Software Design Metrics for Predicting Maintainability of Service-Oriented Software

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy

Mikhail Perepletchikov
B.App.Sc. (Hons.)

School of Computer Science and Information Technology
College of Science, Engineering and Health
RMIT University
Melbourne, Australia
February, 2009
Declaration

I certify that except where due acknowledgement has been made, the work is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program; any editorial work, paid or unpaid, carried out by a third party is acknowledged; and, ethics procedures and guidelines have been followed.

Mikhail Perepletchikov

School of Computer Science and Information Technology, RMIT University
Acknowledgements

First and foremost, I would like to thank my senior supervisor Dr Caspar Ryan for his unlimited support throughout the whole candidature. Dr Ryan helped me to stay on track during the difficult moments, always suggesting valuable research directions, and at the same time giving me enough freedom to pursue research areas of my interest. This was greatly appreciated.

I am very grateful to my second supervisor Dr Keith Frampton, whose vast industrial experience and knowledge in the software engineering and architecture areas was invaluable in the context of this software engineering related thesis.

Special thank you to Professor Zahir Tari who provided all necessary assistance as the head of my research discipline. More importantly, Prof Tari provided me with financial support throughout the candidature (as part of the research project funded by the Australian Research Council under Linkage scheme no. LP0455234).

I would like to offer my gratitude to the software practitioners, and post-graduate students from the School of Computer Science and Information Technology, RMIT University who participated in the empirical study conducted to evaluate the findings of this thesis.

Last but not least, thank you to my wife Svetlana and our children Anthony and Julia for their constant support, patience and understanding throughout my candidature.
Credits

Portions of the material in this thesis have previously appeared in the following publications:


# Contents

DECLARATION.................................................................................................................. II

ACKNOWLEDGEMENTS ................................................................................................. III

CREDITS........................................................................................................................... IV

CONTENTS......................................................................................................................... V

LIST OF FIGURES ........................................................................................................... XI

LIST OF TABLES ............................................................................................................... XII

ABSTRACT ......................................................................................................................... 1

CHAPTER 1. INTRODUCTION.............................................................................................. 2

1.1 RATIONALE ............................................................................................................. 2

1.1.1 The Significance of SOA ..................................................................................... 3

1.1.2 The Importance of Software Maintainability .................................................. 4

1.1.3 Measuring Structural Properties of Software Designs ................................... 5

1.2 RESEARCH QUESTIONS ......................................................................................... 6

1.3 RESEARCH METHODOLOGY ............................................................................... 7

1.3.1 Define Formal Model of Service-Oriented System Design ............................ 8

1.3.2 Derive Metrics ..................................................................................................... 9

1.3.3 Empirical Evaluation of Metrics ...................................................................... 10

1.3.4 Practical Applicability Analysis ....................................................................... 10

1.4 CONTRIBUTION ..................................................................................................... 11

1.4.1 Coupling and Cohesion Metrics ....................................................................... 11

1.4.2 Formal Model of SO Designs ............................................................................ 12

1.4.3 Initial Design Guidelines .................................................................................. 12

1.4.4 Summary of Contribution ................................................................................ 12

1.5 THESIS STRUCTURE ............................................................................................ 13
CHAPTER 2. LITERATURE REVIEW .................................................................14

2.1 OVERVIEW AND PURPOSE ........................................................................14

2.1.1 Review Structure ....................................................................................14

2.2 SERVICE-ORIENTED ARCHITECTURE (SOA) AND SERVICE-ORIENTED COMPUTING (SOC) 16

2.2.1 Service-Oriented Architecture (SOA) - Concepts and Definitions ..........16

2.2.1.1 Conceptual and Architectural Views of SOA ........................................18

2.2.1.2 Business Processes Modelling ............................................................20

2.2.2 Service-Oriented Computing (SOC) - Key Concepts and Definitions .......21

2.2.2.1 Development Strategies .....................................................................22

2.2.2.2 Existing Development Methodologies ................................................24

2.2.2.3 Service-Oriented Design Considerations ............................................25

2.2.3 Key Terms ...............................................................................................27

2.3 SOFTWARE PRODUCT QUALITY - MAINTAINABILITY .............................27

2.3.1 Quality Model - ISO/IEC 9126 ...............................................................29

2.3.2 Software Maintenance ............................................................................31

2.3.2.1 Measuring Maintainability .................................................................33

2.3.2.2 Predicting Maintainability .................................................................33

2.4 STRUCTURAL PROPERTIES OF SOFTWARE DESIGNS ...........................35

2.4.1 Overview ...............................................................................................35

2.4.2 Coupling ...............................................................................................38

2.4.3 Cohesion ...............................................................................................40

2.4.4 Discussion ............................................................................................42

2.5 SOFTWARE METRICS ............................................................................43

2.5.1 Key Concepts and Definitions ...............................................................44

2.5.2 Theoretical Basis and Validation Approaches .........................................47

2.5.2.1 Principles of Measurement Theory ....................................................48
2.5.2.2 Validation Approaches

2.5.3 Empirical Evaluation of Metrics

2.5.4 Existing Metrics

2.5.4.1 Coupling Metrics

2.5.4.2 Cohesion Metrics

2.5.5 Discussion

CHAPTER 3. FORMAL MODEL OF SERVICE-ORIENTED SYSTEM DESIGN

3.1 Modeling Software Designs

3.1.1 Related Work

3.2 Fundamental Characteristics of SO System Designs

3.3 Model Definitions

3.3.1 System Structure

3.3.2 Relationships

3.3.2.1 Service-Oriented Static Relationship Types

3.3.2.2 Service-Oriented Dynamic Relationship Types

3.3.3 Combined Structure and Relationships

3.3.4 Different Types of SO Systems

3.4 Definitions Listing

3.5 Discussion

CHAPTER 4. SERVICE-ORIENTED COUPLING METRICS

4.1 Overview

4.2 Coupling Assumptions and Metric Weights

4.3 Metrics Definitions

4.3.1 Metric Naming and Organisation

4.3.2 Primary Metrics

4.3.3 Aggregation Metrics
CHAPTER 5. SERVICE-ORIENTED COHESION METRICS

5.1 OVERVIEW

5.2 SERVICE-ORIENTED COHESION - DEFINITION AND TAXONOMY

5.2.1 Summary of the Cohesion Categories

5.3 METRICS DEFINITIONS

5.4 COHESION ASSUMPTIONS

5.5 SUMMARY

CHAPTER 6. EMPIRICAL EVALUATION OF METRICS

6.1 OVERVIEW

6.2 EXPERIMENTAL PROTOCOL

6.2.1 Goals

6.2.2 Participants

6.2.3 Dependent Variables

6.2.3.1 Maintainability Metrics

6.2.3.2 Subjective Cohesion Ranks

6.2.4 Hypotheses and Independent Variables

6.2.4.1 Scope of the Study (Independent Variables)

6.2.4.2 Coupling Hypotheses

6.2.4.3 Cohesion Hypotheses

6.2.5 Experimental Material

6.2.5.1 Coupling System (SYS-COUP)

6.2.5.2 Cohesion System (SYS-COH)

6.2.6 Experimental Tasks

6.2.6.1 Coupling (SYS-COUP) Related Tasks
6.2.6.2 Cohesion (SYS-COH) Related Tasks .................................................................................. 165
6.2.7 Experimental Design ........................................................................................................... 165
  6.2.7.1 Counterbalancing Approach .......................................................................................... 166
6.2.8 Experimental Procedure .................................................................................................... 167
  6.2.8.1 Pilot Study ................................................................................................................... 169
6.2.9 Data Collection Procedure .................................................................................................. 169
6.2.10 Analysis Procedure .......................................................................................................... 171
  6.2.10.1 Statistical Tests .......................................................................................................... 172
6.3 Analysis of the Results – Coupling Hypotheses .................................................................... 172
  6.3.1 Descriptive Statistics ........................................................................................................ 174
  6.3.2 Hypothesis Testing ........................................................................................................... 177
    6.3.2.1 Hypotheses $H_{coup1.1} \sim H_{coup1.6}$: Maintainability Impact .................................. 177
    6.3.2.2 Hypotheses $H_{coup2.1} \sim H_{coup2.6}$: Strength of Coupling Relationships .......... 178
6.4 Analysis of the Results – Cohesion Hypotheses .................................................................... 184
  6.4.1 Descriptive Statistics ........................................................................................................ 184
  6.4.2 Hypothesis Testing ........................................................................................................... 185
    6.4.2.1 Hypothesis $H_{coh1}$: Analysability Impact .............................................................. 186
    6.4.2.2 Hypothesis $H_{coh2}$: Cohesion Indication ............................................................... 186
  6.4.3 Linear Regression Analysis ............................................................................................... 187
    6.4.3.1 Hypothesis $H_{coh1}$: Analysability Impact .............................................................. 188
    6.4.3.2 Hypothesis $H_{coh2}$: Cohesion Indication ............................................................... 189
6.5 Summary ................................................................................................................................. 190
  6.5.1 Threats to Validity ............................................................................................................. 191
    6.5.1.1 Construct Validity .................................................................................................... 191
    6.5.1.2 Internal Validity ....................................................................................................... 191
    6.5.1.3 External Validity ..................................................................................................... 194
CHAPTER 7. SUMMARY AND CONCLUSION .................................................. 195

7.1 SUMMARY ....................................................................................... 195

7.2 PRACTICAL APPLICABILITY .......................................................... 197

7.2.1 Metrics Application in the SO Design Process .............................. 197

7.3 FUTURE WORK .............................................................................. 199

7.3.1 Metrics ......................................................................................... 199

7.3.2 Empirical Studies ......................................................................... 200

7.3.3 Tool Support for Metrics Collection .......................................... 202

APPENDIX A. LITERATURE REVIEW RESOURCES .................................. 205

APPENDIX B. MEASUREMENT THEORY CONCEPTS AND APPLICATION .... 207

APPENDIX C. MODEL NOTATION .......................................................... 209

APPENDIX D. USER PROFILE QUESTIONNAIRE AND PRE-TEST TASK DESCRIPTION .............................................................................................. 210

APPENDIX E. EXPERIMENTAL SOFTWARE SYSTEMS - DOCUMENTATION 213

1. AMS - SOFTWARE REQUIREMENTS SPECIFICATION (SRS) .................. 213

2. AMS (SYS-COUP) – UML CLASS DIAGRAMS ..................................... 218

3. AMS (SYS-COH) – UML CLASS DIAGRAMS ....................................... 222

APPENDIX F. EXPERIMENTAL TASKS ...................................................... 228

1. COUPLING (SYS-COUP) TASK .......................................................... 228

2. COHESION (SYS-COH) TASK ............................................................. 231

REFERENCES ....................................................................................... 234
List of Figures

FIGURE 1-1. RESEARCH ACTIVITIES .................................................................................................................. 8
FIGURE 2-1. LOGICAL VIEW OF SOA ................................................................................................................. 19
FIGURE 2-2. ARCHITECTURAL VIEW OF SOA (MODIFIED FROM [12]) .............................................................. 19
FIGURE 2-3. DESIGN VIEW OF SERVICE-ORIENTED SYSTEMS .................................................................... 22
FIGURE 2-5. BALANCE OF MAINTENANCE ACTIVITIES [10] ........................................................................ 32
FIGURE 2-6. A SIMPLE MAINTAINABILITY PREDICTION MODEL USED IN THIS RESEARCH ................ 34
FIGURE 2-7. MAINTAINABILITY MODEL USED IN THIS RESEARCH (UPDATED VERSION) ...................... 62
FIGURE 3-1. DESIGN OF A MODULAR SOFTWARE SYSTEM (MODIFIED FROM ([30], P.71)) .................. 65
FIGURE 3-2. EXAMPLE DESIGN REPRESENTING SO SYSTEM (SOS) ............................................................... 69
FIGURE 3-3. STATIC ALLOCATION OF IMPLEMENTATION ELEMENTS TO SERVICES .......................... 71
FIGURE 3-4. DYNAMIC ALLOCATION OF IMPLEMENTATION ELEMENTS TO SERVICES .......................... 71
FIGURE 3-5. DEFINITION OF A SERVICE-ORIENTED DESIGN RELATIONSHIP ......................................... 73
FIGURE 3-6. EXAMPLE SO DESIGN (INCLUDING DIFFERENT RELATIONSHIP TYPES) ............................ 76
FIGURE 3-7. EXAMPLE PARTITIONED SERVICE-ORIENTED SYSTEM (PARSOS) ......................................... 81
FIGURE 3-8. EXAMPLE PURE SERVICE-ORIENTED SYSTEM (PURSOS) ......................................................... 81
FIGURE 4-1. EXAMPLE SO DESIGN INCLUDING DIFFERENT RELATIONSHIP TYPES AS CAPTURED BY THE FORMAL MODEL OF SERVICE-ORIENTED SYSTEM DESIGN (MODIFIED FROM SECTION 3.3.2) ... 89
FIGURE 4-2. EXAMPLE SERVICE-ORIENTED SYSTEM DESIGN ................................................................. 94
FIGURE 5-1. GENERAL DEFINITION OF SERVICE COHESION ................................................................. 122
FIGURE 6-1. TASK-COUP1 - FAILURE ANALYSIS EFFICIENCY (FAE) ............................................................. 175
FIGURE 6-2. TASK-COUP2 - MODIFICATION COMPLEXITY (MC) – INCOMING COUPLING .................. 176
FIGURE 6-3. TASK-COUP2 - MODIFICATION COMPLEXITY (MC) – OUTGOING COUPLING .................. 177
FIGURE 6-4. GRAPHICAL REPRESENTATION OF THE REGRESSION RESULTS – TICS/FAE ......................... 188
FIGURE 6-5. GRAPHICAL REPRESENTATION OF THE REGRESSION RESULTS – TICS/CR .......................... 189
List of Tables

TABLE 1-1. THESIS CONTRIBUTION ........................................................................................................... 13
TABLE 2-1. RESEARCH AREAS COVERED IN THE LITERATURE REVIEW ............................................. 15
TABLE 2-2. SO TERMINOLOGY USED IN THIS THESIS ........................................................................... 27
TABLE 2-3. INFLUENCE OF THE STRUCTURAL PROPERTIES OF COUPLING AND COHESION ON SOFTWARE MAINTAINABILITY (S - STRONG; A - AVERAGE; W – WEAK; U - UNKNOWN)................ 43
TABLE 2-4. COUPLING AND COHESION PROPERTIES FROM PROPERTY-BASED SOFTWARE ENGINEERING MEASUREMENT FRAMEWORK ([30] P.76-79) ................................................................................ 52
TABLE 3-1. FORMAL MODEL OF SO SYSTEM - DEFINITIONS ................................................................. 83
TABLE 4-1. RELATIONSHIPS BETWEEN COUPLING ASSUMPTIONS AND SOFTWARE MAINTAINABILITY AND REUSABILITY ........................................................................................................... 92
TABLE 4-2. WEIGHTED SERVICE-ORIENTED DESIGN RELATIONSHIPS ............................................. 92
TABLE 4-3. COUP-M1: WEIGHTED INTRA-SERVICE COUPLING BETWEEN ELEMENTS (WISCE) .......... 96
TABLE 4-4. COUP-M2: SERVICE INTERFACE TO INTRA ELEMENT COUPLING (SIIEC) ....................... 98
TABLE 4-5. COUP-M3: EXTRA-SERVICE INCOMING COUPLING OF SERVICE INTERFACE (ESISI) ........ 99
TABLE 4-6. COUP-M4: ELEMENT TO EXTRA SERVICE INTERFACE OUTGOING COUPLING (ESIOC) ..100
TABLE 4-7. COUP-M5: WEIGHTED EXTRA-SERVICE INCOMING COUPLING OF AN ELEMENT (WESICE) .................................................................................................................................................. 102
TABLE 4-8. COUP-M6: WEIGHTED EXTRA-SERVICE OUTGOING COUPLING OF AN ELEMENT (WESOCE) ........................................................................................................................................... 103
TABLE 4-9. COUP-M7: NUMBER OF COUPLED INCOMING SERVICES (NCIS) ........................................ 105
TABLE 4-10. COUP-M8: NUMBER OF COUPLED OUTGOING SERVICES (NCOS) ................................. 106
TABLE 4-11. COUP-M9: SYSTEM PARTITIONING FACTOR (SPARF) ....................................................... 107
TABLE 4-12. COUP-M10: SYSTEM PURITY FACTOR (SPURF) ................................................................. 108
TABLE 4-13. COUP-M11: RESPONSE FOR OPERATION (RFO) .............................................................. 110
TABLE 4-14. COUP-AM1: TOTAL WEIGHTED INTRA-SERVICE COUPLING OF A SERVICE (TWISC) .... 112
TABLE 4-15. COUP-AM2: TOTAL WEIGHTED INCOMING EXTRA-SERVICE COUPLING OF A SERVICE (TWIESC) ....................................................................................................................... 112
TABLE 4-16. COUP-AM3: TOTAL WEIGHTED OUTGOING EXTRA-SERVICE COUPLING OF A SERVICE (TWOESC) ....................................................................................................................... 113
<table>
<thead>
<tr>
<th>Table No.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-12</td>
<td>DESCRIPTIVE STATISTICS: TASK-COUP2 (MODIFICATION COMPLEXITY (MC) VALUES - OUTGOING COUPLING)</td>
</tr>
<tr>
<td>6-13</td>
<td>PAIRED T-TEST RESULTS (TWO-TAILED P-VALUES) FOR HYPOTHESIS H_{COUP1.1} - H_{COUP1.6} (N=10)</td>
</tr>
<tr>
<td>6-14</td>
<td>ONE-WAY ANOVA RESULTS FOR MC VALUES (TASK-COUP2) – INCOMING COUPLING (N=30)</td>
</tr>
<tr>
<td>6-15</td>
<td>LSD GROUPS COMPARISON OF ONE-WAY ANOVA RESULTS FOR MC VALUES – INCOMING COUPLING (N=30)</td>
</tr>
<tr>
<td>6-16</td>
<td>ONE-WAY ANOVA RESULTS FOR FAE VALUES (TASK-COUP1) – OUTGOING COUPLING (N=30)</td>
</tr>
<tr>
<td>6-17</td>
<td>ONE-WAY ANOVA RESULTS FOR MC VALUES (TASK-COUP2) – OUTGOING COUPLING (N=30)</td>
</tr>
<tr>
<td>6-18</td>
<td>LSD GROUPS COMPARISON OF ONE-WAY ANOVA RESULTS FOR MC VALUES – OUTGOING COUPLING (N=30)</td>
</tr>
<tr>
<td>6-19</td>
<td>DESCRIPTIVE STATISTICS – FAILURE ANALYSIS EFFICIENCY (FAE) VALUES</td>
</tr>
<tr>
<td>6-20</td>
<td>DESCRIPTIVE STATISTICS - COHESION RANK (CR) VALUES</td>
</tr>
<tr>
<td>6-21</td>
<td>ONE-WAY ANOVA RESULTS FOR THE HYPOTHESIS H_{COH1} (N=50)</td>
</tr>
<tr>
<td>6-22</td>
<td>ONE-WAY ANOVA RESULTS FOR THE HYPOTHESIS H_{COH2} (N=50)</td>
</tr>
<tr>
<td>6-23</td>
<td>REGRESSION STATISTICS FOR TICS AND FAE VALUES (N = 5)</td>
</tr>
<tr>
<td>6-24</td>
<td>REGRESSION STATISTICS FOR TICS AND CR VALUES (N = 5)</td>
</tr>
<tr>
<td>6-25</td>
<td>SUMMARY OF THE EMPIRICAL EVALUATION RESULTS (LEGEND: STATISTICALLY SIGNIFICANT (√), NOT SIGNIFICANT (×), NOT RELATED (NR))</td>
</tr>
</tbody>
</table>
Abstract

As the pace of business change increases, service-oriented (SO) solutions should facilitate easier maintainability as underlying business logic and rules change. To date, little effort has been dedicated to considering how the structural properties of coupling and cohesion may impact on the maintainability of SO software products. Moreover, due to the unique design characteristics of Service-Oriented Computing (SOC), existing Procedural and Object-Oriented (OO) software metrics are not sufficient for the accurate measurement of service-oriented design structures.

This thesis makes a contribution to the field of SOC, and Software Engineering in general, by proposing and evaluating a suite of design-level coupling and cohesion metrics for predicting the maintainability of service-oriented software products early in the Software Development LifeCycle (SDLC). The proposed metrics can provide the following benefits: i) facilitate design decisions that could lead to the specification of quality SO designs that can be maintained more easily; ii) identify design problems that can potentially have a negative effect on the maintainability of existing service-oriented design structures; and iii) support more effective control of maintainability in the earlier stages of SDLC.

More specifically, the following research was conducted as part of this thesis:
- A formal mathematical model covering the structural and behavioural properties of service-oriented system design was specified.
- Software metrics were defined in a precise, unambiguous, and formal manner using the above model.
- The metrics were theoretically validated and empirically evaluated in order to determine the success of this thesis as follows:
  a. Theoretical validation was based on the property-based software engineering measurement framework. All the proposed metrics were deemed as theoretically valid.
  b. Empirical evaluation employed a controlled experimental study involving ten participants who performed a range of maintenance tasks on two SO systems developed (and measured using the proposed metrics) specifically for this study. The majority of the experimental outcomes compared favourably with our expectations and hypotheses. More specifically, the results indicated that most of the proposed metrics can be used to predict the maintainability of service-oriented software products early in the Software Development LifeCycle (SDLC), thereby providing evidence for the validity and potential usefulness of the derived metrics. Nevertheless, a broader range of industrial scale experiments and analyses are required to fully demonstrate the practical applicability of the metrics. This has been left to future work.
Chapter 1. Introduction

Service-Oriented Computing (SOC) is an emerging software development paradigm, which is based on the principle of encapsulating application and business logic within independent, reusable, and business-oriented software services. Presently, little research effort has been dedicated to considering how the structural properties of service-oriented software designs may influence the maintainability of final software products. More significantly, software metrics for measuring service-oriented design properties in an automated and objective manner do not exist.

This thesis makes a contribution to the field of SOC by presenting a suite of theoretically validated and empirically evaluated software metrics for measuring structural properties of coupling and cohesion of service-oriented designs. The metrics can be used as early predictors of the maintainability quality characteristic of service-oriented software systems. Maintaining software products is a resource-intensive process; therefore developing software that can be more easily maintained should be a key objective of any software engineering process. To this end, the derived metrics will allow identification and thus mitigation of potential maintainability problems early in the Software Development LifeCycle.

This introductory chapter serves four purposes. Firstly, Section 1.1 discusses the rationale behind this research. Secondly, Section 1.2 presents the research questions. The methodology followed in this research in order to answer the research questions is then described in Section 1.3. Finally, Section 1.4 summarises the contributions made by this thesis.

