoked by the client process (usually prefixed by `clnt_`) while some functions are invoked by the server process (usually prefixed by `svc_`).

The Curses Screen Handling Library

- `initscr`
- `endwin`
- `move`
- `getstr`
- `mwprintw`
- `clear`
- `clrtoeol`
The Standard C Library

The functions provided by the standard C library will not be listed here because of the large number of functions in the library. Please consult a reference manual such as [62].

Unix File Access Functions

- close
- fstat
- lseek
- open
- read
- unlink
- write

Unix Communication Functions

- bind
- getsockname (getsock)
- listen
- recvfrom
- select
- sendto
- socket
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>Relationships between performance modelling activities</td>
<td>xi</td>
</tr>
<tr>
<td>1.1</td>
<td>The (lack of) balance between resources needed and resources provided as described in [92]</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>A graph describing a possible relationship between transaction response time and hardware device utilisation in a transaction-processing computer system</td>
<td>4</td>
</tr>
<tr>
<td>1.3</td>
<td>Issues that must be addressed to reduce reduced long-term measurement cost</td>
<td>5</td>
</tr>
<tr>
<td>1.4</td>
<td>The relationship between workload and performance (from [42, pp. 5])</td>
<td>6</td>
</tr>
<tr>
<td>1.5</td>
<td>The load parameters are applied directly to the contention model while the mapping between offered services and resources is modelled by a composite work model.</td>
<td>6</td>
</tr>
<tr>
<td>1.6</td>
<td>Software as a mapping between external workload and hardware</td>
<td>9</td>
</tr>
<tr>
<td>1.7</td>
<td>Software as a hierarchy of mappings between external workload and hardware</td>
<td>10</td>
</tr>
<tr>
<td>1.8</td>
<td>A composite work model of a client-server system</td>
<td>12</td>
</tr>
<tr>
<td>1.9</td>
<td>An example of a component-subcomponent relationship</td>
<td>12</td>
</tr>
<tr>
<td>1.10</td>
<td>An example of a complexity matrix</td>
<td>13</td>
</tr>
<tr>
<td>1.11</td>
<td>Performance engineering during software design</td>
<td>15</td>
</tr>
<tr>
<td>1.12</td>
<td>Sizing of a computer system</td>
<td>16</td>
</tr>
<tr>
<td>1.13</td>
<td>Performance evaluation of different alternatives</td>
<td>16</td>
</tr>
<tr>
<td>2.1</td>
<td>No co-operation between processes</td>
<td>22</td>
</tr>
<tr>
<td>2.2</td>
<td>Co-operating processes</td>
<td>23</td>
</tr>
<tr>
<td>2.3</td>
<td>Distributed co-operating processes</td>
<td>23</td>
</tr>
<tr>
<td>2.4</td>
<td>Two communicating processes</td>
<td>24</td>
</tr>
<tr>
<td>2.5</td>
<td>Transaction execution paths</td>
<td>24</td>
</tr>
<tr>
<td>2.6</td>
<td>The steps of a remote procedure call (from [97, pp. 457])</td>
<td>28</td>
</tr>
<tr>
<td>3.1</td>
<td>The example application as a client-server system</td>
<td>32</td>
</tr>
<tr>
<td>3.2</td>
<td>The menu hierarchy in the client-server system</td>
<td>33</td>
</tr>
<tr>
<td>3.3</td>
<td>Software components of the bank application (client)</td>
<td>35</td>
</tr>
<tr>
<td>3.4</td>
<td>Software components of the file server</td>
<td>35</td>
</tr>
<tr>
<td>3.5</td>
<td>Configuration of a DAIS system at ICL</td>
<td>36</td>
</tr>
<tr>
<td>3.6</td>
<td>A Tandem computer with 6 CPUs</td>
<td>39</td>
</tr>
<tr>
<td>4.1</td>
<td>Activity diagram of two transactions which contend for a resource</td>
<td>49</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4.2</td>
<td>Activity diagram of two transactions which contend for a server</td>
<td>49</td>
</tr>
<tr>
<td>4.3</td>
<td>A partial activity diagram which shows that transaction#1 is modelled</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>independently of any other transactions</td>
<td></td>
</tr>
<tr>
<td>4.4</td>
<td>Simultaneous resource possession in activity diagrams</td>
<td>50</td>
</tr>
<tr>
<td>4.5</td>
<td>Typical activity diagram of a software server</td>