1.1 Rationale

Service-Oriented Computing (or Service-Oriented Architecture (SOA))\(^1\) has recently emerged as a major paradigm for developing software systems [64, 66, 186, 219]. Systems created within the SOC approach, that is Service-Oriented (SO) systems, aim to exhibit high flexibility and agility, facilitating rapid business changes and promoting software reuse [68, 187, 215].

The fundamental concepts of service-orientation have been described in the research and industry literature [12, 57, 118, 183] and software tools for assisting in the development of SO applications are becoming more widely used. Nonetheless, guidelines for designing high-quality service-oriented software that can be easily maintained are yet to be fully established,

\(^1\) Note that for the remainder of this thesis, the term SOC will be used to represent the development paradigm used to develop applications conforming to a specific type of system architecture, a Service-Oriented Architecture (SOA).
and formal mechanisms for assessing and estimating the maintainability of SO applications do not yet exist.

In previous paradigms such as Procedural and Object-Oriented (OO) development, it was shown that various quality characteristics (such as maintainability) can be predicted, and consequently improved, early in the Software Development Lifecycle (SDLC) by examining the structural properties of software designs, such as coupling and cohesion [35, 45, 76]. To effectively quantify these properties, a number of software metrics were proposed and employed to assist in the identification of the design problems and early prediction of software quality attributes [32, 44, 53, 146]. Early prediction of maintainability is of utmost importance given that the maintenance activities are resource intensive; therefore, it is crucial to identify and fix the potential problems as early as possible.

At present, little research effort has been dedicated to considering how the coupling between services and cohesiveness of individual services in service-oriented systems may impact on the maintainability of software products. Moreover, due to the unique characteristics of SOC described in Section 2.2, the existing Procedural and OO metrics are not sufficient for the accurate measurement of the service-oriented design structures [189]. Therefore, this thesis formally defines, and theoretically and empirically evaluates a suite of SOA-specific design-level metrics. The metrics support rigorous assessment of structural properties (coupling and cohesion) of service-oriented design artefacts, thereby assisting in the detection of the design deficiencies and facilitating prediction of maintainability early in the development process.

1.1.1 The Significance of SOA

Enterprise information systems are becoming increasingly large and complex requiring more precise mechanisms for managing software complexity and, more importantly, meeting the demands of highly-dynamic business environments. In order to efficiently support these objectives, the SOC paradigm [12, 64, 66, 186, 230] was introduced as an extension to the existing development approaches (such as Procedural and OO development).

SOC provides a flexible and agile development model by introducing an additional layer of software abstraction – a service layer. Service-oriented applications are structured as a collection of independent, business-aligned software services, which can be composed into executable business processes. The business processes encapsulate business logic and rules, separating them from the software implementation of services, thus promoting higher reusability of the individual services and facilitating rapid propagation of business changes and reduction of maintenance efforts [18, 220, 241].

Note that the key terms and concepts related to this research will be highlighted in italic font throughout the remainder of the thesis.
SOA is becoming an increasingly popular choice of system architecture. For example, according to a market survey from Forrester Research [96]:

- 21% of North American and European (NA-EU) enterprises said that they plan to adopt SOA by the end of 2007, which should bring an overall SOA penetration in NA-EU enterprise markets to 62%;

- 22% of Asia-Pacific enterprises and 14% of NA-EU small-and-medium businesses planned to adopt SOA in 2007, bringing total projected penetration in these markets to 59% and 40% respectively.

The recently released follow-up report indicates that the above adoption targets have been largely met by the companies participating in the survey in terms of transforming the underlying IT infrastructures into well-planned service-oriented solutions, and that SOA will continue its strong market momentum in 2008 and beyond [97].

Additionally, this rapid uptake of SOA has been strongly supported by major software vendors who offer a number of service-oriented middleware platforms and development environments and tools. Moreover, the Object Management Group (OMG) has recently set up ‘The SOA Consortium’ [227] with the support from IBM, Sun, Cisco, SAP, and SoftwareAG in order to achieve the following objectives by 2010: i) 75% of the Global 1000 companies self-proclaim SOA Success; ii) 75% of Major Government Agencies self-proclaim SOA Success; iii) 50% of mid-size businesses self-proclaim SOA Success; where SOA Success is defined in terms of value generation, and increase in business and IT agility [227].

To summarise, SOC is becoming an important software development paradigm, shifting focus from monolithic software to composite applications consisting of autonomous, and reusable and maintainable software services that can be easily composed into executable business processes [52, 184, 241]. The key concepts of SOC will be described in greater detail in Section 2.2.

1.1.2 The Importance of Software Maintainability

Developing quality software should be the key target of any software engineering process, with software maintainability being one of the most important quality characteristics, representing the capability of the software product to be modified [231]. According to the ISO/IEC 9126-1:2001 standard, software maintainability can be subdivided into four sub-characteristics: analysability, changeability, stability, and testability [111]. These sub-characteristics can be directly measured using standardised metrics prescribed by ISO/IEC [112, 113].

The Software Development LifeCycle (SDLC) consists of a number of typically iterative and interleaving development phases [132]. One of these phases is software maintenance,
which is resource intensive given that the bulk of the project effort is consumed by the continuous perfection, correction, and adaptation of existing software resources [135, 147]. Although the reported numbers vary, it has been estimated by various researchers that the maintenance phase of the SDLC consumes more than 60% of the overall project resources [49, 103, 123, 205]. Therefore, developing software that is difficult to maintain can contribute to project failures due to the cost and time overruns [123, 231].

More importantly, creating highly maintainable software is especially crucial for an emerging generation of constantly-evolving service-oriented enterprise applications. As the pace of business change increases due to globalisation and e-commerce, SOA-based systems should be able to rapidly adapt to customers’ needs by seamlessly integrating changes to the underlying business logic and rules [173]. This can be more readily achieved when the software is highly maintainable. Moreover, the time needed to complete software maintenance activities can play a major role when determining the capability of enterprises to adjust to changing market conditions and to implement innovative products and services in order to stay competitive. At present, given that service-oriented solutions are typically new and are yet to undergo major software changes, it is not clear whether the desired behaviour will be exhibited when modifications are made. Maintainability is discussed further in Section 2.3.

1.1.3 Measuring Structural Properties of Software Designs

The maintainability of any software product can only be directly measured when the product has been developed and released, and subsequent changes are made. Although assessing the maintainability of the finished products will result in the most precise measurements, this approach has a considerable disadvantage since any discovered problems will be more costly to fix at the post-production stage [135, 231].

Therefore, various research initiatives have been focused on establishing predictive models that support estimation of software maintainability early in the SDLC [49, 177]. Estimating the maintainability of software prior to its release could result in the loss of measurement accuracy and is potentially a tedious task to perform. Nonetheless, such early estimation can decrease the cost of fixing any potential problems given that the preventive and corrective actions can be performed more efficiently during the earlier stages of development [54].

One of the key factors in these predictive models is the structure of software as represented by its structural design properties (refer to Section 2.4), namely size, complexity, coupling, and cohesion [1, 7, 146]. Consequently, a large number of metrics have been proposed for measuring the structural properties of designs in a quantitative and automated manner [34, 44, 98, 102]. The existing structural metrics were defined for software systems developed using the OO [33, 43, 44, 98, 146] or Procedural [72, 156] development approaches; therefore, they are not necessary applicable to the key principles of SOC as described in Section 2.5.
Previous studies indicate that structural **coupling** and **cohesion** (measured using various metrics) can have a strong causal impact on maintainability [5, 35, 79, 85, 86, 214]. Consequently, there is a need to define a suite of metrics for measuring **coupling** and **cohesion** of service-oriented software designs. Such metrics can provide the following benefits:

- Identify problems in existing service-oriented design structures;
- Justify key trade-offs in design decisions;
- Allow for more effective control of maintainability [1, 7].

Moreover, such metrics can provide a foundation for a comprehensive design methodology. This is because the metrics will encapsulate key principles of service-oriented design, thereby providing support for eliciting initial service-oriented design guidelines and rules.

Note that software designs also exhibit additional structural properties that could influence the maintainability of software, such as complexity and size. These properties are not investigated in this work since the decision was made to focus on the properties that were deemed to be most important, based on the analysis of the problem domain and measurement objectives as explained in Section 2.4.1. In brief:

i) design-level complexity can be viewed as the combination of coupling and cohesion [55]; therefore, the proposed metrics can be adapted to indirectly measure complexity;

ii) the size of software is not dependent on any particular development paradigm. As a result, existing metrics (such as SLOC/LOC [76] or FPA [224]) can be readily used to measure the size of service-oriented software.

### 1.2 Research Questions

The primary goal of this research is to derive a suite of software metrics for quantifying the structural properties of coupling and cohesion of SO designs in order to predict software maintainability. In doing so, this thesis addresses the following five research questions:

1. **What are the distinguishing characteristics of SO designs?**

   This is answered in Chapters 2 and 3 of this thesis, with fundamental characteristics of service-oriented software being identified and documented in Chapter 2, and then formally captured by the model presented in Chapter 3.

2. **Can existing Procedural and OO metrics correctly measure structural properties (such as coupling and cohesion) of SO designs?**

   Answered by the findings of a case-study that empirically evaluated the applicability of existing metrics to service-oriented designs. The results of the case-study, summarised in Section 2.5.5, indicated that the existing metrics are not sufficient for SO designs.
CHAPTER 1. INTRODUCTION

3. **Which metrics should be used to measure coupling and cohesion in SO designs?**

A new suite of metrics for measuring structural properties of coupling and cohesion in service-oriented designs is formally defined and theoretically validated in Chapter 4 (coupling metrics) and Chapter 5 (cohesion metrics).

4. **Can measures of design-level coupling and cohesion be used as useful predictors of maintainability of SO software products?**

This is answered in Chapter 6 where the derived metrics have been evaluated empirically in order to statistically test the correlation between the derived measures of coupling and cohesion, and the maintainability of service-oriented software products.

5. **Can measurement of service-oriented design coupling and cohesion be conducted in an automated manner?**

This is discussed in Chapter 7 where the derived metrics are shown to fulfil the desirable pragmatic properties since they are technology independent and can be collected in an automated manner using a dedicated software tool. Note that developing a metric collection tool was considered to be outside of the research scope and is part of future work.

1.3 Research Methodology

This section presents the overall methodology followed in this research in order to derive and theoretically and empirically evaluate a suite of SO design metrics, thereby answering the research questions defined in the previous section.

A critical analysis and comprehensive review of existing work in the areas of SOC, software maintainability, and software metrics was conducted in order to gain knowledge and expertise required to effectively perform research activities described in this section. The results of this analysis/review (Chapter 2) contribute to answering Research Question 1.

Furthermore, an initial case-study has been conducted in order to empirically determine whether some of the widely-used Procedural and OO metrics can correctly measure the structure of service-oriented designs. The study, presented in [190], demonstrated that the metrics under investigation cannot quantitatively distinguish between SO designs that were considered qualitatively different, thus providing an answer to Research Question 2.

The actual metric derivation process uses the approaches proposed by Shepperd and Ince [218] and Briand et al. [37]. Such approaches provide systematic guidance for the metric derivation process, thereby insuring that the metrics conform to the following widely-accepted validity criteria [71, 213]: i) represent accurately the entities and attributes they purport to quantify; ii) possess a ‘face value’; iii) be practically applicable.
CHAPTER 1. INTRODUCTION

The metrics derivation process is outlined in Figure 1-1 together with the corresponding thesis chapters and research questions, with each research activity described in greater detail in the following sub-sections.

1.3.1 Define Formal Model of Service-Oriented System Design

As a prerequisite to the measurement of any software property, it is necessary to formally model the entity under investigation (service-oriented design’), thereby establishing a mechanism for defining metrics in an unambiguous and formal manner making sure that the derived metrics accurately represent the entities and attributes they purport to quantify.

A formal model of service-oriented design will capture an understanding of the core design principles and characteristics of SOC, as elicited through: i) detailed critical review of previous work; ii) informal face-to-face or correspondence-based discussions with experts in the area; and iii) skills and development experience of the present author. The model will also indirectly assist in answering Research Question 1.

---

Note that the terminology used in this thesis to describe different concepts of service-orientation (including ‘service-oriented design’ and ‘SOC’) can be found in Section 2.2.3 – Table 2-2.
1.3.2 Derive Metrics

The overall metrics derivation process consists of the following steps:

1. **Establish measurement goals.**

   The measurement goals should be defined in line with the research objectives, with the “Goal/Question/Metric” (GQM) method [19], being commonly used as a guide for defining goals along the following dimensions:
   
i) **Object of study** that defines the entities and attributes under investigation;
   ii) **Purpose of measurement** that shows the potential use of the metrics;
   iii) **Quality focus** that assists in selecting the dependent attributes used in the assumptions and experimental hypotheses;
   iv) A **viewpoint** that specifies who is affected by the results of measurements;
   v) A description of the **environment** that provides context of the obtained results.

   For example, the key measurement goal of this research can be formulated according to the above template as:
   
i) Analyse the coupling and cohesion of service-oriented designs,
   ii) for the purpose of evaluation and prediction,
   iii) with respect to software maintainability,
   iv) from the point of view of software engineers,
   v) in the context of experimental SO software systems.

2. **Establish informal assumptions and experimental hypotheses.**

   The assumptions assist in the metrics derivation and validation process by establishing informal connections between structural design properties of coupling and cohesion, and software maintainability as captured by its sub-characteristics (analysability, changeability, stability, and testability) based on an understanding of the problem domain and the review of existing literature. Note that the sub-characteristics of maintainability are discussed in detail in Section 2.3.1. Furthermore, the formally redefined assumptions will serve as experimental hypotheses to be tested during the empirical evaluation of metrics (in Chapter 6).

3. **Metrics definition and theoretical validation.**

   Rigorous and precise mathematical notations and techniques should be used during the derivation and subsequent theoretical validation of the metrics. To this end, the definitions captured by the formal model of service-oriented software design will allow defining metrics in a precise, unambiguous, and formal manner. Moreover, the evaluation of the completeness
of the proposed metrics can be performed based on the coverage of structural and behavioural aspects captured by the model.

Additionally, in order to derive metrics that are valid from the measurement theory perspective, it is important to clearly specify the following characteristics [111]:

i) Metric type (direct/indirect) - the metrics derived in this research are direct measures since they do not involve other design properties in their measurement activities.

ii) Metric scale - the derived metrics are defined on ratio and absolute scale, which are the most informative type of measurement scale.

iii) Measurement unit – the derived metrics use count as the measurement unit.

Note that as with the formal model of SO design, the metrics are based on an intuitive understanding of the core principles of SOC, thereby insuring that the derived metrics possess a 'face value'. Finally, it is important to demonstrate the theoretical validity of metrics. Therefore, the derived metrics were theoretically validated using the property-based software engineering measurement framework [30] described in Section 2.5.2.

The above-described metric derivation process will allow answering Research Question 3.

1.3.3 Empirical Evaluation of Metrics

Empirical evaluation shows the usefulness of metrics in practice, thus being the crucial activity in establishing the overall validity of a given metric. The empirical evaluation of the metrics derived in this thesis involves experienced software engineers and post-graduate students performing maintenance tasks on two service-oriented systems exhibiting different structural properties. Statistical methods are then used to test the correlation between design coupling and cohesion, as measured by the derived metrics and encapsulated by the experimental hypotheses, and maintenance efforts measured using existing ISO/IEC metrics [112, 113]. Note that established experimental techniques for collecting data and analysing the results are used during the empirical evaluation. For example in this thesis, the correlation and regression analysis techniques were used since they provide a robust method suitable for exploratory research [34]. The empirical evaluation addresses Research Question 4.

1.3.4 Practical Applicability Analysis

It is not enough to simply validate metrics theoretically and empirically, the metrics should also be practically applicable. To be useful in real projects the metrics should exhibit the following pragmatic characteristics [98]:

1) The metrics can be easily collected in an automated manner – otherwise it will be difficult to efficiently apply metrics to large-scale projects.
2) The metrics should be technology independent – otherwise they will have a limited scope of use, and comparison across products developed using different technologies will be difficult.

3) The metrics should be integrated into the software process to support the decision making during the design and implementation phases of SDLC.

Applicability analysis will allow answering Research Question 5.

1.4 Contribution

This section outlines the contributions of this thesis, with the main contribution being a suite of software metrics for measuring structural properties of service-oriented design artefacts as described in Section 1.4.1. Secondary contribution is a formal model covering the structural and behavioural properties of service-oriented system designs as described in Section 1.4.2. Additionally, the metrics and a formal model can lay a foundation for the derivation of SOA-specific design methodology as briefly discussed in Section 1.4.3. Finally, the summary of the contribution is shown in Section 1.4.4.

1.4.1 Coupling and Cohesion Metrics

The main contribution of this thesis is the derivation of a suite of design-level metrics for measuring coupling [191, 194] and cohesion [192] in service-oriented systems (Chapters 4 and 5 respectively). The metrics can be used as early predictors of quality characteristics of service-oriented software, with this work being particularly concerned with the quality characteristic of maintainability, thus allowing organisations to identify potential quality problems in the early stages of the SDLC.

The proposed metrics are theoretically valid since they are shown to exhibit mathematical properties of coupling and cohesion as defined in the property-based software engineering measurement framework of Briand et al. [30]. More importantly, the metrics have been evaluated empirically and the results indicate a correlation between the coupling and cohesion of service-oriented designs (as measured by the metrics) and the maintenance efforts. The empirical evaluation consisted of a number of experiments, where participants were asked to perform maintenance activities on two software systems that exhibited different structural characteristics as reflected by the metrics. The relationship between the coupling and cohesion metrics, and measures of maintainability was then analysed, showing statistically significant correlation for a number of the metrics derived in this research (as described further in Chapter 6). Therefore, we can conclude that the derived metrics can be considered as theoretically valid and potentially useful predictors of software maintainability.
1.4.2 Formal Model of SO Designs

The secondary contribution of this thesis is the definition of a formal mathematical model covering the structural and behavioural properties of service-oriented system design [193, 195] (Chapter 3). This model captures the design structure of service-oriented systems as a bi-directional graph expressed using set-theoretic notation [80]. Vertices in the graph symbolise design artefacts representing logical and physical software entities found in service-oriented systems. Edges correspond to the relationships between these design artefacts, representing both structural and behavioural dependencies.

There are two major benefits of this model. Firstly, the model formalises the fundamental design concepts of SOC, thus supporting a better understanding of the issues related to service-oriented development. Secondly, and more importantly in the context of this thesis, the model provides means for defining and theoretically validating software metrics in a precise, unambiguous, formal manner.

Note that the proposed model was designed to be as generic and technology agnostic as possible in order to facilitate wide applicability. Nonetheless, the model can be readily specialised to cover the constraints imposed by a specific implementation technology [193].

1.4.3 Initial Design Guidelines

Although it is not one of the immediate goals of this research, the derived metrics can lay a foundation for a service-oriented design methodology by providing means of identifying initial design-level guidelines and patterns. For example, specific design guidelines can be formulated in terms of concrete metric values. Additionally, the proposed formal model enforces constraints on the overall design structure and possible relationships between design artefacts, thus providing means to evaluate the conformance of a given system design to the fundamental characteristics of SOC. Such constraints should be captured by the development methodology. Note that the derivation of the complete service-oriented design methodology is beyond scope of this thesis and is part of future work.

1.4.4 Summary of Contribution

<table>
<thead>
<tr>
<th>CONTRIBUTION</th>
<th>ACADEMIC BENEFITS</th>
<th>INDUSTRY BENEFITS</th>
</tr>
</thead>
<tbody>
<tr>
<td>METRICS</td>
<td>- Extending the concepts of coupling and cohesion for service-oriented (SO) software systems. - Replicating (and specialising) the repeatable process of derivat...</td>
<td>- Allowing comparison and selection of alternative SO design structures, and supporting justification of key trade-off design decisions in SOC.</td>
</tr>
</tbody>
</table>
### CHAPTER 1. INTRODUCTION

| **MODEL** | - Promoting a better understanding of the SOC paradigm, by encapsulating the major structural and behavioural design properties of SO software.  
- Demonstrating the process of extending the generic model of a software design [30] for a specific paradigm (SOC).  
- Providing means for defining and theoretically validating service-oriented design metrics in a precise and formal manner. |
| --- | --- |
| | - The model can provide a theoretical support for various software tools. More specifically, the model can be used to support:  
1) automated design consistency checks and metric collection;  
2) creation of architectural and design diagrams using graphical representation (the specification of which is part of future work) of the artefacts and relationships captured by the model. |

| **Table 1-1. Thesis Contribution** |

| **1.5 Thesis Structure** |

The remainder of this thesis is separated into six chapters. Chapter 2 reviews and critically analyses existing work in the areas related to this research, thereby providing a foundation for the remaining chapters. Chapters 3-6 cover the metrics derivation process and associated activities (as shown in Figure 1-1). More specifically, Chapter 3 presents a formal model of SO software designs, which in turn provides the formalism for the definition of coupling and cohesion metrics in Chapters 4 and 5. The metrics are then empirically evaluated in Chapter 6. Finally, Chapter 7 presents concluding remarks (including the analysis of the practical applicability of the derived metrics) and outlines some future research directions.

(February, 2009)
Chapter 2. Literature Review

This chapter reviews and analyses existing work in the areas of Service-Oriented Architecture and Computing, software maintainability, and software design properties and metrics. The review provides the necessary background for the work presented in this thesis, and assists in formulating answers for the research questions defined in the previous chapter.

2.1 Overview and Purpose

The review process was based on the guidelines proposed by Kitchenham [127] and evaluated by Brereton et al. [26], which incorporate the procedures for performing a systematic literature review in the context of software engineering. The major review activities, the search strategy employed for selecting appropriate review material, and the specific review sources are described in Appendix A.

The purpose of this literature review is twofold. Firstly, it was designed to investigate various research topics related to this thesis in order to provide answers to Research Questions Q1 and Q2 described in Section 1.2. Secondly, it provides background needed to understand the intended contribution of this research (namely a suite of software metrics for measuring coupling and cohesion of service-oriented designs), thereby indirectly assisting in answering Research Questions Q3, Q4, and Q5 (Section 1.2).

More specifically, a number of research topics have been identified and reviewed. These topics are listed in Table 2-1 together with the corresponding section numbers, with the major research areas highlighted in bold font. Note that the order of the presentation of research topics is not consistent with the order of corresponding research questions since all effort was made to produce a structurally sound chapter where the topics are grouped into sections based on their conceptual relevance. Additionally, some of the reviewed topics are presented in multiple chapters in order to improve the readability of the thesis, and also make it easier to compare the contribution of this research to that of the existing work.

2.1.1 Review Structure

The literature review is documented in four separate sections. Each section comprises a grouping of related research topics (from Table 2-1) as follows: Section 2.2 discusses the fundamental characteristics and design principles of Service-Oriented Computing (SOC) and Service-Oriented Architecture (SOA). Section 2.3 overviews the areas of software quality in general and software maintainability in particular. Section 2.4 examines the structural properties of software designs. Finally, Section 2.5 describes the area of software metrics.

(February, 2009)
<table>
<thead>
<tr>
<th>RESEARCH QUESTION</th>
<th>TOPICS OF INTEREST</th>
<th>SECTION NUMBER</th>
</tr>
</thead>
</table>
| **Q1** | ▪ SOC (and SOA), including:  
  o key concepts and definitions  
  o conceptual and architectural structure  
  o technological aspects (for example, web services and Business Process Modelling (BPM) approaches)  
  o software engineering principles  
  ▪ development strategies and methodologies  
  ▪ design principles and characteristics | 2.2  
  2.2.1-2.2.3  
  2.2.1  
  2.2.1  
  2.2.2  
  2.2.2 |
| **Q2** | ▪ software design properties in general  
  o design properties of coupling and cohesion  
  ▪ software metrics in general  
  o existing metrics for measuring coupling and cohesion | 2.4, 2.4.1  
  2.4.2, 2.4.3  
  2.5, 2.5.1  
  2.5.4, 4.1, 5.1 |
| **Q3** | ▪ measurement theory, and metrics derivation and theoretical validation approaches  
  ▪ formal models of software | 2.5.2  
  2.5.2, 3.1 |
| **Q4** | ▪ software product quality in general  
  ▪ quality characteristic of maintainability and its various sub-characteristics  
  ▪ maintainability prediction factors  
  ▪ empirical validation of metrics | 2.3  
  2.3.1, 2.3.2  
  2.3.3  
  2.5.3, 6.1 |
| **Q5** | ▪ practical applicability and tool support | 7.2 |

*Table 2-1. Research areas covered in the literature review*
CHAPTER 2. LITERATURE REVIEW

2.2 Service-Oriented Architecture (SOA) and Service-Oriented Computing (SOC)

This section describes the fundamental characteristics of Service-Oriented Architecture (SOA) and Service-Oriented Computing (SOC). Additionally, given that the main focus of this research is the design of service-oriented (SO) systems (in particular its structural properties); this section reviews and discusses the major principles of SO design. It is important to note at this stage that the concepts of SOA and SOC are relatively new and the related research is still in its infancy, lagging behind the industry initiatives in the area [104]. Although all effort was made to produce an objective and well-supported overview of SOA and SOC, some of the presented concepts and definitions are based on the experience of the present author and on informal discussions with the software practitioners and researchers in the area [60, 99, 141, 182, 206, 248].

Note that the terms SOA and SOC are often used interchangeably in the existing literature. In this thesis, SOA and SOC are treated as related, but at the same time distinct concepts as reflected by the structure of this section, where SOA and SOC are described independently from one another. More specifically, SOA represents an abstract high-level architectural model that covers all aspects of provisioning, consumption, and management of software services (or systems comprised of such services) [159]; whereas SOC is the development paradigm used to analyse, design and implement the individual SO systems that can be integrated into SOA.

2.2.1 Service-Oriented Architecture (SOA) - Concepts and Definitions

Service-Oriented Architecture (SOA) represents an abstract model of system architecture that employs business-aligned software services, which can be composed and orchestrated using executable business processes to fulfil a specific domain or business requirement. Papazoglou et al. define services as “autonomous, platform-independent entities that can be described, published, discovered, and loosely coupled in novel ways” [184, p. 64]. Services in SOA are commonly treated as ‘black-boxes’ from the architectural perspective, where the corresponding service interfaces constitute the only visible part to the rest of the architecture [67].

Numerous definitions of SOA have been proposed in the research and industry literature, including: i) a business-centric architectural approach enabling organisations to integrate systems and processes as repeatable services [11]; ii) a consistent approach for defining services in the IT systems that align with business functions and processes [173]; iii) a logical way of designing software systems to provide services to end-user applications or other services distributed in a network [184]; and iv) an architectural model that aims to enhance the agility
and productivity of an enterprise by using services as the primary means through which solution logic is represented [68]. Moreover, the notion of SOA has been extended in recent publications to cover the specific application domains. For example, Woods and Mattern [241] introduce the concept of ESA (Enterprise Services Architecture), which is the framework for enabling easier evolution of IT resources using the combination of business semantics and core principles of SOA.

It has been suggested that SOA can provide a number of advantages over the other architectural models in terms of the reusability, business agility, and interoperability of the produced software [18, 66, 130, 173, 208]. Such characteristics constitute the fundamentals of SOA.

Services in SOA are highly reusable because they are independent self-contained entities that do not depend on the state or context of other services in the system, and thus can be reused in the context not known at the design time. Additionally, services are typically composed into business processes represented in terms of business concepts rather than system level implementation details [245]. Such processes can be designed by business analysts with the aid of software tool support and then transformed into executable modules or business process scripts, which are deployed and executed using middleware. Encapsulating business logic and rules in the business processes, thereby separating them from the actual software implementation, promotes reusability and increases the business agility of software. Moreover, the business processes can be easily modified by business analysts without a need for implementation-level changes, again increasing the business agility of software and facilitating rapid business change and reduction of maintenance efforts [18, 187, 241].

Interoperability is supported by the technological aspects of SOA. At present, services in SOA are typically implemented as platform-independent Web Services that communicate via XML-based SOAP protocol and are described using WSDL 1.1 (or recently standardised WSDL 2.0) interfaces [4, 94]. This allows for seamless interoperability between different platforms and programming languages. To this end, SOA treats individual software systems as independent services geared for integration, and uses them to build agile networks of collaborating service applications.

Note however that the implementation of Web Services is not restricted to the SOAP stack of protocols given that SOA is technology agnostic. For example, Richardson and Ruby [203] suggested recently that Web Services can benefit from the RESTful implementations on top of HTTP, in which services are defined in a resource-oriented fashion instead of a more conventional function-oriented manner. The process of modelling applications as a collection of RESTful services is simpler than SOAP-based Web Services because the number

---

*Representational State Transfer (REST) is the generic architectural style for modelling web-based applications and resources. According to Fielding, the foundation of REST is said to be directly interleaved with that of the Web itself [78].*
CHAPTER 2. LITERATURE REVIEW

of architectural decisions associated with REST is relatively smaller (as reflected by the smaller number of alternative technologies and standards) [188]. Nevertheless, the core design elements of REST are not readily suited for the process of service composition, due to the lack of conceptual and technological support for the integration with the current Business Process Modelling approaches (refer to Section 2.2.1.2) that constitute an integral part of SOA [188].

Given the desire to conduct worldwide business and other collaboration in a uniform interoperable manner, as well as the need to efficiently compose, leverage and reuse existing business resources, Service-Oriented Architecture utilising web services and various WS-* specifications [236] appears to be a highly suitable solution. Note that it is not an objective of this thesis to champion SOA; indeed this is not needed since SOA is already a popular architectural choice, with many organisations having adopted (or are planning to adopt) SOA [96].

2.2.1.1 Conceptual and Architectural Views of SOA

At a conceptual (or logical) level, SOA consists of three primary components: i) service providers, who publish service descriptions and realise software services; ii) service consumers, who discover a service description, and invoke a service; and iii) service registries or repositories (such as UDDI [246]) that maintain a directory of services to be discovered by the consumers [64, 67]. This high-level conceptual model is illustrated in Figure 2-1.

Additionally, the conceptual model of SOA introduces another fundamental characteristic of service-orientation – loose-coupling. This is because the service consumers and providers are separated from one another via service registries, meaning that there is no need for explicit relationships between both parties. That is, the service consumers can select (or discover dynamically at run-time [173]) required services from the registry without depending on a particular service provider. Moreover, one of the primary motivations for using Web Services, is that they are accessed through language and location independent interfaces, which also promotes loose-coupling from the integration perspective [4].

Note however that it is possible to design SOAP-based Web Services in a tightly-coupled manner. For example, Web Services Resource Framework (OASIS WS-RF), which has been recently standardised by OASIS [175], defines specification for modelling and accessing stateful resources using Web services. Specifically, OASIS WS-RF provides support for the management of application state through properties associated with Web Services. Such ‘forceful’ injection of a state into Web Services, which are meant to be stateless according to the core principles of SOA [68], could result in tightly-coupled applications.
The architectural-level view of SOA is shown in Figure 2-2. At the architectural level, SOA implementations consist of: i) the major services (1) in the system (the design and implementation of which is treated as “black-box” at the architectural level); ii) associated business processes (2) that are used to compose\(^6\) individual services in order to provide extended functionality to the consumers (3); and iii) various integration (4) and management–related (5) aspects.

---

\(^5\) This high-level logical structure is commonly referred to as ‘find-bind-execute’ model [104].

\(^6\) The compositional aspect introduced by business processes is another fundamental characteristic of SOA [241]
The business processes themselves are typically exposed using standardised service interfaces (for example, WSDL-based interfaces [4, 94]), and as such can be included in service registries as common services.

The integration architecture is typically covered by the Enterprise Service Bus (ESB) [40] implementations that provide middleware-level support for the integration of disperse service-oriented applications in terms of message and event-based interactions, and seamless data integration [161]. From the architectural perspective, an ESB provides an abstraction layer on the top of existing enterprise messaging systems in order to minimise direct dependencies between the provided services and their potential consumers. Moreover, an ESB can provide support for business process choreography and orchestration implementations [40] described in the following sub-section.

The management-related aspects, such as service monitoring and Quality of Service (QoS) enforcement, are also supported by the middleware-based solutions or dedicated software components (for example, intelligent agents [219]).

### 2.2.1.2 Business Processes Modelling

Business processes reflect workflows within and between organisations. Business process modelling (BPM) describes activities that interact with various intra/inter organisational elements while supporting the operation of the business [187]. Specifically, the purpose of business process modelling is to provide a mechanism for composing software services together in order to provide some well-defined business functionality. This includes two distinct compositional approaches: orchestration and choreography [17].

Orchestration specifications incorporate a local view of the business interactions, where one centralised business process entity controls the flow of the process execution, and invocation of the required services. In contrast, choreography specifications capture the global perspective of the business interactions across different enterprises or organisational divisions without imposing the need for a centralised control insofar as each participant in a choreography interacts with other participants via peer-to-peer message exchanges on [200]. It is important to note that the orchestration specifications can be directly mapped to the executable business process scripts, whereas the choreography specifications are not directly executable since they are designed to capture the overall high-level messaging behaviour and associated business rules of a workflow without considering low-level details such as the specific format of message exchanges.

There are a large number of techniques proposed for business process modelling ranging from flow charts to UML and Petri Nets, each having various supporting business process languages. Such languages allow business process models to be designed, and in the case of orchestrations, directly executed via middleware support. For example:
CHAPTER 2. LITERATURE REVIEW

- Business Process Modeling Notation (BPMN 1.2) [238] is a standard for modelling and specifying business process choreographies based on a flowcharting technique. The BPMN specification relies on a number of supporting standards such as XML Process Definition Language (XPDL) which is a file format used to store various aspects of BPMN diagrams [17]; and Web Services Choreography Description Language (WS-CDL) which provides a formalism for describing peer-to-peer collaborations between workflow participants using pi-calculus [233].

- Web Services Business Process Execution Language (WS-BPEL 2.0) [176] is the latest in the series of orchestration languages, uniting the ideas from the XLANG [226] and WSFL [144] languages, and extending the original Business Process Execution Language for Web Services (BPEL4WS [8]) specification. WS-BPEL 2.0 is arguably the most widely used orchestration language since it was developed by a consortium of major software vendors (namely IBM, Microsoft, and BEA) and has been recently standardised by OASIS [176].

Business processes are an integral part of SOA, constituting one of the fundamental design and implementation constructs in service-oriented systems. As such, they will be treated as distinct service implementation artefacts in the formal model of SO system design presented in Chapter 3.

2.2.2 Service-Oriented Computing (SOC) - Key Concepts and Definitions

While SOA represents a conceptual and architectural model without enforcing any constraints on the actual design and implementation of services (that is, services in SOA are treated as ‘black boxes’) and the individual service-oriented systems, Service-Oriented Computing (SOC) is the concrete software development paradigm based on the concept of encapsulating application logic within autonomous, stateless services exposed via well-defined service interfaces [67, 104, 181, 186]. Services in SOC are autonomous and stateless insofar as they do not depend on the context or state of other services in the system [183].

SOC covers all development phases of the Software Development Lifecycle (SDLC), including requirements engineering, system analysis and design, software implementation, testing, and maintenance of the final products. As such, SOC can be considered as synonymous to Service-Oriented Software Engineering (SOSE). Note that the term SOSE is not commonly used in the existing literature, although Papazoglou et al. [184] recently defined SOSE as one of the major research areas that requires attention of the research community. Also, the main focus of this research is the design of service-oriented systems; therefore, the other development phases are only covered briefly in this review.

---

7 Note that the underlined font will be used in the remainder of this chapter to indicate that the discussed material is directly related to the contribution and outputs of this research.
Figure 2-3 illustrates the design-level view of service-oriented system, where the design and implementation of the individual services is taken into consideration. For example, Design1 consists of three services, s1, s2, and s3, where each service consists of two distinct fundamental design artefacts: service interfaces and service implementation elements. As was described previously, services can be implemented using a range of different technologies and development paradigms. Similarly, there are no technological constraints on the languages and description formats used to describe service interfaces (although WSDL is commonly used to describe service interfaces in present implementations). To this end, services in SOC are somewhat similar to components in Component-Based Development (CBD) [11, 212], but they are typically more coarse-grained and business-related then components and implementation inheritance and its complications (common to components in CBD) are not present in SOC [145].

2.2.2.1 Development Strategies
There are three main strategies that can be used to develop service-oriented solutions: top-down, bottom-up, and meet-in-the-middle [12]. Such strategies are typically complementary and iterative and can be integrated into existing widely-accepted development processes (such as, for example, Rational Unified Process (RUP) [133]).
**Top-down** strategy starts with the functional and non-functional requirements and business process models and refines them in a stepwise fashion down to a software implementation. The top-down development is often referred to as domain decomposition, which consists of the decomposition of the business domain into its functional areas and subsystems [11, 139]. The crucial task of a top-down strategy is to identify the units of software (atomic services) of ‘right’ granularity that can be reused in different contexts. Atomic services can then be composed into coarser-grained composite services or business processes. The issue of service granularity is important to this research since we consider the identification of service interfaces as important SO design activities as discussed further in Section 2.2.2.3.

**Bottom-up** strategy is mainly related to the existing (legacy) systems, where the developers work upwards to the requirements and business process models by building services on top of existing systems. A bottom-up strategy includes two different techniques. Firstly, the developers can add a layer of service interfaces on top of existing systems, without changing the internal structure of such systems. Secondly, legacy systems can be refactored in such a way that the internal structure of the software system becomes service-oriented [142, 145]. To this end, examining the structural properties of software designs (using software metrics derived in this research) can assist software engineers in making an informed decision regarding whether it is best to refactor the system, or simply add a layer of service interfaces to it.

**Meet-in-the-middle** strategy is a combination of top-down and bottom-up approaches. At present, the only well-described meet-in-the-middle technique is a goal-service modelling (GSM) approach initially proposed by Levi and Arsanjani [12, 139], and recently elaborated by Arsanjani et al. [13] as part of the IBM’s Service-Oriented Modelling and Architecture (SOMA) [106] development methodology. GSM aligns existing software assets with business goals, by combining the top-down and bottom-up strategies, so that all services in the system can be traced back to some well-defined business goal.

Note that a top-down development strategy is arguably more interoperable than a bottom-up approach since avoiding language-specific types and starting with interface and message definitions can lead to a much higher likelihood of interoperability [138]. The drawback of top-down approach is that, in its full generality, it can only be applied to systems developed entirely from scratch [4].

Also note that there are conflicting opinions as to which general strategy should be used when developing service-oriented systems. For example, according to Spencer [221] and Fowler et al. [81], developers should not try to design an application into disparate Web services that talk to each other. Rather, they should build the application and expose various parts of it as Web Services (treating them as Remote Facades [82]). In contrast, Barry [18] and Singh et al. [219] indicate that simply adding Web Services to an existing application will not produce a service-oriented solution. They argue that the system should be composed
CHAPTER 2. LITERATURE REVIEW

from discrete internal and external services. The former view advocates a bottom-up approach, where developers build the application, add web services to it, and then combine services into business processes. Conversely, the latter view prescribes a top-down approach based on business domain decomposition.

2.2.2.2 Existing Development Methodologies

Although methodological support for the development of SO software applications is in its infancy, there are a number of approaches that cover various aspects of SO development:

- IBM’s Service-Oriented Modelling and Architecture (SOMA) [13, 105] and the “Methodology for Service Architectures 1.0” from OASIS [118] provide support for the identification and specification of services at the business level, as well as the composition of services into executable business processes. However, neither approach addresses design and implementation issues beyond the definition of service interfaces and identification of main service components that realise the services. Additionally, SOMA is a proprietary methodology available only by purchasing consultancy services from IBM (although, the detailed summary of the approach has been recently published in IBM’s Systems Journal [13]).

- IBM Redbook “Patterns: Service-Oriented Architecture and Web Services” [65] concerns various Web Service related technologies (such as SOAP, WSDL, UDDI), however, rather than containing abstract methodological processes or patterns, the redbook is more a technology specific developers manual. Same can be said about the IBM’s SOA Programming Model [77] which aims to simplify the creation and use of business services by making (IBM specific) middleware functions more accessible to the developers.

- The most complete SO development methodology to date is defined by the industry practitioner Erl [66-68]. Although this methodology offers principles of service design, and briefly discusses the structural properties and non-functional characteristics of service-oriented software, it lacks formal foundations and metrics, which can lead to ambiguity and lack of design verifiability. Furthermore, the methodology is not supported by empirical evaluation and thus is based more on Erl’s subjective judgement than a carefully constructed scientific approach. Nonetheless, Erl’s contribution is valuable since it provides detailed guidance for software practitioners. It is also regarded as useful academic text on SOA and SOC. For example, one of Erl’s books [66] is currently used as a reference text for the ‘Web Services’ subject taught at RMIT University, School of Computer Science and IT.

As for research contributions, the work of Papazoglou et al. [183, 184, 186] includes the most comprehensive support in terms of scope and coverage, describing the entire “Web Services Development Life Cycle” [183] including: Planning; Analysis and Design; Construction and Testing; Provisioning; Execution and Monitoring phases. Nonetheless, their methodology is still evolving, and as such, is not mature enough for the wide adoption in the industry.

(February, 2009)
The above methodological approaches can, either individually or in combination, be readily used to develop SO applications. Nonetheless, they do not provide any guidance in terms of the structural properties of service-oriented designs (such as coupling between services and cohesion of individual services). This can have a significant impact on the quality of the produced SO software products since it was shown in previous paradigms that structural properties of software have direct influence on software quality, especially in terms of the maintainability of software. To this end, the structural design metrics derived in this research can: i) enhance the existing approaches by providing formal and quantitative means for evaluating the quality of produced software designs; ii) form a foundation for a new methodological design approach.

2.2.2.3 Service-Oriented Design Considerations

Software design is the artefact produced in the design phase of the SDLC, which can be captured in the form of a physical document or other kinds of representation that articulate the intent of the software engineer [107]. According to Erl [68], Service-Oriented design incorporates principles for creating services with distinct design characteristics that support the overall vision of SOA. To this end, the major goal of the service-oriented design process is to provide a methodological support for the software practitioners facing a task of designing service-oriented solutions that can be integrated into an overall SOA. Such support must take into consideration the fundamental characteristics of SOA described in the previous section, namely reusability, business agility, interoperability, loose-coupling, and composability.

Note that the influence of the design process on the above characteristics varies. For example, interoperability refers to the platform-agnostic nature of Web Services, and as such, it is restricted by specific technological implementation and cannot be directly influenced at the design stage. Similarly, business agility is somewhat restricted by the need to include business processes in the system design, which can also be considered as technological constraint. Although, such restriction can be loosened by replacing the business process scripts with dedicated software components (such as services), as long as these components encapsulate all business logic and rules, and serve as the orchestrators of other services in the system. Moreover, loose-coupling in the context of SOA typically refers to the integration aspects (including separation of service consumers from service providers via the service registries) rather than the actual structural property of software design or implementation. Again, such integration related coupling cannot be directly influenced at the design stage.

The remaining two characteristics (reusability and composability) are highly dependent on the structure of service-oriented system designs. More specifically, they are related to two

---

8 The structural properties of software are discussed in detail in Section 2.4.
9 Presently, there is no standardised language or notation for expressing SO design artefacts.
CHAPTER 2. LITERATURE REVIEW

imperative features of SO design (and SOC in general) – service autonomy and granularity and relatedness of service operations [67, 134, 247].

Service autonomy means that all design elements in a given software system are clearly separated into distinct, stateless, and self-contained services that communicate with each other strictly via the service interfaces. That is, there are no direct inter-service relationships between services in the system. For example, Design1 shown in Figure 2-3 conforms to the principle of service autonomy, whereas Design2 does not conform to this principle. Note that from the architectural (SOA) perspective, both services will look exactly the same since they have identical service interfaces. The reusability of a given service will depend largely on its autonomy. This is because it would be difficult to ‘extract’ a service from one system and reuse it in another if the implementation of this service is linked directly to the implementation of other services.

We refer to this direct linkage as one of the most important aspects of structural coupling. The notion of autonomy is then directly related to the structural property of coupling, which is investigated in this thesis.

Note that some researchers and practitioners question the idea of structuring software systems as collection of services [221]. This is because there is a common misconception in the research and industry communities that services in SOC have to be implemented as Web Services. Given that Web Services are resource intensive due to the XML marshalling, structuring the whole system as a collection of Web Services may have a negative impact on its performance. In this research, we view SOC purely as the development paradigm, and as such we do not restrict service implementations to Web Services.

Service granularity and relatedness of its operations is another key design consideration in SOC. That is, service-orientation highlights the challenge of granularity, where services are typically categorised into fine-grained and coarse-grained types [67, 219]. A fine-grained service addresses a small unit of business functionality. In contrast, a coarse-grained service abstracts larger chunks of business capability within a single interaction. To date, there is no agreed criterion for determining the right granularity of services.

The concept of granularity is important because it has direct impact on the composability and reusability of services. For example, fine-grained services should conceptually be easier to reuse and composed into more complex composite services compared to the coarse-grained services [68]. Another important characteristic of SOC is the relatedness of the operations exposed in a service interface. Such ‘relatedness’ can be considered as major indicator of service cohesion. To our knowledge, the concept of service cohesion is yet to be investigated or even discussed in the existing literature. This is surprising given that cohesion has been long recognised as one of the most important structural properties of software.
The fundamental concepts of SOC described in this section, will be discussed further and formalised in Chapter 3. The metrics for measuring the coupling and cohesion of service-oriented software designs are presented in Chapters 4 and 5.

### 2.2.3 Key Terms

The terminology used in the rest of this thesis is shown in Table 2-2.