<td>51</td>
</tr>
<tr>
<td>4.6</td>
<td>Activity diagram of a transaction</td>
<td>52</td>
</tr>
<tr>
<td>4.7</td>
<td>A queueing centre (a) and a delay centre (b)</td>
<td>53</td>
</tr>
<tr>
<td>4.8</td>
<td>An example of a queueing network model with 2 workload classes</td>
<td>53</td>
</tr>
<tr>
<td>4.9</td>
<td>Extra features in queueing network models</td>
<td>54</td>
</tr>
<tr>
<td>4.10</td>
<td>The procedure for solving product-form queueing networks</td>
<td>56</td>
</tr>
<tr>
<td>4.11</td>
<td>A queueing network model of the client-server system</td>
<td>59</td>
</tr>
<tr>
<td>4.12</td>
<td>A partial queueing network model which takes simultaneous resource</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>possession into account</td>
<td></td>
</tr>
<tr>
<td>4.13</td>
<td>Explicit modelling of time-sharing of a CPU. The total resource</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td>demand on the CPU is V times the duration of the time slot</td>
<td></td>
</tr>
<tr>
<td>4.14</td>
<td>The basic concepts in Petri nets</td>
<td>62</td>
</tr>
<tr>
<td>4.15</td>
<td>Extensions to standard Petri nets. A timed transition is shown in a)</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>and an immediate transition is shown in b). In c) a random switch is</td>
<td></td>
</tr>
<tr>
<td></td>
<td>necessary to determine which immediate transition that should fire</td>
<td></td>
</tr>
<tr>
<td>4.16</td>
<td>A partial generalised stochastic Petri net which represent a transaction</td>
<td>66</td>
</tr>
<tr>
<td>4.17</td>
<td>A software contention model and a device contention model for a</td>
<td>68</td>
</tr>
<tr>
<td></td>
<td>client-server system (see Chapter 3.1)</td>
<td></td>
</tr>
<tr>
<td>5.1</td>
<td>Different kinds of workload model</td>
<td>71</td>
</tr>
<tr>
<td>5.2</td>
<td>Various phases for the construction of static and dynamic workload</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td>models (from [27])</td>
<td></td>
</tr>
<tr>
<td>5.3</td>
<td>An example of a user behavior graph for the client-server example</td>
<td>77</td>
</tr>
<tr>
<td>5.4</td>
<td>Hierarchic performance modelling ([56])</td>
<td>78</td>
</tr>
<tr>
<td>6.1</td>
<td>An composite work model of the client-server application</td>
<td>84</td>
</tr>
<tr>
<td>6.2</td>
<td>An information processing operation in a work-space W (from [58])</td>
<td>84</td>
</tr>
<tr>
<td>6.3</td>
<td>The abstract virtual machine (from [58]). The small circles denote</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td>operations.</td>
<td></td>
</tr>
<tr>
<td>6.4</td>
<td>Different levels of processing in an SP model</td>
<td>87</td>
</tr>
<tr>
<td>6.5</td>
<td>Some basic SP concepts and their graphical representation</td>
<td>89</td>
</tr>
<tr>
<td>6.6</td>
<td>The relationship between a component and its subcomponents</td>
<td>90</td>
</tr>
<tr>
<td>6.7</td>
<td>An illegal composite work model fragment if the links are of the</td>
<td>94</td>
</tr>
<tr>
<td></td>
<td>same type</td>
<td></td>
</tr>
<tr>
<td>6.8</td>
<td>Subcomponents of the bank application component</td>
<td>95</td>
</tr>
<tr>
<td>6.9</td>
<td>Subcomponents of the file server component</td>
<td>95</td>
</tr>
<tr>
<td>6.10</td>
<td>Subcomponents of the user interface component</td>
<td>96</td>
</tr>
<tr>
<td>6.11</td>
<td>Subcomponents of the RPC library component</td>
<td>96</td>
</tr>
<tr>
<td>6.12</td>
<td>Subcomponents of the file system component</td>
<td>97</td>
</tr>
<tr>
<td>6.13</td>
<td>Two different granularities of the same part of a composite work</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>model where complexity matrices are parameterised</td>
<td></td>
</tr>
<tr>
<td>7.1</td>
<td>Not all the required measurement data can be obtained</td>
<td>103</td>
</tr>
<tr>
<td>7.2</td>
<td>Interactions between software components</td>
<td>107</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