<table>
<thead>
<tr>
<th>TERM</th>
<th>DEFINITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOA</td>
<td>High-level architectural model covering logical and architectural aspects of service-orientation.</td>
</tr>
<tr>
<td>SOC / or SO DEVELOPMENT PARADIGM</td>
<td>The actual development paradigm covering the process of developing software applications structured in terms of autonomic services. SOC covers all phases of the software development lifecycle (SDLC) ranging from requirements specification and analysis to the maintenance and other post-release activities.</td>
</tr>
<tr>
<td>SO SOFTWARE SYSTEM/ or PRODUCT</td>
<td>Fully implemented and released software system. The system can contain a number of different artefacts including (SRS, analysis and design documents, executable source code, and other related documentation)</td>
</tr>
<tr>
<td>SO SOFTWARE DESIGN</td>
<td>i) The design of the SO software system (product); or ii) the activities undertaken when designing SO software system (process)</td>
</tr>
</tbody>
</table>

Table 2-2. SO terminology used in this thesis

### 2.3 Software Product Quality - Maintainability

Developing high quality software products is of prime importance and should be a key target of any software engineering process independent of the development paradigms or technological platforms in use [111]. One early definition of software quality was proposed by Rubey and Hartwick [209], where quality was considered as synonymous with ‘program goodness’ and characterised as ‘how easy the program should be to run production with and how easily it can be modified’ [209, p. 671]. The authors reasoned that for a software program to be of high quality, it must possess the applicable quality attributes as assessed by quantitative measurements.
Software quality was later recognised to be a somewhat ambiguous and multidimensional concept where different views are expected to exist [89]. For example, most of the existing definitions of quality can be generally classified into three different, and sometimes contradicting, perspectives: i) the conformance of software products to the functional requirements [89, 199, 231]; ii) the user satisfaction [160, 232]; and iii) the lack of errors or unexpected behaviour [39, 89, 169].

To support software engineers in performing a systematic and rigorous assessment of software quality, several highly-referenced models of software product quality have been introduced by Boehm et al. [25], McCall [157], Kitchenham [125], and Dromey [62]. These models are structured in a hierarchical, top-down manner, where the concept of software quality is divided into a number of quality attributes which in turn are further decomposed into sub-attributes. The system quality is then evaluated in a bottom-up fashion, where the degree to which each of the quality sub-attributes is present in the product reflects the overall quality of the software. Such models can provide a valuable insight into the area of software quality by covering important quality concepts and dimensions. Nonetheless, the existing research work on quality modelling is considered to be somewhat subjective, incomplete and not strong enough to gain wide acceptance [122].

In order to provide a unified and comprehensive framework for specifying and evaluating the quality of software products, the Joint Technical Committee of the International Standards Organisation (ISO) and the International Electrotechnical Commission (IEC) defined the international standard for software product evaluation, ISO-9126:1991 [110], which combined and extended the concepts and guidelines originally proposed by Boehm et al. [25] and McCall [157] into one generic model for characterising quality. This standard was recently replaced by the widely-used set of four standards, ISO/IEC 9126:1-4 [111-114] that incorporate a more prescriptive software quality model including a comprehensive set of metrics.

In this thesis, the decision was made to use the quality model and metrics defined in ISO/IEC 9126:1-4 standards when investigating and measuring the maintainability of service-oriented software so to be consistent with the current industry practices. Note that we acknowledge the concerns of some researchers in relation to Software Engineering (SE) standards in general [196]; and ISO/IEC 9126:1-4 in particular. For example, Al-Kilidar et al. [2] demonstrated two weaknesses of ISO/IEC 9126 standards in terms of overlapping between some of the measured properties and ambiguity in the definition of one of the quality attributes (software usability). Nonetheless, we believe that application of established international standards should be encouraged in both SE research and industry communities. This is because standards encapsulate uniform approaches for solving problems by concretising the common informal practices and development concepts [229], and therefore used extensively in all other engineering disciplines. Moreover, the quality model described in ISO/IEC
9126:1 standard has been additionally evaluated by Jung et al. [120] using a survey-based study. Although the results of the study again reveal possible ambiguities in the way the model defines its quality attributes and sub-attributes, they show direct evidence of the overall validity of the model.

2.3.1 Quality Model - ISO/IEC 9126

The set of ISO/IEC 9126 standards consists of four parts that address the following areas:

- Quality model - ISO/IEC 9126-1:2001 [111]
- Internal metrics - ISO/IEC 9126-3:2003 [113]

The quality model prescribed in ISO/IEC 9126-1 is summarised in this section. The metrics for measuring software maintainability (from the ISO/IEC 9126-[2-3]:2003 standards) are briefly discussed in Section 2.3.2.1. These metrics have been used as dependent variables in the empirical evaluation of the coupling and cohesion metrics derived in this thesis, and will be described in more detail in Chapter 6.

The ISO/IEC 9126 quality model captures software product quality as a multidimensional concept comprised of six characteristics which are further subdivided into sub-characteristics\(^{10}\) that can be measured directly by various internal or external quality metrics. The model was designed to be as generic as possible, and as such, it does not target any particular development paradigm or technological implementation [111]. For example, this model can be used effectively in its present state to assess the external quality of service-oriented software products, as was done in this thesis for the particular case of software maintainability.

Figure 2-4 illustrates the ISO/IEC quality model. It also provides the definitions for the maintainability quality characteristic and its four sub-characteristics: analysability, stability, changeability, and testability that can be directly measured using the internal and external ISO/IEC metrics. The analysability, stability, and changeability sub-characteristics of maintainability will be explicitly mapped to the metrics derived in this work in order to establish experimental hypotheses for the empirical study described in Chapter 6. The testability sub-characteristic is not investigated since it refers to the general capability of any software product to be tested, and as such, is not dependent on a particular development paradigm. Also, the investigation of other quality characteristics was considered outside of the research scope, but could be conducted in future work as discussed in Chapter 7.

---

\(^{10}\) The ISO/IEC 9126 standard substituted the terms [quality] attributes and sub-attributes, as commonly used in prior work, with [quality] characteristics and sub-characteristics respectively.
Figure 2-4. ISO/IEC 9126-1 quality model [14] – maintainability
2.3.2 Software Maintenance

The ISO/IEC 9126-1 standard does not provide the description of the actual software maintenance phase including its various activities and associated issues; therefore the area of software maintenance is discussed in detail in this sub-section in order to provide further rationale and additional background for this research.

Maintenance is the phase of the Software Development LifeCycle [131] that deals with the post-production modifications of a software product, and has long been regarded as one of the most resource-consuming development phases. For example, Boehm et al. [25, p.593] suggest that major benefit of the improved capability to deal with software quality considerations for any software development organisation would be an improvement in software maintenance cost-effectiveness.

It has been suggested that maintenance activities consume more than half of the overall project resources [103, 137, 148, 205]. More specifically, Page-Jones [180] notes that 60% of the whole lifetime cost of the system is spent on maintenance, while Pressman [199] states that most software development companies spend between 60% and 70% of the project resources on correcting, adapting, enhancing and reengineering existing software, and Zuse [251] writes that over 70% of the overall development effort is spent on testing and maintaining software products. This shows that developing software that is difficult to maintain (that is software exhibiting low maintainability) could result in project failures due to the time and cost overruns [153, 199, 231].

Interestingly, Holgeida et al. note that the amount of time spent on maintenance activities is shown to be stable on 60% (versus 40% spent on the development activities) in many empirical studies conducted over the last thirty years, “and not increasing to take up a larger and larger part of the work [due to software ‘ageing’ and increases in size], which many claimed would happen” [103, p. 690]. Although the authors do not deliberate on this point, it is reasonable to assume that the improved understanding of software development practices, and more recently the introduction of new development paradigms such as OO, allowed to manage effectively this supposed explosion of maintenance efforts. This could also be considered as one of the main reasons for considering a wide adoption of SOC since as the software products continue to become increasingly large and complex, SOC can facilitate more efficient development process and easier implementation of maintenance tasks. For example, properly designed service-oriented solutions should exhibit a high degree of reusability, and according to Mari [151, p. 25], reusability can have a positive effect on maintainability due to the reduction of development costs.
CHAPTER 2. LITERATURE REVIEW

There are four widely-accepted types of maintenance activities originally defined by Lientz and Swanson [147] and later supported by other researchers and practitioners [153, 197, 199, 225, 231]:

- **Corrective**: fixing software faults (or defects), where a fault can result from errors introduced during the requirements, design or implementation development phases;

- **Preventive**: various activities aimed at increasing the maintainability of a product and preventing software faults before they occur by, for example, including additional documentation and improving the design structure;

- **Adaptive**: adapting software to changes in the environment, where environment can include hardware, middleware, operating systems and other technology related factors;

- **Perfective**: functional modifications to the system performed in order to accommodate for new or changed user requirements or to enhance the existing functionality.

These activity types are consistent with the ISO/IEC 9126 standard, where maintainability is characterised in terms of *corrections, improvements* (preventive or perfective) or *adaptation* of software.

Most of the maintenance efforts are typically spent on the *perfective* activities: Lientz and Swanson [147] demonstrate that at least half of the maintenance efforts can be considered as perfective, and Pigoski [197] notes that about 55% of all software change requests are related to new or changed requirements (perfective maintenance). More recently, the International Software Benchmarking Standards Group (ISBSG) investigated the maintenance patterns of 54 commercial software systems [109] from the communications, finance, and manufacturing domains in order to determine percentage of time spent on the individual types of maintenance activities. The results of the study are shown in Figure 2-5, where *perfective* maintenance was again shown to be the predominant activity.

![Figure 2-5. Balance of Maintenance Activities [109]](image)

(Febauary, 2009)
The perfective maintenance is highly relevant to the service-oriented software products which typically include a large number of business rules and associated business processes as discussed in Section 2.2. Such business processes are shown to be the most unstable part of software applications [234]. This suggests the potential increase of the rate and number of perfective maintenance tasks required to keep up with the rapidly changing business requirements. To this end, developing service-oriented software products that exhibit high maintainability is one of the key challenges of SOC. Note that developing highly maintainable software implies increasing development costs; therefore, the best practical level of maintainability is typically the accepted (or stakeholder agreed) level [83].

2.3.2.1 Measuring Maintainability

A number of quantitative measures have been proposed to directly assess the maintainability of software mainly based on the cost and effect of the modification activities [1, 83, 147, 231]. To measure the maintainability of service-oriented software products in this research, we used maintainability metrics defined in ISO/IEC 9126-(2-3) standards [112, 113] so as to be consistent with the earlier decision to utilise these standards. These metrics are used to directly measure the sub-characteristics of maintainability (refer to Figure 2-4) and are separated into external and internal metric types. They are summarised below with the detailed description of all metrics presented in Section 6.3 (as was mentioned previously, these metrics will be used as dependent variables in the empirical study described in Chapter 6).

- ISO/IEC 9126-2:2003 External metrics: The external metrics are computed by observing the behaviour of the maintainer or user when the software is maintained. For example, the external metric for measuring the changeability sub-characteristic of maintainability is the Modification Complexity (MC) metric defined as: \( MC = \frac{\text{Sum}(A)}{N} \), where \( A \) is the work time spent to change; and \( N \) is the total number of changes.

- ISO/IEC 9126-3:2003 Internal metrics: The internal metrics are computed by measuring the effect of modifications on the product itself. For example, one of the internal metrics for measuring the stability sub-characteristic of maintainability is the Modification Impact Localisation (MIL) metric defined as: \( \text{MIL} = \frac{A}{B} \), where \( A \) is the number of emerged adverse impacts in the system after modifications; and \( B \) is the total number of modifications made.

2.3.2.2 Predicting Maintainability

Quality characteristics of software, such as maintainability, should be estimated as early in the Software Development Lifecycle (SDLC) as possible in order to allow timely identification and correction of the potential quality problems prior to the release of the software product if required. For example, the early prediction of maintainability can allow software practitioners to optimise future maintenance costs [164, 205]. To this end, a number of quality prediction models [16, 62, 73, 74, 148, 177, 197] have been established in the form of:
Quality Characteristic = \( f \) (influencing factor/s), where a given quality characteristic is considered to be a function \( f \) of relevant factors that can influence the quality characteristic in question.

In terms of the maintainability prediction, the influencing factors can be generally categorised into: documentation-related factors and the structural properties of software designs.

Example documentation-related factors include: i) readability of source code (the percentage of comment lines in total code), documentation contents quality, and understandability of software (the correlation between documentation and source code) [1]; ii) documented preconditions and post-conditions for all functions in source-code, comments for all source-code blocks, and self-descriptive identifiers [62]; and iii) overall quality of the documentation [153] (for example, the thoroughness of the activity logs [113]).

The completeness and quality of the product documentation can have a considerable impact on the analysability sub-characteristic of maintainability given that it is directly related to the important and time-consuming cognitive task of program comprehension [223], which takes up approximately half of all maintenance efforts [137]. Nevertheless, the documentation-related aspects cannot be used effectively to predict the other sub-characteristics of maintainability that are of interest to this research, namely changeability, and stability.

In contrast, the structural properties of software designs (such as size, complexity, coupling, and cohesion) are shown to influence all aspects of software maintainability [83, 170, 225]. According to Zuse, “good software design causes lower maintenance costs” [251, p.7]. This is because maintenance activities can be performed efficiently only if the earlier development phases (such as design phase) are done correctly [205].

The structural properties of software designs constitute the fundamental construct in the simple maintainability causal model used in this research. Figure 2-6 provides a schematic view of this model, which is loosely based on the quality model proposed by Bansiya [16].
where the quality of OO software designs was modelled based on the corresponding structural properties. The structural design properties themself are discussed further in Section 2.4. Note that the presented model is incomplete, showing only these aspects that have already been covered in this review. The model will be augmented with additional constructs at the end of this chapter after all the relevant concepts have been covered.

Finally, it also important to mention that in addition to the software product-related factors discussed above, there are a number of process-related external factors that can also influence the maintainability of software. The process-related factors do not consider the software product itself; instead, they cover various issues related to the development process practices and other project related considerations. Such factors can include: i) social aspects related to program comprehension [223]; ii) user knowledge and maintainer effectiveness [148]; iii) the quality of the maintenance processes and practices (based on, for example, Software Maintenance Maturity Model [9]); and iv) the thoroughness of software inspections during the design and implementation phases of SDLC [243]. These factors are not investigated in this thesis because: i) they are difficult to control prior to the software release; and iii) the investigation of project-related factors is out of scope of this research.

2.4 Structural Properties of Software Designs

The design of any software product possesses a number of properties that can be assessed by measuring the structure of the design artefacts using software metrics. Such structural properties are said to capture the (internal) quality of software and are commonly referred to as internal quality characteristics [16, 35, 62, 71, 74, 98, 251] since they do not describe the visible quality of a product, rather, they have a causal impact on the (external) quality characteristics such as maintainability, reliability, and performance. According to Samoladas et al. [211, p. 84], the external quality characteristics should always be correlated to internal quality characteristics.

2.4.1 Overview

There are four major structural properties that are commonly used to represent the quality of any software design irrespective of the development paradigm in use: size, complexity, coupling, and cohesion. These properties can be broadly defined [16, 34, 63, 84, 222] as:

- **Size**: a number of design artefacts in the system design; or the amount of functionality a software system provides to a user;
- **Complexity**: degree of difficulty in understanding the structure of design artefacts; or the amount of the internal work (algorithmic complexity) performed by a design artefact;
CHAPTER 2. LITERATURE REVIEW

- **Coupling**: a number of relationships between design artefacts; or the strength of a relationship established by a connection from one artefact to another;

- **Cohesion**: degree to which the elements of a design artefact belong together according to some defined criteria.

An interesting observation can be made in relation to the above definitions: they all include measurement-related keywords, such as number, amount, degree, and strength, suggesting that the definitions are precise and unambiguous. This is misleading since the definitions of structural properties are typically semantic and are subject to different interpretations. For example, the strength of coupling relationships can be interpreted in a number of ways. According to Briand et al. [34, p. 724], the properties of size, complexity, cohesion, and coupling are hardly ever defined in a precise and unambiguous way. This ambiguity in the definition of structural properties is mainly due to the following reasons:

1) **Multi-dimensional nature of properties**

   Structural properties of software typically incorporate multi-dimensional aspects [24], that is, the properties can be conceptually separated into a number of sub-properties or influencing factors. From the measurement perspective, the properties can be considered as the complex attributes [98]. For example, some of the influencing factors of coupling can include: types of the relationship, interface complexity, and the direction of communication [33]. To this end, the correlation between external and internal quality can be characterised as: **external quality attribute** (for example, software maintainability) is reflected by the **internal quality attributes/structural properties** (for example, coupling) which in turn are reflected by the **internal quality sub-attributes** (for example, direction of coupling communication). Furthermore, to constrain the definition of the structural properties, it is necessary to identify the common influencing factors based on the specific technological and conceptual viewpoints [71].

2) **Level of abstraction**

   The structural properties of software can be measured at different levels of abstraction, ranging from requirements specifications through to executable implementations, with the target level of abstraction influencing the definition and consequent measurement of structural properties. For example, the property of size can be applicable to requirements specification documents, software designs, and software implementations. The other three properties of coupling, cohesion, and complexity are typically investigated at the design and implementation level. The level of program abstraction has a significant influence on the definition of structural properties. For example, complexity is commonly defined in terms of the algorithmic complexity of software modules [99, 156, 237]. The information required to measure the algorithmic complexity sometimes not available at the design stage, and thus, such defini-
tions would be restricted to software implementation (with the exception of atypical cases where this information has been defined in the requirements specification itself and propagated throughout the design).

It has been recommended that structural properties should be evaluated as early as possible since the sooner problems in the software structure can be identified, the lesser the effort required to correct them [10, 15, 34]. Moreover, measuring structural properties late in the development process (after the implementation phase) defeats the purpose of such attributes being used as predictors of external quality attributes. This is because the external quality attributes can be measured directly if the system is already implemented.

The quantification of structural properties is more difficult at the design stage compared to the implementation stage because data available during the design stage is usually limited. Most of the previous work in the software quality area examined the structural properties at the implementation level, but more recent research suggests that such properties should be examined as early in the development lifecycle as possible [16, 36, 179]. In this research, the structural properties are investigated at the design level so to provide mechanism for the earliest possible evaluation of software maintainability.

3) Different design paradigms

Previous research has shown that the use of different development paradigms, such as Procedural design and OO, will result in systems with different structural properties [63, 102]. This is because structural properties have more dimensions, and as such, are more difficult to measure in OO systems compared to procedural ones due to the existence of many additional design constructs and mechanisms. For example, OO introduced additional design concepts of: object abstraction, inheritance, polymorphism, and class hierarchies that can impact the structural properties of software. Similar can be said about SOC, where the introduction of additional level of design abstraction, namely a service, and associated design principles of service autonomy and service granularity (discussed in Section 2.2.3 and formalised in Chapter 3) can influence the design structure of software.

Note that this thesis investigates the impact of service-orientation on coupling and cohesion properties only. The properties of size and complexity are not investigated due to the following reasons:

- *Size* should be directly dependent on the functional requirements of the software system; therefore little can be done from the software engineering perspective to ‘improve’ the size of the system. Additionally, *size* is independent from the development strategy in use and existing approaches for measuring size [76] [224] should be directly applicable to SOC.

- *Complexity* can only be fully quantified after the implementation of software is con-
CHAPTER 2. LITERATURE REVIEW

cluded; it cannot be directly measured during the design phase. This is because typical approaches for investigating complexity are based on examining the algorithmic complexity (or internal complexity) of implemented software modules as described by Henderson-Sellers [98] and McCabe [156].

- Complexity is commonly viewed as the combination of coupling and cohesion when investigated at the design level. For example, Chidamber and Kemerer [44] define OO complexity as the complex attribute that is influenced by many factors including system coupling and class cohesion, and Briand et al. suggest that all aspects of software design properties can be related to the complexity [27, p. 69]. To this end, investigating the coupling and cohesion in this research allows us to ‘indirectly’ cover design complexity.

The following sub-sections describe the properties of coupling and cohesion in more details. Note that the existing metrics for measuring these properties in Procedural and OO development are presented in a separate section (Section 2.5.4) to improve thesis readability.

2.4.2 Coupling

The concept of coupling was originally defined for procedural systems by Stevens et al. as “the measure of the strength of association established by a connection from one module to another” [222, p. 233], where coupling was classified based on the type of connection – data or control. The authors had later extended their definition [242] in order to characterise four major factors that influence coupling: i) type of connection between modules; ii) complexity of the interface; iii) type of information flow; and iv) binding time of connection.

The notion of coupling was then extended for object-oriented (OO) systems due to the existence of additional mechanisms that can influence coupling, such as polymorphic relationships [44, 63, 102, 146]. Also, there are two main subjects of interest (or design constructs) in OO design, namely classes and methods, as opposed to procedural systems where the procedure (a module consisting of code statements) is the main subject of interest. To this end, coupling in OO systems is defined as “the interdependency of an object on other objects in a design representing the count of other objects that would have to be accessed by a given object in order for that object to function correctly” [16, p.7]. Currently, there are four major frameworks characterising various dimensions of OO coupling:

- Eder et al. [63] describe coupling in terms of three different types of relationships including: i) interaction relationships between methods; ii) component relationships between classes; and iii) inheritance between classes. These relationships are then used to derive three dimensions of coupling: interaction; component; and inheritance. For each dimension of coupling, the different strengths of coupling are identified. For example, the strengths of interaction coupling are listed below from strongest to weakest: Content, Common, External,
Control, Stamp, Data, No direct coupling. Note that a complete software implementation is required to determine the component and inheritance dimensions of coupling; however, the interaction coupling in which a method invokes another method (of a different class) can be examined at the design stage.

- Hitz and Montazeri [101, 102] propose two different types of coupling: class level coupling (CLC) and object level coupling (OLC). CLC captures i) relationships between a method of a given class with a method of another class via direct call; and ii) references from a method of a given class to the attributes of another class. OLC captures coupling based on the state dependencies between two objects during run-time. CLC is considered to be important when investigating software maintainability because changes in one class may lead to changes in other classes which use it. OLC on the other hand, influences various run-time activities such as testing and debugging. As with the Eder et al. [63] framework, a number of general factors determining the strength of a particular coupling type are identified. Note that both types of coupling are difficult to examine at design time, with the only exception being method-to-method interactions as part of CLC which can be obtained from UML sequence diagrams and can be considered as similar to the interaction coupling of Eder et al. [63].

- Hall et al. [95] categorise coupling into four different aspects: i) size of interface (amount of data passed to the module); ii) type of information flow (control or data); iii) type of passed data (simple data or entire structures); and iv) type of connection (information passed is global or in parameter lists). Additionally, three coupling domains that cover the above aspects are proposed: i) in-coupling representing the complexity (size) of the interface; ii) out-coupling representing the interactions between a module and other modules; and iii) global connection representing the complexity of global variable usage in a program. The out-coupling is conceptually similar to the interaction coupling of Eder et al. [63] and CLC of Hitz and Montazeri [101, 102].

- Briand et al. [33] consider coupling to be representative of the interactions between classes. In contrast to the previous three frameworks which mainly focus on the implementation-level coupling, this framework examines coupling based on the information available during the high-level design phase. According to the authors “eliminating design flaws and errors early before they can propagate to subsequent phases can save substantial amounts of money” [33, p. 97]. Given our goal of measuring coupling at the design level, the definition of service-oriented design coupling in this thesis generally follows the framework of Briand et al. This framework concentrates on coupling caused by interactions that occur between classes. Three coupling aspects are identified that determine the overall strength of coupling of a given design artefact:
1) **Type of interaction.** Defines the mechanism by which two classes c and d are coupled:
   i) Class-attribute: class c is the type of an attribute of class d;
   ii) Class-method: class c is the type of a parameter of method m_d; or class c is the return type of method m_d;
   iii) Method-method: method m_d directly invokes method m_c; or m_d receives via parameter a pointer to m_c thus invoking it indirectly.

2) **Coupling relationship.** Two classes can be connected via one of the three common relationships: *inheritance, friendship, and other*. Note that Briand et al. targeted C++ based systems in their framework, and as such the *friendship* relationship type is language dependent. Nevertheless, Briand et al. suggest that the relationship types can be easily redefined based on the technological constraints.

3) **Locus of impact of an interaction.** Can be *export* (class c is the used class) in the interaction, or *import* (class c is the using class) in the interaction.

   In this thesis, the types of interaction and coupling relationships from Briand et al. framework [33] are redefined according to the fundamental principles of service-orientation. Also, the locus of impact of an interaction is considered when investigating coupling. For example, service-oriented design coupling relationships cover: i) types of service design artefacts involved in interactions; and ii) locality aspects of the relationships in respect to the service boundary, that is, whether the relationship is inter- or intra-service. The redefined types of interactions and associated relationships are formalised in Chapter 3. Additionally, a new dimension of coupling is proposed in Chapter 4, *service autonomy*, which cannot be directly mapped to the above framework.

### 2.4.3 Cohesion

The notion of cohesion has been widely discussed in the context of the OO and procedural paradigms with various qualitative classification schemes being proposed to describe different levels of cohesion [63, 222, 242]. For procedural systems, cohesion was originally defined by Stevens et al. as a “*measure of the degree to which the elements of a module belong together*” [222]. It was also suggested that in a highly cohesive procedural module, all elements should be related to the performance of a single function. Additionally, the authors proposed six semantic categories of module cohesion that were later elaborated and extended by Yourdon and Constantine [242]. The seven resultant categories are defined below ranging from the weakest to the strongest types of module cohesiveness:

- **Coincidental:** the elements of a module have nothing in common besides being within the same module;
Logical: elements with similar functionality such as input/output handling are collected in one module;

Temporal: the elements of a module have logical cohesion and are performed within the same time period;

Procedural (added in [242]): the elements of a module are connected by some control flow;

Communicational: the elements of a module are connected by some control flow and operate on the same set of data;

Sequential: the elements of a module have communicational cohesion and are connected by a sequential control flow;

Functional: the elements of a module have sequential cohesion, and all elements contribute to a single task in the problem domain, thus potentially minimising maintenance efforts [31].

The notion of cohesion was later extended for the OO paradigm in a framework proposed by Eder et al. [63], where cohesion was redefined as the “degree to which the methods and attributes of a class belong together” in order to cover for the additional design constructs introduced by the OO paradigm. More specifically, Eder et al. adopted the original cohesion categories of Stevens et al. [222] when investigating the cohesiveness of individual class methods, at the same time introducing five new qualitative categories of OO class cohesion:

Separable (weakest): the objects of a class represent multiple unrelated data abstractions. For instance, the cohesion of a class is separable, if the methods and attributes can be grouped into two sets such that any method of one set invokes no methods and references no attributes of the other set;

Multifaceted: the objects of a class represent multiple related data abstractions. The relation is caused by at least one method of the class which uses all the data abstractions;

Non-delegated: there exist attributes which do not describe the whole data abstraction represented by a class, but only a component of it. That is, the attributes of the class interpreted as relation schema violate third normal form;

Concealed: there exists some useful data abstraction concealed in the data abstraction represented by the class. Consequently, the class includes some attributes and methods which might make another class;

Model (strongest): the class represents a single, semantically meaningful concept.

Similar class-level categories of cohesion have been also suggested by Bieman and Kang (“relatedness of module components” [23, p. 259]).
It is important to note that the process of assigning design artefacts to a particular cohesion category has a subjective nature, and thus cannot be automated. As such, the above classifications have limited practical applicability. Nevertheless, they can provide strong conceptual premises for establishing practical approaches for characterising and quantifying cohesion using software metrics. The existing cohesion metrics are overviewed in Section 2.5.4.

In this thesis, the conceptual categories of cohesion introduced by Stevens et al. [222] and Eder et al. [63] are extended and modified in order to account for the unique characteristics of service-oriented designs. Additionally, two service-oriented categories of cohesion, external and implementation have been introduced. The conceptual categories are then used to drive the definition of measurable characteristics and derivation of service-oriented cohesion metrics in a systematic manner consistent with the principles of measurement theory. The categories and associated metrics are presented in Chapter 5.

2.4.4 Discussion

The structural properties of coupling and cohesion of software designs are yet to be thoroughly investigated in the context of SOC. For example, a commonly used term loose-coupling refers to the technological and integration based aspects of SOA, rather than the actual design principles incorporated by SOC as described in Section 2.2.2. This is unfortunate since it has been suggested that high quality software should be underpinned by a properly structured software design that exhibits low coupling and high cohesion in any development paradigm [32, 44, 72].

More specifically, the structural properties can be used as a guide for choosing alternative design approaches and artefacts. For instance, a design approach may be preferred over another because it produces designs consisting of loosely-coupled artefacts, or a design artefact may be preferred over another because it is more cohesive [30]. Such application of structural properties is important to the emerging field of SOC given a lack of mature design methodologies. More importantly in the context of this research, the properties are shown to be valuable early predictors of external quality characteristics (such as maintainability) in both Procedural and OO paradigms. For example, Table 2-3 shows the perceived influence of coupling and cohesion on the sub-characteristics of maintainability.

Note that coupling and cohesion are commonly considered to be conflicting factors. This is because coupling is reduced when the relationships among modules are minimised. To this end, the simplest way to achieve best possible coupling is to develop a system consisting of one (large) module only. This approach would be considered as bad design practice since the resultant module will be unnecessary large and difficult to analyse, which can potentially result in the decreased cohesion.
CHAPTER 2. LITERATURE REVIEW

<table>
<thead>
<tr>
<th></th>
<th>ANALYSABILITY</th>
<th>CHANGEABILITY</th>
<th>STABILITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUPLING</td>
<td>$S$ [146] [170]</td>
<td>$S$ [121] [170] [55]</td>
<td>$S$ [170] [55]</td>
</tr>
<tr>
<td>COHESION</td>
<td>$S$ [146] [55]</td>
<td>$W$ [121] [53]</td>
<td>$U$</td>
</tr>
</tbody>
</table>

Table 2-3. Influence of the structural properties of coupling and cohesion on software maintainability ($S$ - strong; $A$ - average; $W$ – weak; $U$ - unknown)

Most of the existing work in quality estimation and prediction based on the structural properties of software investigates coupling and cohesion in isolation [32, 53, 244]. Nevertheless, some recent empirical results suggest that coupling and cohesion should be examined in combination when predicting the maintainability of software products [55].

In this research, the decision was made to study and measure coupling and cohesion in isolation so to be consistent with previous studies in which it was demonstrated that coupling and cohesion can have a distinct causal impact on external quality attributes such as maintainability or fault-proneness in both Procedural [100] and OO [7, 38, 45, 49, 92, 164, 177] software. As such, a derived suite of design-level SO coupling and cohesion metrics, which is the central contribution of this thesis, can be separated into the coupling and cohesion metric types because it was designed to measure these concepts in isolation. The following section provides a detailed overview of the software metrics area.

2.5 Software Metrics

The need to develop high quality software products has led to an increasingly large body of work being performed in the area of software measurement [87], where measuring software quality involves the use of metrics to assign a value to the attributes under investigation [112-114]. Note that although the ISO/IEC 9126 standards define metric as a measurement scale and a method used for measurement, in Software Engineering the term metric is sometimes considered synonymous to measure [72]. In this thesis, we follow the ISO/IEC 9126 definitions where measure indicates the actual number or category obtained by making a measurement. Therefore, there is a clear distinction between both terms.

A correctly implemented measurement process can provide software development organisations with concrete mechanisms for controlling the quality of software products in an effective and systematic manner [160]. Additionally, a significant challenge to software engineers is to avoid neglecting proper development process while advancing among the technology
dimension [75]. To this end, metrics can provide mechanisms for quantifying the ways in which processes, products, and technologies relate to one another [158].

It has been argued that software engineering is fundamentally an empirical subject, and as such, metrics should play a pivotal role within it [76]. In fact, there is a large number of software engineering metrics being proposed in the research and industry literature. For example, Zuse [251] estimated in 1998 that there are approximately 1500 different metrics being proposed for measuring various aspects of software products and processes. Furthermore, the area of software measurement has been recognised as one of the most crucial software engineering disciplines in SEI’s Capability Maturity Model Integration (CMMI) process improvement methodology [46].

Nonetheless, according to Fenton “metrics continue to lie at the margins of software engineering” [76]. This is mainly due to the large gap between the theory and practice in the software metrics area as discussed by Glass [90, p. 221]. Moreover, in the past, software measurement has typically suffered from a lack of: i) standardised terminology; and ii) a formalism for expressing metrics in an unambiguous and fully operational manner (that is, a manner in which no additional interpretation is required on behalf of the user of the measure) [37]. Section 2.5.1 describes key concepts and definitions related to software metrics.

There are two general types of criticism applicable to current software metrics:

i) Various researchers in the metrics field [34, 72, 98, 207, 237] have noted that most existing software metrics were derived without any theoretical foundation, and as such, they lack appropriate mathematical properties. This suggests that software metrics should be created and validated with theoretical and mathematical rigor. Section 2.5.2 discusses the key principles of measurement theory that was used to derive theoretically sound metrics in this thesis, and also overviews existing approaches for theoretical validation of metrics.

ii) Although some of the existing well-known metrics are theoretically sound, they lack empirical evaluation [42, 228], which is arguably the most important validation of any metric since it allows establishing models for predicting (external) quality of software. Section 2.5.3 describes existing approaches for empirical evaluation of metrics, and also presents some empirical studies related to this research.

Finally, Section 2.5.4 describes the existing Procedural and OO coupling and cohesion metrics in order to provide necessary background for this research.

2.5.1 Key Concepts and Definitions

The software measurement framework proposed by Fenton [71, 72, 76] is widely considered to provide the most complete conceptual model and terminology for reasoning about software metrics. This framework has been adopted by a number of prominent researchers in the met-
CHAPTER 2. LITERATURE REVIEW

rics area [27, 160, 251], and more importantly, it is consistent with the concepts and definitions prescribed by the ISO/IEC 9126 family of quality standards used in this research. The measurement framework revolves around three fundamental measurement constructs: entities, attributes, and measurement.

1) Entities. A key task of any software measurement is to identify and characterise the entities that we intend to measure. Three separate classifications types of entities have been proposed:

- Process – a collection of related software engineering activities, methods, and practices employed when developing or maintaining the products;
- Product – an artefact that is the output of the process activities, including documentation, software design, or the actual code of a software program;
- Resource – an input used by the process activities to produce and maintain products, including hardware resources and personnel available throughout the SDLC.

Note that these types of entities are inter-dependent. For example, product deficiencies can imply the existence of a problem in the actual process used to develop this product [251].

The entity investigated in this research is the service-oriented software design, which can be classified into product entity type according to the above. Therefore, the rest of the definitions and concepts presented in this section target the metrics for measuring product type entities.

2) Attributes. An attribute of a given entity represents any measurable feature or property of this entity where there is a fundamental distinction between external and internal attributes. External attributes are characteristics or features of the software product that are externally visible. For example, software quality characteristics, such as maintainability, defined in ISO/IEC 9126 standard (described in Section 2.3.1) are examples of external attributes. In contrast, internal attributes such as structural properties of software designs (described in Section 2.4.1) can be measured in terms of the product itself.

The attributes investigated in this thesis are the structural properties (or internal attributes) of coupling and cohesion of service-oriented software designs.

3) Measurement. Measuring software involves direct or indirect quantification of the attributes of entities. Direct measurement of an attribute does not depend on the measurement of other attributes. In contrast, indirect (or derived) measurement of a given attribute involves the measurement of other attributes. Additionally, there are two typical measurement applications of metrics, assessment and prediction. Measurement for assessment is applicable to the attributes of existing entities, whereas predictive measurement of an attribute is based on a predictive model and associated hypotheses that link the measures of the attributes of existing
entities (such as properties of software designs) to external attributes of some future entity (such as maintainability of final software products).

The metrics derived in this research are direct metrics for the assessment of coupling and cohesion of service-oriented software designs and subsequent prediction of analysability, stability, and changeability sub-characteristics of maintainability of software products.

Additionally, there are two fundamental measurement constructs that have to be clearly defined for all metrics in order to support the objective interpretation of the obtained metric values and identification of the applicable statistical analysis techniques [14, 75, 235]. These constructs are measurement scale and types of measures:

**Measurement Scale.** There are five possible types of measurement scale with each scale type covering a set of values, continuous or discrete, or a set of categories to which an attribute is mapped. More specifically, each scale type can be formally captured in the form of \( M' = f(M) \), where \( f \) is the admissible function (or admissible transformation [27]) indicating a possible mapping from metric \( M \) to metric \( M' \). The scale types play a pivotal role in determining the theoretical soundness of the metrics by constituting a key construct of the measurement theory described in Section 2.5.2. The scale types are defined below from the least informative type to the most informative type:

- **Nominal**: \( M' = f(M) \), where \( f \) is any one-to-one mapping

  The nominal scale indicates some form of classification. There is only one possible empirical relation defined for nominal scale, equality, which can be mapped to the formal relations ‘=’ and ‘≠’. For example, classifying software designs into “OO-based” and “SOA-based” leads to nominal scale metrics.

- **Ordinal**: \( M' = f(M) \), where \( f \) is any monotonic increasing mapping

  The ordinal scale indicates some form of classification and ordering. The possible empirical relations are related to equality and order (formal relations ‘<’ and ‘>’). For example, assigning values “high”, “medium”, and “low” to the quality of software designs leads to an ordinal scale of metrics.

- **Interval**: \( M' = f(M) \), where \( f(M) \) is in the form of \( aM + b, a>0 \)

  The interval scale represents an ordered rating scale where the difference between two metric values has an empirical meaning. However, the ratio of two measures may not have the same empirical meaning because a zero position of \( M' \) does not indicate the absence of the quantity. The empirical relations possible are related to equality, order, and difference (formal relations ‘+’ and ‘- ’). For example, the temperature measured using the degrees (Celsius) is defined on an interval scale.

- **Ratio**: \( M' = f(M) \), where \( f(M) \) is in the form of \( aM, a>0 \)

  The ratio scale is an interval scale with the additional property that its zero position indicates the absence of the quantity being measured. This implies that in addition to the differ-
ence between two measures, the proportion of two measures have the same empirical meaning. The empirical relations possible are equality, order, difference, and relative difference (formal relations ‘/’ and ‘*’). For example, measuring the coupling of a system by counting the number of relationships between its design artefacts leads to a ratio scale metric. The coupling metrics derived in this work are defined on a ratio scale as discussed in Chapter 4.

- **Absolute**: $M' = M$ since they can be measured only in one way

  The absolute scale implies that any empirical and formal statement relating to measures is meaningful. Typically, the measure is considered to be defined on the absolute scale when it represents the result of dividing one ratio scale type measure by another ratio scale type measure where the unit of measurement is the same [111]. For example, dividing LOC by the number of comment lines in the code will result in an absolute scale measure. Note that most of the existing literature only defines the former four scale types [71, 89, 218] since absolute scale can be considered as a specialised case of a ratio scale. The absolute scale type is introduced in this thesis in order to be consistent with the definitions of scale types from the ISO/IEC 9126 standards. The cohesion metrics derived in this work are defined on ratio and absolute scales as described in Chapter 5.

**Types of measures.** In order to concretise the procedures for collecting metrics data, interpreting the results of measurement, and normalising measures for comparison, it is important to identify the type of measurement (and a corresponding measurement unit) employed by a metric. For example, only measures of the same type can be directly compared or combined into more complex metrics. There are three main types of measurement commonly used in software engineering: size (e.g. function size), time (e.g. elapsed time), and count (e.g. number of relationships between design artefacts). The metrics derived in this work use count as the measurement type as described in Chapters 4 and 5.

### 2.5.2 Theoretical Basis and Validation Approaches

The process of measuring software attributes should follow a well-defined theoretical approach, where software metrics adhere to the fundamental principles of measurement. The theory used commonly to guide the measurement of software products is the representational theory of measurement, or measurement theory [204]. Measurement theory serves as the basis for developing, reasoning about, and applying metrics [27, 72, 237, 251]. More specifically, it prescribes the important mathematical properties to the metrics ensuring that: i) metrics are categorised into distinct scale types; ii) statistical techniques are applied appropriately based on the scale types of metrics (for example, parametric statistics cannot be applied to the metrics defined on a lower than interval scale); and iii) transformations that are not permissible for some scale types are avoided [27].

(February, 2009)
CHAPTER 2. LITERATURE REVIEW

The use of measurement theory is especially relevant to the area of software design metrics given a lack of accepted definitions of the structural design properties as described in Section 2.4.1. The following brief overview of the measurement theory combines descriptions provided by Briand [27], Fenton [75], Melton [160], and Zuse [251].

2.5.2.1 Principles of Measurement Theory

The fundamental principle of measurement theory is that if in a given problem domain there exists an empirical understanding\(^{11}\) of relationships of objects within this domain, then such relationships could be formalised mathematically. Moreover, there could be some common understanding of one or more binary operations that can be applied to these objects. This principle can be formally captured using three key constructs: empirical relational system, formal (or numerical) relational system, and a mapping between the empirical and formal systems that represents specific metrics.

The empirical relational system \((E)\) is a model of the problem domain representing the common knowledge about the phenomenon to be measured. The empirical system needs to be mapped to a formal relational system, or a formal model, \((F)\) which formalises the intuitive understanding of the relationships between attributes in a precise mathematical way. The formal model of service-oriented design is presented in Chapter 3.

A theoretically valid metric \((\mu)\) should then demonstrate the equivalence between the empirical and formal systems, where the mapping from one relational system to another that preserves all relations and defines all admissible transformations is called a homomorphism. According to Zuse [250], the homomorphism is the fundamental notion of measurement that leads to the definition and classification of measurement scales. The formal definitions of the measurement theory constructs \((E, F, \mu)\) can be found in Appendix B, which also illustrates an example application of measurement theory to the measurement the height of a human.

Software artefacts and their properties are not physical objects and their relations are not well understood compared to the physical properties, such as height. To this end, there is a need to constrain and validate software metrics using axioms that prescribe required mathematical characteristics to the metrics based on the intuitive understanding of the problem domain as explained in the Subsection 2.5.2.2. The axiomatic metric validation approach is used in this research to validate the derived metrics.

Note that there is a common understanding that software attributes in general should be measurable on at least an ordinal scale [158]. In our opinion, when measuring structural properties of software designs, it is advisable to define metrics on a ratio scale in order to support more complete reasoning about properties in question. It is not sufficient to state that

\(^{11}\) In the context of measurement theory, empirical understanding reflects the intuition about some part of the “real world”.

(February, 2009)
“service S1 is more cohesive than service S2” (implies ordinal scale). Software engineers should be able to reason about cohesion in terms of, for example, “service S1 is \textit{N times more cohesive} than service S2” (implies ratio scale). This is also consistent with the view of Briand et al. [27, 30] who suggest that all structural properties of software should be defined on a ratio scale. As described previously, the metrics derived in this work are defined on ratio and absolute scales.

2.5.2.2 Validation Approaches

There are a number of approaches for the validation of metrics. Some of them are informal and primarily subjective in nature, whilst others have theoretical and axiomatic bases. Axiomatic approaches provide a formal objective framework for comprehensive metrics validation. In contrast, the informal approaches describe some desirable properties of metrics that should also be taken into consideration so as to demonstrate the overall \textit{usefulness} of the metrics, but they are difficult to validate and are typically subjective.

The notion of \textit{usefulness} is important in the context of software engineering where some well-known metrics fail to satisfy the basic requirements of measurement theory, but are still considered to be useful [32]. For example, the original classification of design cohesion proposed by Stevens et al. [222] (described in Section 2.4.3) was meant to be examined on an ordinal scale, but Eder et al. [63] show that the categories are actually defined as the mixture of nominal and ordinal scale types. Therefore, such classification of cohesion should be considered invalid from the measurement theory perspective. However, this classification is widely-accepted since it captures well the intuitive understanding of cohesion in (procedural) software designs. Similar can be said about some of the quantitative OO metrics. For example, an influential and commonly-used suite of OO metrics proposed by Chidamber and Kemerer (CK suite of metrics) [44] has been criticised in terms of its validity from the measurement theory perspective [101, 124].

The following describes some of the major validation approaches and associated validity criteria. The informal approaches are only briefly summarised here since they are not strictly used in this research due to our objective to derive the most formal and unambiguous metrics possible. \textbf{The axiomatic approaches, on the other hand, are discussed in greater detail since they provide foundation for the complete theoretical validation of the derived metrics.}

\textbf{Informal approaches.} There are a number of informal approaches for examining the general validity of metrics. Among those that are widely referenced are: Schneidewind’s metrics validation methodology [213], Henderson-Sellers’ [98] approach, and the ISO/IEC 9126 metrics validity criteria. These approaches define general validity criteria (or desirable properties) for any software metric, with some commonly specified criteria include: i) consistency; ii) discriminative power; iii) repeatability; iv) reproducibility; and v) objectivity (the metrics
should be computed in a precise manner). Additionally, it is suggested that a measure might not be useful when it is used for assessment purposes only, that is, software metrics should be used in predictive models similar to the maintainability prediction model used in this research.

**Axiomatic (formal) approaches.** Axiomatic approaches assist in determining the theoretical and mathematical soundness of a given metric based on its conformance to the formalised intuitive understanding of the attributes under study. More specifically, such approaches define various axioms that are used to validate the homomorphism between the empirical and formal relational systems. That is, the axioms can demonstrate that a given metric really measures the software characteristic it is supposed to measure at the same time conforming to the general principles of measurement theory. For example, any theoretically valid metric should be able to distinguish between two dissimilar entities [204].