8.1 The IMSE architecture ...................................... 124
10.1 A complexity matrix which relates two work model components ... 135
10.2 A complexity matrix whose elements are complexity functions .... 136
10.3 Sets with varying number of set members ..................... 139
11.1 The initial work model of the client-server example ............. 145
11.2 The result of decomposition in the client-server example ....... 145
11.3 A composite work model of the client-server example ........... 146
11.4 The relationship between complexity matrices after a decomposition of a component-subcomponent relationship ............. 147
11.5 Relationships between work model components and measurable components ........................................ 149
11.6 Details in measurement of invocations between components .... 150
11.7 Two composite work models. The model in b) is a more detailed version of a). All components have just one operation ...... 156
11.8 The composite work model in b) is a more detailed version of a). 160
12.1 A minimal composite work model of the client-server example ... 164
12.2 A composite work model of the client-server example where the client and the server are modelled explicitly ............. 168
12.3 A composite work model of the client-server example where also the file system is modelled explicitly ................... 172
12.4 A composite work model of the client-server example where also the remote procedure call library is modelled explicitly .... 175
12.5 A composite work model of the client-server example where also the UNIX communication library (e.g., socket) is modelled explicitly . 176
12.6 The same component is used for representing the communication between the client and the server ............................... 177
12.7 Different RPC components are represented for the client–side and the server–side ............................................. 177
12.8 A minimal composite work model of the ICL system ............. 178
12.9 A composite work model of the ICL system where the application and the database server are modelled explicitly ............ 180
12.10 A composite work model of the ICL system where the application, the database server, and the local database management systems are modelled explicitly ....................................... 181
12.11 The communication software for each capsule in the system is modelled explicitly in the composite work model. In a) only one layer is introduced whereas in b) two layers are introduced .......... 182
12.12 The processing of each capsule in the system may be characterised in terms of high-level program statements .................. 184
12.13 The "final" SP model at ICL ................................ 185
12.14 A minimal composite work model of the hospital system ....... 187
12.15 A composite work model of the hospital system where the most significant components have been identified and modelled explicitly 189
12.16 Composite work model of the hospital system where artificial components are included in the model in order to tidy up. Note that this model does not obey the SP rules. .......................... 193
12.17 Composite work model of the hospital system from the IMSE project 194
12.18 Composite work model of the hospital system where the distribution is hidden within the CPU and the Disk components. ............... 194

13.1 A software component running on hardware platform A invokes a software component running on hardware platform B .................. 196
13.2 Different viewpoints during measurement (note the two "eyes" in the drawing). In a) the viewpoint is from within a process whereas in b) the viewpoint is between processes. .......................... 198
13.3 Transaction execution paths ............................................. 198
13.4 Distributed co-operating processes ...................................... 199

14.1 An SP model of the client-server system ................................. 206
14.2 An example of a borderline between static and dynamic modelling . 208
14.3 Activity diagram of a transaction ........................................ 209
14.4 A contention model of the client-server application which explicitly takes simultaneous resource possession into account ............... 210
14.5 A partial Petri net which represents a transaction .................... 210
14.6 A coarse contention model of the client-server application .......... 211
14.7 A software contention model and a device contention model for a client-server system (see Chapter 3.1) .............................. 212

15.1 A completeness test for a composite work model .................... 218
15.2 Work-oriented validation .................................................. 219
15.3 Performance-oriented validation ......................................... 219
List of Tables

6.1 Operations on the components in Figure 6.1 ........................................ 85
12.1 Example of inputs to WAT. ................................................................. 191
12.2 Average resource demands of the application operations obtained by cluster analysis ......................................................... 192
Bibliography


[42] Domenico Ferrari. *Computer Systems Performance Evaluation.* Prentice-Hall,

Workloads. In D. Ferrari and M. Spadoni, editors, *Experimental Computer

and Tuning of Computer Systems.* Prentice-Hall, Englewood Cliffs, N.J. 07632,
1983.

[45] Rich Friedrich. Requirements for the Performance Instrumentation of the DCE
RPC and CDS Services. Request for comments: 32.0, Open Software Founda-
tion (OSF), Special interest group (SIG) on the Distributed Computing
Environment (DCE), April 1993.


Hill, 1991.

[48] Dieter Haban and Dieter Wybranietz. A Hybrid Monitor for Behavior and
Performance Analysis of Distributed Systems. *IEEE Transactions on Software
Engineering*, 16(2):197–211, February 1990.

[49] Günther Haring. Fundamental Principles of Hierarchical Workload Descrip-
tion. In Giuseppe Serazzi, editor, *Collection of Invited Lectures presented at
the International Workshop on Workload Characterization of Computer Sys-

[50] P. Heidelberger and K. S. Trivedi. Queueing Network Models for Parallel Pro-
1108, 1982.

[51] P. Heidelberger and K. S. Trivedi. Analytic Queueing Models for Programs

[52] Philip Heidelberger and Stephen S. Lavenberg. Computer Performance Eval-
uation Methodology. *IEEE Transactions on Computers*, 33(12):1195–1220,
December 1984.


BIBLIOGRAPHY