A seminal work on the axiomatic validation of software metrics is Weyuker’s Axiomatic Approach [237] that defines nine axioms for the validation of software complexity measures. Given that only the structural properties of coupling and cohesion are investigated in this thesis, Weyukner’s axioms are not covered in this review. There are a number of established approaches for the formal axiomatic validation of metrics that extend the work of Weyukner [237] and can be applied to the coupling and cohesion metrics, including:

- **distance-based software measurement framework** proposed by Poels and Dedene [198]. In this framework, the authors examine the validity of software metrics using the fundamental principles of mathematics, where all metrics are defined as *measures of distance*. According to this purely mathematical definition of a metric, there are four important properties that must be satisfied by the metrics: i) *non-negativity*; ii) *identity*; iii) *symmetry*; and iv) *triangle inequality*. This framework can be considered not suitable for the purpose of detailed and comprehensive validation of the structural software metrics since it does not specify properties for the concrete structural attributes of software (such as coupling and cohesion).

- **coupling axioms** proposed by Fenton and Melton [70]. The authors introduce two generic axioms that should hold for coupling measures. Both axioms assume that coupling is a measure of pair-wise connectivity between modules. The first axiom states that if the only difference between two module structure charts $S$ and $S'$ is an extra interconnection in $S'$, then the coupling of $S'$ is higher than the coupling of $S$. The second axiom states that system coupling should be independent from the number of connected modules in the system. For example, if a module is added to the system and the resultant system shows the same level of *pair-wise coupling*, then the coupling of the system remains the same. The second property is arguable and has been criticised by other researchers since coupling is typically considered to be dependent on the number of connections between modules [33, 102]. Therefore, the decision was made not to use the axioms proposed by Fenton and Melton [70] in this research.
property-based software engineering measurement framework proposed by Briand et al. [30, 32, 34], which is a generic framework that extends the common principles of measurement theory by defining precise mathematical properties that characterise the specific structural attributes of software designs, where design can be viewed as a collection of elements, relations, and binary operations. The framework is unique in a sense that it prescribes mathematical characteristics (or axioms) for all structural properties of software including coupling and cohesion. Additionally, the proposed mathematical characteristics can be applicable to the artefacts defined at the design level as opposed to other frameworks which target implementation level metrics.

The property-based software engineering measurement framework of Briand et al. [30, 32, 34] was chosen for the validation of metrics derived in this research since it is generic and comprehensive allowing the precise characterisation of the structural properties of software designs independently of a specific development paradigm. The generality is supported by the definitions of mathematical characteristics and measurement entities using generic design constructs of modules $m$ and modular systems $MS$. Such constructs can be easily redefined for a particular development paradigm as shown in Chapter 3 of this thesis. The comprehensiveness is supported by the definition of mathematical characteristics for all structural properties of software, and also the applicability of properties to the design-level metrics. Furthermore, the framework has been successfully used by other researchers when validating newly derived metrics [168, 207].

The specific mathematical characteristics from the property-based software engineering measurement framework that relate to coupling and cohesion are shown in Table 2-4. The characteristics are used as the basis for the validation of the metrics derived in this research where a given metric can be deemed valid if it conforms to the prescribed characteristics for the corresponding structural property as shown in Chapters 4 and 5.

Note that the characteristics proposed by Briand et al. [30] hold only when applying the admissible transformations on the ratio scale. This decision to constrain the metrics to a ratio scale was criticised by some leading researchers in the area [126, 250] because such a constraint could be considered as over-restrictive given that it automatically invalidates a large number of existing metrics (Briand et al. [30] demonstrated that most of the existing OO metrics violate the prescribed characteristics). For example, Kitchenham et al. [126] suggest that mathematical characteristics used to define measures should not constrain the scale type of measures. Although Kitchenham et al. use Weyuker’s [237] axioms as the example, the same argument can be applied to the mathematical characteristics proposed by Briand et al. [30].

---

12 Note that Briand et al. refer to the mathematical characteristics (or axioms) as properties. Given that in this thesis the term property refers to the structural properties of software (coupling and cohesion), the term mathematical characteristic is used instead to represent properties of Briand et al. so to avoid any confusion.
<table>
<thead>
<tr>
<th>CHARACTERISTIC</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUPLING.1 Non-negativity</td>
<td>the coupling of module ( m ) in modular system ( MS ) is non-negative</td>
</tr>
<tr>
<td>COUPLING.2 Null Value</td>
<td>the coupling of ( m ) in ( MS ) is null if there are no outgoing or incoming relationships</td>
</tr>
<tr>
<td>COUPLING.3 Monotonicity</td>
<td>adding inter module relationships does not decrease coupling of a module</td>
</tr>
<tr>
<td>COUPLING.4 Merging of Modules</td>
<td>the coupling of ( m ) in ( MS ) obtained by merging two modules is not greater than the sum of the couplings of the two original modules/systems since the two modules may have common inter module relationships [that may disappear after the merge]</td>
</tr>
<tr>
<td>COUPLING.5 Disjoint Module Additivity</td>
<td>the coupling of ( m ) in ( MS ) obtained by merging two disjoint modules is equal to the sum of the couplings of the two original modules/systems</td>
</tr>
<tr>
<td>COHESION.1 Non-negativity and Normalisation</td>
<td>the cohesion of ( m ) in ( MS ) belongs to a specified interval ([0, \text{MAX}]). Normalisation allows meaningful comparisons between the cohesion values obtained for different modules/systems since they all belong to the same interval</td>
</tr>
<tr>
<td>COHESION.2 Null Value</td>
<td>the cohesion of ( m ) in ( MS ) is null if there is no intra-module relationship/s among the elements of a (all) module(s) since there is no evidence that the elements should be encapsulated together</td>
</tr>
<tr>
<td>COHESION.3 Monotonicity</td>
<td>adding intra-module relationship/s does not decrease cohesion since such relationships are supposed to provide additional evidence of the relatedness of system elements</td>
</tr>
<tr>
<td>COHESION.4 Cohesive Modules</td>
<td>the cohesion of ( m ) in ( MS ) obtained by putting together two unrelated modules is not greater than the maximum cohesion of the two original modules/systems</td>
</tr>
</tbody>
</table>

Table 2-4. Coupling and Cohesion properties from property-based software engineering measurement framework ([30] p.76-79)
Additionally, it was suggested that mathematical characteristics cannot be used to adequately define abstract and usually semantic attributes such as coupling and cohesion. - Moraska, Briand, Weyuker, and Zelkowitz responded [167]: “without such characteristics, we end up abstracting away all relevant structure from our model, limiting our ability to say anything of interest” p. 187. Furthermore, the authors [167] state that “an important purpose of using properties (or characteristics) as a means of defining measures is to help codify intuition and make underlying assumptions explicit” p. 188.

As discussed earlier, we believe that the definition of structural software metrics should be done on a ratio (or absolute) scale in order to allow for more comprehensive and detailed examination of the design structure, therefore this ‘ratio-scale constraint’ can be considered beneficial for the purpose of this research.

### 2.5.3 Empirical Evaluation of Metrics

Theoretical validation alone does not imply the overall validity of the metrics. This is because the measurement theory only covers the direct measurement of attributes for the assessment purposes; it does not prescribe rules or axioms for the predictive metrics. To validate the predictive power of metrics, it is also imperative to establish empirically the relationship between the metrics and the quality characteristics they purport to predict [34, 98, 213]. The common way to do so is to establish and statistically test the experimental hypotheses that formalise the relationships between the structural properties of software, as measured by metrics, and the quality characteristics in question. The choice of statistical techniques for analysing the empirical data largely depends on the measurement goals, and more importantly, the mathematical properties of metrics such as the underlying measurement scale. For example, a metric should be defined on at least an interval scale in order to allow for effective use of parametric techniques such as ANOVA test (extension of the t-test), Pearson coefficient, or linear regression.

A number of comprehensive empirical studies have been conducted in order to establish the correlation between OO structural metrics (including coupling and cohesion metrics) and the maintainability of OO software products [20, 29, 53, 146, 170]. The experimental design and associated methods and activities of these studies can be readily replicated when evaluating metrics defined for different development paradigms such as SOC. This is because the study structure and objectives are independent of the particular technology or development paradigm in use, only the independent variables will differ (for example, OO metrics can be substituted with SO metrics).

There are a number of commonalities that can be found in the existing empirical studies:
CHAPTER 2. LITERATURE REVIEW

- All studies show correlation between OO design properties, as measured by metrics, and quality of software products in terms of their maintainability, which is consistent with the results of other similar studies conducted in the context of Procedural and OO software to evaluate various quality characteristics (such as software reliability and reusability) [7, 38, 45, 49, 92, 164, 177]. This suggests that such correlation can also be established for the service-oriented software products, thereby providing rationale for this thesis. Moreover, some of the OO metrics which are shown to influence software maintainability can be used for initial benchmarking and comparison with service-oriented metrics as is done in Chapter 6 of this thesis.

- Two major statistical approaches are commonly used: a standard significance testing of variance (t-test and ANOVA test), and correlation and regression [166]. These approaches are well suited for exploratory research and are commonly used in Software Engineering [35]. Note that correlation analysis allows assessing the degree to which one variable is related to another; whereas regression analysis provides the basis for forecasting the values of a variable from the values of one (simple or univariate regression) or more (multiple or multivariate regression) variables by estimating the parameters of the equation linking them. The significance testing, and correlation and simple linear regression techniques and associated indicators (such as the Pearson coefficient \( r \) which reflects the degree of linear relationship between two variables), are used in this thesis to empirically evaluate the newly derived metrics. The specific techniques are described in more detail in Chapter 6.

- All studies are subject to various threats to validity that limit the generalisation and interpretation of the results. For example, most software systems used in the studies were research prototype systems, which are commonly smaller and less complex than real-life industry systems. Also, the data sets in some of the studies contained small sample sizes, thereby reducing the statistical power and reliability of the results. Such threats to validity are common to most empirical studies in software engineering [116]. The study conducted in this research is also subject to validity threats as discussed further in Chapter 6.

  Given that the existing empirical studies are largely ‘technology independent’, some parts of the associated experimental designs have been adopted in this thesis. The following summarises the related experimental aspects (based on the measurement goals):

  **Briand et al. [29]** investigated the effects of the Procedural and OO design techniques and associated design principles perceived to be ‘good’ and ‘bad’ practices, on the maintainability of software designs. The study design was based on a standard within-subjects model [165], with the experimental material consisting of four different software designs developed with permutation of the design techniques and principles. The within-subjects design was employed in this research since it requires fewer participants (the empirical study presented in this thesis consisted of ten participants only) as discussed further in Chapter 6.
Basili et al. [20] evaluated Chidamber and Kemerer (CK) suite of OO metrics [44] (discussed in Section 2.5.4) using a controlled, group-based study. To counterbalance the differences in skills and experience among the participants when allocating them to study groups: i) the level of experience of each student was characterised at the beginning of the study based on questionnaires and interviews; and ii) the “blocking” procedure [119] was employed to minimise any potential learning effects. The participants’ development experience with various paradigms and general understanding of the principles of SOC was evaluated prior to conducting the empirical study in this research. Furthermore, the initial pre-test programming exercise was completed by the participants as described further in Chapter 6. Finally, a “selective orders” procedure [217] was employed in order to objectively allocate the study participants to the experimental tasks (refer to Section 6.2.4).

Dagpinar et al. [53] investigated which object-oriented metrics can be used as significant predictors of the maintainability of software products by analysing the historical data of maintenance activities collected from the logs of sample OO systems. The maintenance activities were categorised into distinct categories: perfective/adaptive and corrective. The maintenance activities conducted in this research were also categorised into the perfective and corrective types following the typical distribution of maintenance activities described in Section 2.3.2. As with the study of Dagpinar et al. [53] this was done in order to simulate real-life industrial settings.

Finally, note that several researchers have criticised the standards of performing and reporting empirical studies in software engineering [116, 128]. The presentation of the empirical study performed in this research follows a well-defined template for reporting controlled experiments in software engineering proposed by Jedlitschka et al. [116, 117]. The template and associated activities support a systematic and well-structured presentation of empirical experiments, making it easier for the reader to understand the structure of the experiments, and assess the validity of the experimental results. Note that the original template proposed in [116] had some inconsistencies in the review sections as highlighted by Kitchenham et al. [128, 129]. The updated version of the template [117], as used in this thesis, has been restructured by the authors in order to address the identified problems. This template consists of four major sections that will be used as the foundation for the structure of Chapter 6. They are:

- **Experiment planning**: i) Goals; ii) Subjects; iii) Hypotheses and Variables; iv) Experimental material; v) Tasks; vi) Experimental Design; and vii) Execution procedure.

- **Analysis**: i) Analysis procedure; ii) Descriptive statistics; and iii) Hypothesis testing.

- **Discussion**: i) Evaluations of results; ii) Threats to validity; and iii) Future directions.
2.5.4 Existing Metrics

There is a large number of metrics proposed for quantifying various aspects of the structural properties of software in Procedural and OO development [33, 44, 63, 95, 102, 201]. Some of the widely-referenced coupling and cohesion metrics are overviewed in this section. Such metrics can complement the proposed SO metrics since they can be readily used to measure the structural properties of individual service implementation elements (such as OO classes and interfaces) in isolation. The additional analysis of metrics directly related to this research, insofar they contribute to the definitions of some of the metrics proposed in this thesis, is provided in Sections 4.2 and 5.2.

2.5.4.1 Coupling Metrics

In the Procedural paradigm, the well-known approach for quantifying coupling is based on the broad categorisation proposed by Stevens et al. [222] (Section 2.4.2). The process of assigning design artefacts to particular coupling categories has a subjective nature, and thus cannot be automated. In contrast, the common approach to quantify the coupling in OO paradigm is to use objective quantitative metrics that can be easily collected in an automated fashion [32, 44, 102, 146]. Note that existing OO metrics are often expressed in an ambiguous manner which makes it difficult to understand how different metrics relate to one another [33]. Moreover, only selected metrics have been validated theoretically and empirically. Nevertheless, there are a number of well-established OO metrics addressing various aspects of coupling:

- Chidamber and Kemerer proposed Coupling Between Objects (CBO) metric (as part of their highly-referenced suite of OO metrics – CK metrics [43, 44]), which is a count of the number of non-inheritance related couples (interactions) with other classes. An object of a class is said to be coupled to another, if methods of one class use methods or attributes of another class. The direction of the interactions between classes was not considered. In later publication [36], a revised definition was proposed in order to include inheritance-based interactions. Theoretically validated: Yes (using Weykner’s axioms); Empirically evaluated: Yes.

- Chidamber and Kemerer [43, 44] also proposed Response for Class (RFC) metric, which represents a set of methods that can potentially be executed in response to a message received by an object of that class. More specifically, RFC = |RS| where RS is the response set for the class, which can be formally defined as RS={M} \cup \{R_i\}, where \{R_i\} is the set of all methods called by method i; and \{M\} is the set of all methods in the class. Refer to the original publications [43, 44] for the explanation of the above formalism. Also note that RFC can be considered as the measure of a dynamic coupling. Theoretically validated: Yes (using Weykner’s axioms); Empirically evaluated: Yes.
Li and Henry [146] derived Data Abstraction Coupling (DAC) metric, which counts the number of abstract data types (or classes) defined in a given class. An abstract data type is considered to be defined in a class \( c \), if it is the type of an attribute of class \( c \). More specifically, DAC is the number of not inherited attributes that have a class as their type. The authors reasoned that the number of variables having an abstract data type indicates the number of data structures dependent on the other classes, which could potentially influence the maintainability of the system. Theoretically validated: No; Empirically evaluated: Yes.

Martin [154] proposed two coupling metrics: efferent coupling (Ce) and afferent coupling (Ca). The metrics are related to the categories of classes, where a category is a set of classes that belong together because they achieve some common goal (in this sense, a service can be considered as a category of design elements). Ce is defined as the number of classes inside a given category that depend upon classes outside this category. In contrast, Ca is defined as the number of classes outside the category that depend upon classes within a given category. Martin fails to specify precisely what constitutes dependencies between classes and categories. Theoretically validated: No; Empirically evaluated: Partially.

Chen and Torngren [41] derived a suite of metrics that are counted based on a number of weighted characteristics, including i) the topology and multiplicity of class interactions; ii) the replication and frequency of interactions; and iii) the accuracy of component properties that appear in a relationship (interaction). Additionally, the authors described a technique for combining coupling of individual classes into an overall system coupling, where domain specific heuristics and technology constraints are used to determine the weighting. Theoretically validated: No; Empirically evaluated: Partially.

Finally note that a number of metrics have been proposed to quantify various dimensions and types of coupling according to the coupling frameworks discussed in Section 2.4.2. For example, Hall et al. [95] derived metrics for measuring “In coupling”, “Out coupling”, and “Global Connection” coupling categories. Similarly, Hitz and Montazeri [101, 102] and Briand et al. [33] propose metrics for quantifying different aspects of coupling according to their frameworks (refer to Section 2.4.2). Furthermore, there are a number of coupling metrics proposed for highly-specialised areas such as Object Constraint Language (OCL) expressions [202] which are shown to influence the analysability of UML-based models. These metrics are not directly related to this research since they are specific to a particular technological concept (namely OCL) and thus not generic enough to be applicable in the context of different development paradigms such as SOC.
2.5.4.2 Cohesion Metrics

In the Procedural paradigm, the well-known approach for quantifying cohesion is based on the taxonomy of cohesion categories defined by Stevens et al. [222] (refer to Section 2.4.3). The process of assigning design artefacts to particular cohesion categories has a subjective nature (similarly to coupling categorisation described previously), and thus cannot be automated. Therefore, more recent research initiatives have focussed on the definition of quantitative cohesion metrics that support an entirely automated measurement process. The existing OO cohesion metrics can be grouped into different categories based on the underlying measurement procedure as follows:

1) Method->Attribute Accesses

The attribute access related metrics, which represent the most common type of cohesion metrics, are based on the supposition that a given OO class is cohesive if all its attributes are used by all the methods of this class. Such metrics can be applied only at the implementation level because class internals are typically not known at the design stage.

- Chidamber and Kemerer derived Lack of Cohesion in Methods (LCOM) metric [43, 44], which is the most often used and referenced OO cohesion metric to date. LCOM is the number of pairs of methods in a class having no common attribute references (Q), reduced by the number of method pairs referencing at least one shared class attribute (P). LCOM will be set to zero in case |Q| < |P|, where zero indicates good cohesion (LCOM is an inverse measure). This artificial reduction of LCOM to zero has been criticised in the research literature [20, 24, 146]. Theoretically validated: Yes (using Weykner’s axioms); Empirically evaluated: Yes.

- Li and Henry [146] and Hitz and Montazeri [101, 102] redefined the LCOM metric since it was shown that LCOM can be overly-dependent on the total number of methods: i) Li and Henry proposed new definition of LCOM (commonly referred to as LCOM1) defined as the number of disjoint sets of local methods in the class, where no two sets intersect and any two methods in the same set sharing at least one class attribute; and ii) Hitz and Montazeri proposed another extension to LCOM (commonly referred to as LCOM2) in order to include method invocations as the additional indication of cohesiveness. That is, LCOM2 assigns local methods to a given set not only based on the attribute accesses, but also based on the invocation of other methods of the same class. Theoretically validated: No; Empirically evaluated: Yes.

- Bieman and Kang [24] proposed two metrics, Tight Class Cohesion (TCC) and Loose Class Cohesion (LCC), which are related to LCOM and its variations since TCC and LCC also evaluate pairs of methods which use common class attributes. However, indirectly used
attributes are also considered, where method \( m \) uses attribute \( a \) indirectly, if \( m \) directly or indirectly invokes a method \( m' \) which uses attribute \( a \). TCC is defined as the percentage of pairs of methods of the class which are directly connected. LCC is defined as the percentage of pairs of methods of the class which are connected both directly and indirectly. The values for both TCC and LCC will range from 0 (worst cohesion) to 1 (best cohesion). Theoretically validated: Partially; Empirically evaluated: Yes.

- Gui and Scott [93] proposed class cohesion metric that is similar to the LCOM-related metrics, but also takes into consideration the strength of cohesion between methods by assigning a value to each pair of related methods based on the number of instance variables common to these methods. Class cohesion is calculated by dividing the sum of all similarities between methods by the total number of pairs of related methods. System-level cohesion is defined as the mean cohesion of all classes in the system. The authors also present some empirical evidence that the proposed metric was a better predictor of class reusability than the LCOM, LCOM1, LCOM2, and TCC/LCC metrics. Theoretically validated: No; Empirically evaluated: Partially.

2) Method Parameters

The parameters-related metrics are based on the supposition that a class is cohesive when all the methods in this class use the same set of parameter types. Such metrics are applicable to software designs since method interfaces are typically known at the design stage.

- Bansiya et al. [15] propose Cohesion Among Methods in a Class (CAMC) metric that measures the degree of correspondence between the parameter types across each of the methods in an OO class. To compute CAMC for a class with \( n \) methods, the union of parameter types in the method signatures of a class \( T \) is constructed; and a set \( M \) of all parameter object types for each method is constructed. An intersection set \( IS \) of \( M \) with the union set \( T \) is then calculated. Finally, the summed cardinality of all the intersection sets is divided by \( T \) multiplied by \( n \) to derive a final value of CAMC. Theoretically validated: No; Empirically validated: Yes.

- Counsell et al. [50, 51] propose Normalised Hamming Distance (NHD) metric which can be considered as extension of CAMC. NHD quantifies the disagreement between rows in a binary matrix constructed based on the parameter types used by the methods of a class. To calculate NHD, the sum of the disagreements between methods over all parameters is computed and then subtracted from 1. It was empirically shown that both CAMC and NHD correlate strongly with LCOM metric, thereby providing a useful alternative for measuring OO cohesion since they can be computed at the design stage. Theoretically validated: No; Empirically validated: Yes.
3) Program Slices

- Bieman and Ott [22] proposed a set of functional cohesion metrics based on program slices, where slicing represents a method for examining the implementation of software and removing code statements that do not effect a computation of interest. The resulting smaller programs (or slices) can be used to assess the attribute usage patterns and the dependency between parts of code and attributes used. Such approach is implementation dependent and cannot be used to measure design cohesion as was noted by Bieman in his later publication [23]. Theoretically validated: No; Empirically validated: Partially.

Additionally, there are a number of recent and fundamentally different approaches for measuring cohesion. For example, Marcus and Poshyvanyk [150] proposed Conceptual Cohesion of Classes (C3) metric, which measures cohesion based on both structural and syntactic aspects by using natural language processing techniques to extract information from the source code identifiers and comments in order to analyse semantics of the problem domain.

2.5.5 Discussion

As was discussed in Section 2.4.4, the structural properties of software can be measured at different levels of abstraction, ranging from high-level design through to executable implementations, with the target level of abstraction influencing the metrics definition and measurement process. Measuring structural properties of software implementation can result in more accurate measurements compared to measuring properties of designs since more detailed data is available. Nonetheless, the metrics should be collected as early as possible since the sooner problems in the software structure can be identified, the smaller the effort required to correct them. Thus, it is beneficial to use metrics that can be applied early in the SDLC to ensure that software design have favourable structural properties, thereby decreasing the number of software errors (or faults) and allowing the developers to fix problems and remove irregularities in an efficient manner [16, p.4]. The suite of service-oriented coupling and cohesion metrics presented in Chapters 4 and 5 respectively is applicable to low-level designs.

Numerous metrics have been proposed to measure coupling and cohesion of OO software, but as Fenton and Pfleeger [75, p.319] note, there is as yet no common agreement on what should be measured in OO systems and which metrics are appropriate. Furthermore, most of the existing metrics lack formal theoretical validation. As discussed in Section 2.5.2, a metric can be deemed theoretically valid if it has been demonstrated that this metric is indeed measuring the attribute it is purported to measure based on the conformance to the accepted axioms. Such axioms support methodical definition of metrics based on the principles of measurement theory. The derived service-oriented metrics were theoretically validated using property-based software engineering measurement framework proposed by Briand et al. [30].
Although many software metrics have been defined for OO development model, only very few have been proposed for service-oriented systems. Previous research has shown that the use of different development paradigms, such as Procedural design and OO, will result in systems with different structural properties [63, 102] as discussed briefly in Section 2.4.1 and elaborated further in Chapter 3. Accordingly, the existing Procedural and OO metrics are not immediately applicable to the structure of service-oriented designs and development principles introduced by service-orientation as was the case with Procedural metrics being insufficient for the principles of OO [44, 63, 146]. The design of service-oriented systems including various structural characteristics and service-specific relationships is formalised in Chapter 3. Additionally, Chapter 3 discusses the major distinct characteristics of SO designs that differentiate them from previous development models (such as Procedural and OO development).

Furthermore, Perepletchikov et al. [189] conducted an exploratory empirical study in which some of the existing Procedural and OO metrics [44, 156] were unable to differentiate between two Service-Oriented designs that were qualitatively different in terms of logical and physical structure. The systems were developed using two contrasting approaches, where one of the approaches employed coarse-grained services, structured using the principles of OO; and another approach was based on embedding business logic into executable BPEL4WS scripts with the system constructed in terms of fine-grained services. Note that the study only investigated a limited number of metrics, namely six metrics from Chidamber and Kemerer (CK) suite [43, 44], and McCabe’s cyclomatic complexity metric [156] which is one of the most used complexity metric for both Procedural and OO software. As such, it cannot be considered as representative and comprehensive.

Nevertheless, it provided initial empirical evidence suggesting that some of the existing metrics cannot be readily applied to SO systems. The designs used in the study and the resultant metric values are described in [189].

To conclude, there is a need to derive and theoretically validate metrics specific to SOC paradigm. Such metrics should also be evaluated empirically to establish the correlation between service-oriented coupling and cohesion and the maintainability of final software products. To this end, the Maintainability model established in Section 2.3.2 (Figure 2-6) can be extended with the additional constructs as shown in Figure 2-7.
Figure 2-7. Maintainability Model used in this research (updated version)
Chapter 3. Formal Model of Service-Oriented System Design

This chapter presents a formal model of service-oriented design covering structural and behavioural properties of the design artefacts in a service-oriented system. The model extends the generic graph-based model of a software system [30] with the core design characteristics of service-orientation. The main purpose of the model is to allow software metrics related to the structural properties of service-oriented software designs to be:

- *Defined in a precise unambiguous manner* since the entity under study, service-oriented software design, is specified in a formal way.
- *Theoretically validated* using the property-based software engineering measurement framework [30] described in Section 2.5.2 which requires software to be modelled using graph-based abstractions.

Additionally, by formalising various types of service-oriented design relationships, the model simplifies the definition of coupling metrics as described further in Chapter 4.

This chapter is organised as follows. Section 3.1 overviews related work on design modelling. Section 3.2 describes the core design characteristics of service-oriented software that should be captured by the model definitions. The model definitions are then presented and discussed in Section 3.3; and listed in a table in Section 3.4 to enable easier referencing in later chapters. Finally, Section 3.5 summarises the derived model.

### 3.1 Modeling Software Designs

In order to define software metrics in an unambiguous and precise manner, the intuitive understanding of the principles of service-oriented design should be consolidated into a formal model. This is because the abstraction of an entity, such as software design, should be as formal as possible in order to objectively measure its attributes [76].

As described in Section 2.5.1 measurement can be defined as the process by which numbers or symbols are assigned to attributes of entities. Such assignment must preserve any empirical observations about the entities and their attributes, thereby maintaining the homomorphism between the empirical and relational systems as prescribed by the rules of measurement theory. To illustrate the key concepts of measurement theory a simple example of measuring height of a human is shown in Appendix B, where it is discussed that when measuring a height (attribute) of a human (entity), the bigger values must be assigned to the taller humans so to be consistent with our intuitive understanding of the attribute of height. The problem is that an attribute may have a dissimilar intuitive meaning for different people,
making it difficult to establish empirical relationships between the entities. Therefore, there is a need to define a formal model of an entity that will reflect a specific viewpoint [72]. For example, a model of a human might specify a particular type of posture. Once such a model is defined, the consensus can be established regarding empirical and formal relations applicable to humans with respect to their height.

The need for formal modelling is particularly relevant in the area of software measurement, where the structural properties of software are not fully understood or consistently defined. For example, even a presumably well-understood property of size and its associated metrics, such as for example Lines-of-Code (LOC), can have different interpretations, thereby requiring a well defined formal model of software in order to avoid ambiguity [72]. Also, modelling software designs allows emphasising specific structural aspects that are relevant to particular measurement goals [251].

3.1.1 Related Work

The widely-referenced model of a generic software system was defined by Briand et al. [30] using a graph-theoretic approach. The model was used by the authors [30] to support the specification of the mathematical characteristics for the structural properties of software as part of the property-based software measurement framework described in Section 2.5.2.

In this generic model, a software system $S$ is represented as a graph, where vertices symbolise software artefacts (elements) and edges correspond to the relationships between these artefacts. Such a graph can be formally captured as a pair $<E, R>$, where $E$ symbolises the set of elements of $S$, and $R$ is a binary relation on $E$ ($R \subseteq E \times E$) representing the relationships between the elements of $S$. Also, a module $m$ of $S$ was defined as $m = <E_m, R_m>$, where $E_m \subseteq E$, $R_m \subseteq E_m \times E_m$, and $R_m \subseteq R$. The modules can overlap each other and can also be defined at a different level of abstraction, for example an Object-Oriented class vs. a segment of code.

Additionally, the representation of a generic software system was expanded by Briand et al. [30] in order to capture the structure of a modular system. This was done in order to support the specification of the mathematical characteristics for the coupling and cohesion properties of software, which can be investigated only in the context of modular systems [30]. The modular system ($MS$) was defined as $MS = <S, M>$, where $S = <E, R>$ is a generic software system, and $M$ is a collection of disjoint modules $m$ of $S$. For example, $E$ can represent a set of OO methods, and $R$ can represent a set of invocations from one method to another. A module $m$ can then symbolise an OO class in system $MS$.

Figure 3-1 shows a modular software design that can be represented as follows:

$E = \{a, b, c, d, e, f, g, h, i, j\}$;
$R = \{(a, b), (a, c), (a, d), (c, f), (d, f), (d, g), (e, b), (f, i), (g, j), (h, e), (i, h), (i, j)\}$;
$M = \{m1, m2, m3\}$, where each individual module $m_x$ consists of the sub-sets of $E$ and $R$.  

(February, 2009)
Figure 3-1. Design of a modular software system (modified from ([30], p.71))

Note that this generic model was designed to represent the structure of any software system, given that it does not reduce the number of possible system representations because software elements, modules and associated relationships can be defined according to specific technology and/or development paradigms. For example, this model has been successfully used (and also extended) by various researchers in order to unambiguously derive and theoretically validate software metrics based on the specific measurement goals:

- Moraska [168] used the model in its original form when deriving metrics for measuring structural properties of size, length, complexity, and coupling of concurrent software systems that have been expressed by means of Petri nets.

- Rossi and Fernandez [207] modified the original model definitions to represent the structure of a software system composed of cooperating distributed components. This was done in order to formally define a set of design metrics specific to distributed systems. The structural modifications were based on substituting the definition of a set of system elements (E) and modules (M) with the set of system components (O) and component clusters (C) respectively. Also, Rossi and Fernandez introduced additional types of relationships between system elements for capturing some of the behavioural aspects of the distributed systems.

- Allen [3] extended the definition of system abstraction S with additional characteristics in order to model explicitly the lack of relationships between the system and its environment (i.e. a disconnected node that represents the environment was added to the original definition of S). The author also provided separate definitions for software systems that include only inter- or intra-module relationships. These extensions were introduced in order to support derivation of software metrics for measuring the size, length, complexity, coupling, and cohesion of generic software systems.
• Briand et. al [33] also extended the generic model in later research publications in order to formally capture the structure of OO systems based on the specific viewpoints and measurement objectives. For example, to theoretically evaluate existing metrics for measuring coupling in OO systems, the structure of software system was redefined using OO System, OO Classes, and Inheritance Relationships as the key modelling constructs. The authors also included the formal definitions of OO class attributes, methods, and their associated parameters in order to make the model more descriptive. Additionally, in a more recent research publication [10], a definition of the generic structure of software system S was redefined in terms of the sets of OO classes (C), objects (O), methods (M), and lines of code (N) in order to derive metrics for measuring dynamic coupling in OO systems based on runtime object interactions.

Neither the original nor the modified models are directly applicable to service-oriented system designs because:

- They treat applications as a collection of software components independent of specific implementation architecture
- They were defined for a particular development paradigm (such as OO), thereby representing a specific technology-based viewpoint making them inapplicable to the particular characteristics of service-oriented designs.

### 3.2 Fundamental Characteristics of SO System Designs

This section summarises the four important characteristics of service-oriented designs (labelled C1-C4 below) discussed in Section 2.2, which cannot be readily captured by the existing model of a generic software system or specific models reviewed above. These characteristics will be incorporated into the model of a service-oriented design presented in the next section.

**C1. SOC introduces more levels of abstraction compared to other development paradigms**

The Procedural paradigm has only one main level of design abstraction: a *procedure*. The Object-Oriented paradigm operates on two levels of design abstraction, where *OO methods* are encapsulated within *OO classes*.

In contrast, the SOC paradigm introduces a third level of abstraction and encapsulation: a *service*. In service-oriented systems, *operations* (e.g. OO methods) are aggregated into *implementation elements* (e.g. OO classes) that implement the functionality of a *service* as exposed through its *service interface*.

**C2. Implementation of services can be achieved using various platforms and languages**
Service-oriented systems can be implemented using a range of different technologies and development paradigms, which is especially relevant given the application of SOC to integration projects. Previous research has shown that the use of different development models, such as Procedural and OO paradigms, will result in systems with different structural properties [63]. Therefore, to allow for more accurate and detailed modelling of SO designs, different service implementation element types should be treated differently, rather than being combined into one single generic element as was done in [3, 30, 168, 207].

**C3. A service interface is an important first-class design artefact**

Correctly identifying service interfaces is challenging and important service-oriented design activity [57, 94]. This is because interface granularity and relatedness of its operations will strongly influence the structural properties of service-oriented designs as discussed in Section 2.2.2. Moreover, service-oriented systems should be structured in terms of independent, self-contained services, with service interfaces being the primary entry points of a system in order to enforce service autonomy [4, 67, 186]. As such, service interfaces must be highly stable as future changes can potentially affect a large number of clients.

**C4. A service is not an explicit design construct**

In existing implementation technologies, a service boundary is logical rather than physical. Therefore, there is a need to define a concrete procedure for the unambiguous allocation of implementation elements to services in order to determine service boundaries, thereby allowing inclusion of services as first-class design artefacts in the model of SO design. Additionally, identifying a service boundary will allow specifying various types of intra- and inter-service relationships (Section 3.3.2) that can influence the coupling of service-oriented designs as described further in Chapter 4.

### 3.3 Model Definitions

This section presents the model of service-oriented system design, which extends a generic model of a software system (described in Section 3.1.1) by incorporating the fundamental characteristics of service-orientation. In this model, the design of service-oriented system is represented as a bi-directional graph [80] that can be expressed using standard set-theoretic notation. Vertices (V) in this graph symbolise software design artefacts found in service-oriented systems, namely service interfaces and various service implementation elements. Edges (E) correspond to the relationships between these artefacts, representing both structural and behavioural dependencies.

For example, an arbitrary design structure (SOS) illustrated in Figure 3-2 consists of:

- a vertex set \( V(SOS) = \{si1, si2, p1, c1, bp1, h1, i1, c2, p2, c3, c4\} \);
- and an edge set $E(SOS) = \{(si1, p1), (si1, c1), (si2, bp1), (si2, c2) (c1, p1), (p1, c1), (p1, h1), (c1, i1), (p1, p2), (c1, c3), (c2, c4), (c4, c3), (p2, si2)\}$, where an edge with end vertices $x$ and $y$ is denoted by $(x, y)$.

Also, the graph of a service-oriented system can be partitioned into a number of sub-graphs representing individual services in the system as shown in Figure 3-2, where a graph $SOS$ has two marked sub-graphs (services), $ser1$ and $ser2$. For example, service $ser2$ consists of a vertex set $V(ser2) = \{si2, c2, bp1, c4\}$, which is a subset of $V(SOS)$; and an edge set $E(ser2) = \{(si2, bp1), (si2, c2), (c2, c4)\}$, which is a subset of $E(SOS)$.

The formal definitions capturing the design of SO system are presented in three parts to improve readability, with Section 3.3.1 defining design artefacts that constitute service-oriented systems; Section 3.3.2 defining various relationships between these artefacts; and Section 3.3.3 combining definitions from the former two subsections into one complete model. Finally, Section 3.3.4 presents a formalism for representing different types of Service-Oriented system designs based on their conformance to the principles of service-orientation.

### 3.3.1 System Structure

This subsection formally defines the structure of a service-oriented system in terms of its constituent services and associated service interfaces and implementation elements. The notation used in the model definitions can be found in Appendix C.

- **DEFINITION 1 (System structure)**

  The service-oriented system structure (SYS) is composed of the sets of various design artefacts as follows:
  
  i) The concept of a generic design element is subdivided into two distinct design artefacts, a service *implementation element* and a service *interface*, in order to cover design characteristic C1 described in Section 3.2.

  ii) The *implementation element* artefact is further subdivided into more concrete implementation types, namely *Business process scripts*\(^{13}\) (bps), *OO classes* (c), and *Procedural packages*\(^{14}\) (p). These types represent common technologies used to implement service-oriented systems. This was done in order to cover design characteristic C2.

  iii) The *service interface* (si) is defined as a separate design construct in order to cover design characteristic C3. Furthermore, the structural characteristics of interface types (OO interfaces or Procedural packages) are also different from that of concrete implementation types [82, 242]. As a result, the *OO interfaces* (i) and *Package headers* (h) are defined as separate elements in the model.

---

\(^{13}\) For example, WS-BPEL 2.0 scripts (refer to Section 2.2.1)

\(^{14}\) Collection of procedures that can be written in any procedural/structural-based language (such as C).
Formally, a system structure (SYS) can be defined as:

\[
SYS = <SI, BPS, C, I, P, H>
\]\[D1]\[1]

where SI is the set of all service interfaces \(si\) in SYS; BPS is the set of all business process scripts \(bps\) in SYS; C is the set of all OO classes \(c\) in SYS; I is the set of all OO interfaces \(i\) in SYS; P is the set of all procedural packages \(p\) in SYS; and H is the set of all package headers \(h\) in SYS.

**Definition 1.1 (Service structure)**

The sets representing the compositional elements of a service (\(s\)) are subsets of the sets comprising the total elements of the system (SYS), with the exception of the service interface which is a single element because a service has only one service interface.

Formally, a service (\(s\)) can be defined as:

\[
s = <si_s, BPS_s, C_s, I_s, P_s, H_s>\]\[D1.1]\[1]

if and only if \(si_s \in SI \land (BPS_s \subseteq BPS \land C_s \subseteq C \land I_s \subseteq I \land P_s \subseteq P \land H_s \subseteq H) \land (BPS_s \cup C_s \cup I_s \cup P_s \cup H_s \not\owns \emptyset)\).

Note that \(\emptyset\) symbol represents service membership. As was described previously (characteristic C4), a service boundary is logical rather than physical in current implementation technologies. Therefore, the allocation of elements to services is performed by considering the possible call paths in response to invocations of operations exposed in a service interface.

As an example, consider the design shown in Figure 3-3 and Figure 3-4, in which:
i) service interface si2 has a service operation sopA(...) which is realised by the following sequence of calls:  
1) si2.sopA(...) -> c2.opB(...);  
2) c2.opB(...) -> c4.opC(...);

ii) class c4 has operation opC1(...) which invokes operation opD(...) belonging to class c3.

Upon examining the relationships between all system elements (static coupling), without taking into consideration the chain of calls initiated from service interface si2, an element c3 will be allocated to service ser2 (as shown in Figure 3-3) given that elements c4 and c3 are coupled together via service unrelated relationship initiated by the call c4.opC1(...) -> c3.opD(...). Such allocation would be incorrect because element c3 should not be a part of ser2 given that c3 is not reachable through methods invoked on c4 through operation sopA() of interface si2. Figure 3-4 shows the correct assignment of elements to services performed by examining the chain of calls initiated from the service interface si2.

The information required to perform the allocation of design elements to services can be derived from behavioural design artefacts such as sequence or collaboration diagrams, flow charts or data flow diagrams; or by tracing the actual executable code if available. In practice, service unrelated relationships (such as c4.opC1(...) -> c3.opD(...)) would most likely occur when designing a service-oriented systems using a bottom-up approach (refer to Section 2.2.2). Such relationships break the rule of service autonomy and should be avoided [68].

Finally, note that some of the implementation element types could be absent from the system and/or service structure. As a result, the corresponding sets of elements would be empty (indicated by \( \emptyset \)), but the Definitions D1 and D1.1 would still hold. For example, the following is the representation of a service-oriented system SOS and a service ser1 from Figure 3-2, where service ser1 has no elements in the set of Business process scripts (BPS):

\[
\text{SOS} = \langle \text{SI}, \text{BPS}, \text{C}, \text{I}, \text{P}, \text{H} \rangle = \{\text{si1, si2}\}, \{\text{bp1}\}, \{\text{c1, c2, c3, c4}\}, \{\text{i1}\}, \{\text{p1, p2}\}, \{\text{h1}\};
\]

\[
\text{ser1} = \langle \text{si}_{\text{ser1}}, \text{BPS}_{\text{ser1}}, \text{C}_{\text{ser1}}, \text{I}_{\text{ser1}}, \text{P}_{\text{ser1}}, \text{H}_{\text{ser1}} \rangle = \langle \text{si1, \emptyset, \{c1, c3\}}, \{\text{i1}\}, \{\text{p1, p2}\}, \{\text{h1}\}\rangle
\]

To make the model more detailed and descriptive, we now present the definitions of the operations of elements and their associated parameters, and attributes of elements.

\[ \text{DEFINITION 2 (operations of an element)} \]

Design elements can have one or more callable operations, which can be treated generically for all element types and defined formally as:

For each element \( e \in \text{SI} \cup \text{BPS} \cup \text{C} \cup \text{I} \cup \text{P} \cup \text{H} \) let \( \text{Op}(e) \) be the set of operations \( \text{op} \) of element \( e \) \[ \text{[D2]} \]

In addition, operations can be defined individually to cover for the specific element types. For example, operations included in a service interface can be defined as:

For each service interface \( si \in \text{SI} \) let \( \text{SOp}(si) \) be the set of service operations \( \text{sop} \) of service interface \( si \).
DEFINITION 2.1 (operation parameters, return type, and pre- and post-conditions)
- Operations can have (optional) input parameters, which can be formally defined as:
For each operation \( \text{op} \in \text{Op}(e) \) let \( \text{Param}(\text{op}) \) be the set of parameters \( \text{par} \) of \( \text{op} \) [D2.1]

Additionally, parameters can be defined for the specific operation types. For example, parameters of a service interface operation \( \text{sop} \) can be defined as:
For each service operation \( \text{sop} \in \text{SOp}(\text{si}) \) let \( \text{Param}(\text{sop}) \) be the set of parameters \( \text{par} \) of \( \text{sop} \).
- Operations can have (optional) return type, which can be formally defined as:
For each operation \( \text{op} \in \text{Op}(e) \) let \( \text{returnType}_{\text{op}} \) be the return type of \( \text{op} \) [D2.1]
The return types can be defined individually to cover for the specific operation types. For example, the return type of a service interface operation \( \text{sop} \) can be defined as:

For each service operation \( \text{sop} \in \text{SOp(si)} \) let \( \text{returnType}_{\text{sop}} \) be the return type of \( \text{sop} \).

Operations can have (optional) pre- and post-conditions, which can be formally defined as:

For each operation \( \text{op} \in \text{Op(e)} \) let \( \text{Cond(op)} \) be the set of pre- and/or post-conditions \( \text{cond} \) of \( \text{op} \). \[ \text{D2.1} \]

As was the case with the input parameters and return type defined above, the pre- and post-conditions can be re-defined to cover for the specific operation types. For example, pre- and post-conditions of a service interface operation \( \text{sop} \) can be defined as:

For each service operation \( \text{sop} \in \text{SOp(si)} \) let \( \text{Cond(sop)} \) be the set of pre- and post-conditions of \( \text{sop} \).

\[ \text{D2.2} \]

**DEFINITION 2.2** (attributes of an element)

Design elements can have one or more attributes, which can be treated generically for all element types and defined formally as:

For each element \( e \in \text{SI} \cup \text{BPS} \cup \text{C} \cup \text{I} \cup \text{P} \cup \text{H} \) let \( \text{Atr(e)} \) be the set of attributes \( \text{atr} \) of element \( e \).

Additionally, attributes can be defined individually to cover for the specific element types. For example, attributes of an OO class can be re-defined in terms of class variables:

For each OO class \( c \in \text{C} \) let \( \text{Var(c)} \) be the set of member variables \( \text{var} \) of class \( c \).

### 3.3.2 Relationships

This subsection presents the definitions of various types of relationships between service-oriented design elements, where a generic concept of a relationship is described in Figure 3-5. This definition of generic relationship is consistent with the *types of interactions* from the coupling framework proposed by Briand et al. [33] (described in Section 2.4.2), except for the technology-dependent associations that are needed to cover for the inherently diverse nature of service-oriented design elements.

**DEFINITION 3** (relationships between design artefacts in service-oriented systems)

Given that not all combinations of element \( a \) to element \( b \) relationships are technologically achievable, the relationships are described below in terms of the *common, possible, and improbable* sets of service-oriented design relationships. These sets are based on current technological constraints and the experience of the present author, but are not considered definitive and could change in response to changing technology.
CHAPTER 3. FORMAL MODEL OF SERVICE-ORIENTED SYSTEM DESIGN

A relationship is said to exist between two service-oriented design elements a and b
\((a \in SI \cup BPS \cup C \cup I \cup P \cup H)\) and \(b \in SI \cup BPS \cup C \cup I \cup P \cup H)\) if:

i) An operation of a (op \(\in Op(a)\)) invokes an operation defined in b (op \(\in Op(b)\))

ii) An operation of a (op \(\in Op(a)\)) references an attribute defined in b (atr \(\in Atr(b)\))

iii) An element a is the type of an attribute of element b

iv) An element a is the type of a parameter of an operation defined in b
(par \(\in Param(op \in Op(b))\))

v) An element a is mapped to element b via technology-dependent association. For example, two OO classes related through OO inheritance, or wsdl-based service interface operation mapped to a business process script via middleware support

Furthermore, if b is also related to a according to the above, this is considered to be a separate relationship.

Figure 3-5. Definition of a Service-Oriented design relationship

Common relationships (\(R_c\)) represent relationships that are likely to occur in all service-oriented systems, in which collaboration between software elements is done either through a service interface or directly between implementation elements belonging to the same development paradigm. For example an OO class (c) invoking another OO class (c) directly (class-to-class CC relationships), or through an OO interface (CI) can be considered as a common relationship since the elements involved in the relationship belong to the same paradigm.

This set of common relationships can be formally defined as:

\[ R_c = <CSI \cup SIC \cup CC \cup CI \cup IC \cup II \cup PSI \cup SIP \cup PP \cup PH \cup HH \cup BPSSI \cup SIBPS \cup BPSBPS>, \]

where CSI \(\subseteq C \times SI\), SIC \(\subseteq SI \times C\), CC \(\subseteq C \times C\), CI \(\subseteq C \times I\), IC \(\subseteq I \times C\), II \(\subseteq I \times I\), PSI \(\subseteq P \times SI\), SIP \(\subseteq SI \times P\), PP \(\subseteq P \times P\), PH \(\subseteq P \times H\), HH \(\subseteq H \times H\), BPSSI \(\subseteq BPS \times SI\), SIBPS \(\subseteq SI \times BPS\), BPSBPS \(\subseteq BPS \times BPS\).15

For example, a set of relationships CI representing subset of all OO classes to interfaces relationships (C \(\times I\)) for system SOS would be CI \(=\{(c1, i)\}\) in the design shown in Figure 3-2, where each single relationship is represented as the ordered pair (source, destination).

Possible relationships (\(R_p\)) represent relationships which are technology (or paradigm) dependent; insofar as the design elements collaborate with the elements belonging to a different development paradigm. For example a function (procedure) within a Procedural package (p) is called from a method of an OO class (c) via a native interface. The set of possible relationships can be formally defined as:

15 the \(\times\) symbol represents a Cartesian product between two given sets (refer to Appendix D)

73

(February, 2009)
\[ R_p = \langle CP \cup PC \cup CH \cup CBPS \cup BPSC \cup BPBS \cup BPSP \cup BPSH \cup PI \rangle, \]
where
\[ CP \subseteq C \times P, \quad PC \subseteq P \times C, \quad CH \subseteq C \times H, \quad CBPS \subseteq C \times BP, \quad BPSC \subseteq BP \times C, \quad BPBS \subseteq BP \times P, \quad BPSH \subseteq BP \times H, \quad PI \subseteq P \times I \]

**Improbable (technology dependent) relationships** \((R_i)\) represent relationships that are considered to be improbable within the logical and current technological constraints of a service-oriented system. For example, a WSDL-based service interface \((si)\) cannot call another service interface (or other explicit interface types such as OO interface \((i)\) or Package header \((h)\)) directly, as this would be done through a separate implementation element. Also, a Package header \((h)\) can be related to other headers only (via "includes" relationships), but cannot be coupled directly to other implementation elements. Finally, it is impossible to have a relationship from an OO interface \((i)\) to the elements belonging to different development paradigms such as Procedural packages \((p)\) and headers \((h)\), and Business Process Scripts \((bps)\).

For completeness, the improbable relationships are defined below:
\[ R_i = \langle SISI \cup SII \cup ISI \cup SIH \cup HS \cup HCI \cup HIBP \cup HIBP \cup HI \cup HIBP \cup IBPS \rangle, \]
where
\[ SISI \subseteq SI \times SI, \quad SII \subseteq SI \times I, \quad ISI \subseteq I \times SI, \quad SIH \subseteq SI \times H, \quad HS \subseteq H \times SI, \quad HP \subseteq H \times P, \]
\[ HC \subseteq H \times C, \quad HI \subseteq H \times I, \quad HBPS \subseteq H \times BP, \quad IH \subseteq I \times HI, \quad IP \subseteq I \times P, \quad IBPS \subseteq I \times BP \]

The set of overall relationships in a service-oriented system design can therefore be represented as a union of all *common* \((R_c)\) and *possible* \((R_p)\) relationships between various design elements. This overall set of relationships \((R)\) is formally defined as:
\[ R = R_c \cup R_p \quad \text{[D3]} \]

::: DEFINITION 3.1 (relationships between design artefacts belonging to a service)

The set representing the relationships belonging to a particular service \(s\) is the subset of the overall set of relationships \((R)\). This subset includes relationships between elements belonging to a particular service and can be formally defined as:
\[ R_s = R_{cs} \cup R_{ps} \quad \text{[D3.1]} \]
if and only if \(R_{cs} \subseteq R_c, \ R_{ps} \subseteq R_p, \) and \(R_c \cup R_p = R\) according to Definition D3.

### 3.3.2.1 Service-Oriented Static Relationship Types

The following definitions cover the design relationships from the perspective of a service by addressing various *intra-* and *extra-service* relationship types that can influence the structural properties of service-oriented software designs. Such relationship types encapsulate important design principles of service-orientation. For example, the direct extra-service relationships from one implementation element to another (IR and OR relationships specified in Defini-
tions D4.3 and D4.4) should be avoided given that SO systems should be structured in terms of independent services, where all inter-service interactions are performed strictly via service interfaces (design characteristic C3 from Section 3.3). The structural service relationship types are defined formally below in order to provide the foundation for the coupling metrics presented in Chapter 4, with the formal definitions of such metrics being based on the definitions of the structural relationships.

Note that the following relationship types can be considered as coupling relationships according to the coupling framework of Briand et al. [33] adopted in this research and described in Section 2.4.2. Similarly, the service relationships defined below cover the locus of impact aspect [33] (import or export coupling) with incoming and outgoing relationships covered separately.

Box DEFINITION 4.1 (relationships between a service interface and service implementation elements)

The set of direct service interface to implementation relationships IIR(s), which represents the relationships between a service interface $si_s$ and the implementation elements $e$ of service $s$, is formally defined as:

$$IIR(s) = \{(si_s, e) \in R_s | R_s \subseteq (SIBPS \cup SIC \cup SIP) \land si_s \in SI \land e \in (BPS_s \cup C_s \cup P_s)\} \quad [D4.1]$$

For example, the IIR set for service $ser1$ shown in Figure 3-6 is:

$$IIR(ser1) = \{(si1, c1), (si1, p1)\}.$$  

Note that as previously described in Definition D3, a service interface cannot be connected directly to other types of explicit interfaces (such as OO interface or Package header) due to technological constraints. Therefore, these relationships are not included in the definition of IIR(s).

Box DEFINITION 4.2 (relationships between service implementation elements)

The set of internal service relationships ISR(s), which represents the interconnection of implementation elements $e_1$ and $e_2$ belonging to service $s$ can be formally defined as:

$$ISR(s) = \{(e_1, e_2) \in R_s | R_s \subseteq (CC \cup CI \cup IC \cup II \cup PP \cup PH \cup HH \cup BPSBP \cup CP \cup PC \cup CH \cup CBPS \cup BPSC \cup BPSI \cup PBPS \cup BPSP \cup BPSH \cup PI) \land e_1, e_2 \in (BPS_s \cup C_s \cup I_s \cup P_s \cup H_s)\} \quad [D4.2]$$

For example, the ISR set for service $ser1$ shown in Figure 3-6 is:

$$ISR(ser1) = \{(c1, p1), (p1, c1), (c1, i), (i, c2), (p1, h), (p2, h)\}.$$
DEFINITION 4.3 (relationships between the service implementation elements of a given service and the elements belonging to the rest of the system - incoming)

The implementation elements $e_1$ belonging to the rest of the system are connected to the implementation elements $e_2$ belonging to a particular service $s$ via incoming relationships $IR(s)$ defined formally as:

$$IR(s) = \{ (e_1, e_2) \in R_s \mid R_s \subseteq (CC \cup CI \cup IC \cup II \cup PP \cup PH \cup HH \cup BPSBPS \cup CP \cup PC \cup CH \cup CBPS \cup BPSC \cup BPSI \cup PBPS \cup BPSP \cup BPSH \cup PI) \land e_1 \in \text{BPS-BPS}_s \cup \text{C-C}_s \cup I - I_s \cup P - P_s \cup H - H_s) \land e_2 \in \text{BPS}_s \cup \text{C}_s \cup I_s \cup P_s \cup H_s) \} \quad [D4.3]$$

For example, the IR set for service $ser1$ shown in Figure 3-6 is: $IR(ser1) = \{(c3, c1)\}$.

DEFINITION 4.4 (relationships between the service implementation elements of a given service and the elements belonging to the rest of the system - outgoing)

The implementation elements $e_1$ belonging to a particular service $s$ are connected to the implementation elements $e_2$ belonging to the rest of the system by outgoing relationships $OR(s)$ defined formally as:

$$OR(s) = \{ (e_1, e_2) \in R_s \mid R_s \subseteq (CC \cup CI \cup IC \cup II \cup PP \cup PH \cup HH \cup BPSBPS \cup CP \cup PC \cup CH \cup CBPS \cup BPSC \cup BPSI \cup PBPS \cup BPSP \cup BPSH \cup PI) \land e_1 \in \text{BPS-BPS}_s \cup \text{C-C}_s \cup I - I_s \cup P - P_s \cup H - H_s) \land e_2 \in \text{BPS-BPS}_s \cup \text{C-C}_s \cup I - I_s \cup P - P_s \cup H - H_s) \} \quad [D4.4]$$

For example, the OR set for service $ser1$ shown in Figure 3-6 is: $OR(ser1) = \{(c2, c4)\}$.
CHAPTER 3. FORMAL MODEL OF SERVICE-ORIENTED SYSTEM DESIGN

- **DEFINITION 4.5** (relationships between service implementation elements of a service and other service interfaces - incoming)

The service interface $si$ (of a service $s$) is connected to other elements $e$ in the system by service incoming relationships $SIR(s)$ defined formally as:

$$SIR(s) = \{(e, si) \in R_s \mid R_s \subseteq (BPSSI \cup CSI \cup PSI) \land e \in (BPS - BPS \cup C - CS \cup P - PS) \land si = si_s \land si \in SI\} \quad [D4.5]$$

For example, the SIR set for service $ser1$ shown in Figure 3-6 is: $SIR(ser1) = \{(c4, si1)\}$.

- **DEFINITION 4.6** (relationships between service implementation elements of a service and other service interfaces - outgoing)

The implementation elements $e$ (of a service $s$) are connected to other services in the system (strictly through service interfaces $si$) by service outgoing relationships $SOR(s)$ defined formally as:

$$SOR(s) = \{(e, si) \in R_s \mid R_s \subseteq (BPSSI \cup CSI \cup PSI) \land e \in (BPS_s \cup CS_s \cup PS_s) \land si \neq si_s \land si \in SI\} \quad [D4.6]$$

For example, the SOR set for service $ser1$ shown in Figure 3-6 is: $SOR(ser1) = \{(p2, si2)\}$.

### 3.3.2.2 Service-Oriented Dynamic Relationship Types

This subsection defines dynamic relationships present in service-oriented design structures, where a dynamic relationship represents the runtime collaboration between multiple elements in response to a specific operation invocation.

- **DEFINITION 5** (direct collaboration relationships between service-oriented design entities)

To capture the dynamic aspects of service structures, a concept of a collaboration ($c_o$) was introduced. A collaboration $c_o$ captures elements that interact in order to achieve some desired functionality in response to all possible invocations of operation $o$ belonging to some element $e$. Formally:

$$c_o \in CO(e) = \langle Param(o \in Op(e)), CS \rangle \quad [D5]$$

where $Param(o \in Op(e))$ represents parameters to the operation $o$ belonging to set of operations $Op(e)$ of element $e$ as per Definitions D2 and D2.1; $CO(e)$ is a set of all collaborations of element $e$; and $CS$ is the set of collaboration sequences ($cs_{o \in Op(e)}$). A collaboration sequence captures the set of interacting elements that achieve functionality exposed in operation $o$ based on specific inputs (i.e. parameter values) and can be defined as:

$$cs_{o \in Op(e)} \in CS(e) = \langle SI_{cs}, BPS_{cs}, C_{cs}, I_{cs}, P_{cs}, H_{cs} \rangle \quad [D5.1]$$


(February, 2009)
where \( SI_{cs} \subseteq SI, BPS_{cs} \subseteq BPS, C_{cs} \subseteq C, I_{cs} \subseteq I, P_{cs} \subseteq P, H_{cs} \subseteq H \). This represents the set of interacting elements that achieve functionality exposed in operation \( o \) based on specific inputs. In terms of graph theory notation [80], collaboration sequence \( cs_{o \in O(e)} \) represents an open or closed walk starting at element \( e \).

**Definition 5.1** (indirect collaboration relationships between service-oriented design entities)

Additionally, a concept of an indirect collaboration (\( ic_o \in ICO(e) \)) was introduced in order to capture the indirect collaboration sequences (\( ics_{o \in O(e)} \in ICS(e) \)) that include indirectly connected elements determined based on the overall static coupling disregarding whether the elements are interacting to achieve some specific functionality (as was described in Section 3.3.1 using collaboration sequences initiated by the service interface operation as an example). Note that the definitions of \( ic, ICO, ics, \) and \( ICS \) are the same as the ones for \( c, CO, cs, \) and \( CS \), only the semantic rules for assigning the elements to collaborations are different (i.e. elements will be included in the indirect collaboration as long as they are connected via any of the previously defined relationship types). Formally:

\[
\begin{align*}
   ic_o & \in ICO(e) = \langle \text{Param}(op \in Op(e)), ICS \rangle \quad \text{[D5.2]} \\
   ics_{o \in O(e)} & \in ICS(e) = \langle SI_{cs}, BPS_{cs}, C_{cs}, I_{cs}, P_{cs}, H_{cs} \rangle \quad \text{[D5.3]}
\end{align*}
\]

Not that defining direct collaborations allows formally specifying the service membership operation \( \langle \rangle \) (Section 3.3.1). For example, an element \( e \) is said to be a member of service \( s \) if and only if \( e \) belongs to some collaboration sequence \( cs \in CS \) as part of collaboration \( c = \langle \text{Param}(so \in SO(si_s)), CS \rangle \).

### 3.3.3 Combined Structure and Relationships

This section presents a complete model by combining the definitions of system elements and relationships from Sections 3.3.1 and 3.3.2. Additionally, it defines key set-theoretic operations (such as inclusion, union, and intersection) that have to be defined in order to support the theoretical validation of metrics in Chapters 4 and 5.

**Definition 6** (SO System and Service)

A service-oriented system \( SOS \) consists of a number of design elements and associated relationships and can be formally defined as:

\[
SOS = \langle SI, BPS, C, I, P, H, R \rangle \quad \text{[D6]}
\]

Given a system (\( SOS \)), a service \( ser \) can be formally defined as:

\[
ser = \langle si_{ser}, BPS_{ser}, C_{ser}, I_{ser}, P_{ser}, H_{ser}, R_{ser} \rangle \quad \text{[D6.1]}
\]
is a service of SOS if and only if $s_{ser} \in SI \land (BPS_{ser} \subseteq BPS \land C_{ser} \subseteq C \land I_{ser} \subseteq I \land P_{ser} \subseteq P \land H_{ser} \subseteq H) \land R_{ser} \subseteq R \land R_{ser} \subseteq (IIR (ser) \cup ISR(ser) \cup IR(ser) \cup OR(ser) \cup SIR(ser) \cup SOR(ser)) \land (BPS_{ser} \cup C_{ser} \cup I_{ser} \cup P_{ser} \cup H_{ser} \subseteq ser)$

Given the above definitions, the inclusion, union and intersection set operations\(^16\) for services can be defined as follows:

- **Inclusion**: service $s = <s_{ls}, BPS_s, C_s, I_s, P_s, H_s, R_s>$ is said to be included in service $t = <s_{lt}, BPS_t, C_t, I_t, P_t, H_t, R_t>$ (notation $s \subseteq t$) if $BPS_s \subseteq BPS_t \land C_s \subseteq C_t \land I_s \subseteq I_t \land P_s \subseteq P_t \land H_s \subseteq H_t \land R_s \subseteq R_t$

- **Union**: The union of services $s = <s_{ls}, BPS_s, C_s, I_s, P_s, H_s, R_s>$ and $t = <s_{lt}, BPS_t, C_t, I_t, P_t, H_t, R_t>$ (notation $s \cup t$) is the service $st = <s_{lst}, BPS_{st}, C_{st}, I_{st}, P_{st}, H_{st} \cup H_s \cup H_t, R_{st}, R_s \cup R_t>$, where service interface $s_{lst}$ contains operations from both $s_{ls}$ and $s_{lt}$

- **Intersection**: The intersection of services $s = <s_{ls}, BPS_s, C_s, I_s, P_s, H_s, R_s>$ and $t = <s_{lt}, BPS_t, C_t, I_t, P_t, H_t, R_t>$ (notation $s \cap t$) is the service $st = <s_{lst}, BPS_{st}, C_{st}, I_{st}, P_{st}, H_{st}, R_{st}, R_s \cap R_t>$, where interface $s_{lst}$ contains only operations that can be supported by the intersected elements originally belonging to services $s$ and $t$.

Furthermore, to accommodate definition of metrics in Chapters 4 and 5 and to formalise some of the important characteristics of software services, the empty, disjoint, composite and atomic services can be defined as follows:

- **Empty service**: service $s = <\emptyset, \emptyset>$ (notation $\emptyset$) is the empty service

- **Disjoint services**: services $s$ and $t$ are said to be disjoint if $s \cap t = \emptyset$

- **Composite service**: service $s$ with $SOR (s) \cup OR (s) \neq \emptyset$ is said to be a composite service

- **Atomic service**: service $s$ with $SOR (s) \cup OR (s) = \emptyset$ is said to be an atomic service

### 3.3.4 Different Types of SO Systems

This sub-section defines service-oriented systems in the context of a modular design [30]. Additionally, the definitions are separated into specific types of service-oriented system designs based on the conformance of a given system to the key structural principles of SOC, service encapsulation and autonomy (described in Section 2.4.2). That is, we introduce formally a new structural (coupling) design property specific to service-oriented paradigm, service-autonomy, which is based on the conformance of the system design to the principles of

\(^16\) The set operations are used in the theoretical validation of metrics based on the property-based software engineering measurement framework of Briand et al. [30].

(February, 2009)
SOC in terms of structuring the software system as a collection of services where all implementation elements belong to one and only one service.

**Definition 7** (Partitioned SO System)

A system that is entirely partitioned into services (i.e. there exist no implementation elements that do not belong to a service) is considered a partitioned service-oriented system (PARSOS). Formally,

\[
\text{PARSOS} = \langle \text{SOS}, \text{SER} \rangle
\]

is a partitioned service-oriented system, if and only if

- \(\text{SOS} = \langle \text{SI}, \text{BPS}, \text{C}, \text{I}, \text{P}, \text{H}, \text{R} \rangle\) is a service oriented system as per Definition D6;
- \(\text{ser} = \langle \text{si}_{\text{ser}}, \text{BPS}_{\text{ser}}, \text{C}_{\text{ser}}, \text{I}_{\text{ser}}, \text{P}_{\text{ser}}, \text{H}_{\text{ser}}, \text{R}_{\text{ser}} \rangle\) is a service of \(\text{SOS}\) (Definition D6.1);
- \(\text{SER}\) is a collection of services \(\text{ser}\) such that:
  \[
  \forall \text{bps} \in \text{BPS} (\exists \text{ser} \in \text{SER} (\text{bps} \in \text{BPS}_{\text{ser}})) \land \forall \text{c} \in \text{C} (\exists \text{ser} \in \text{SER} (\text{c} \in \text{C}_{\text{ser}})) \land \\
  \forall \text{i} \in \text{I} (\exists \text{ser} \in \text{SER} (\text{i} \in \text{I}_{\text{ser}})) \land \forall \text{p} \in \text{P} (\exists \text{ser} \in \text{SER} (\text{p} \in \text{P}_{\text{ser}})) \land \\
  \forall \text{h} \in \text{H} (\exists \text{ser} \in \text{SER} (\text{h} \in \text{H}_{\text{ser}}))
  \]

**Definition 7.1** (Pure SO System)

A system that is partitioned into a set of services, where: i) every implementation element is part of one and only one service (i.e. all services in the system are disjoint); ii) all inter-service interactions are performed strictly via service interfaces; is considered to be a pure service-oriented system (PURSOS). Formally,

\[
\text{PURSOS} = \langle \text{SOS}, \text{SER} \rangle
\]

is a pure service-oriented system, if and only if

- \(\text{SOS} = \langle \text{SI}, \text{BPS}, \text{C}, \text{I}, \text{P}, \text{H}, \text{R} \rangle\) is a service oriented system (Definition D6);
- \(\text{ser} = \langle \text{si}_{\text{ser}}, \text{BPS}_{\text{ser}}, \text{C}_{\text{ser}}, \text{I}_{\text{ser}}, \text{P}_{\text{ser}}, \text{H}_{\text{ser}}, \text{R}_{\text{ser}} \rangle\) is a service of \(\text{SOS}\) (Definition D6.1);
- \(\text{SER}\) is a collection of services \(\text{ser}\) such that:
  \[
  \forall \text{bps} \in \text{BPS} (\exists \text{ser} \in \text{SER} (\text{bps} \in \text{BPS}_{\text{ser}})) \land \forall \text{c} \in \text{C} (\exists \text{ser} \in \text{SER} (\text{c} \in \text{C}_{\text{ser}})) \land \\
  \forall \text{i} \in \text{I} (\exists \text{ser} \in \text{SER} (\text{i} \in \text{I}_{\text{ser}})) \land \forall \text{p} \in \text{P} (\exists \text{ser} \in \text{SER} (\text{p} \in \text{P}_{\text{ser}})) \land \\
  \forall \text{h} \in \text{H} (\exists \text{ser} \in \text{SER} (\text{h} \in \text{H}_{\text{ser}})) \land \\
  \forall \text{ser}_i, \text{ser}_j \in \text{SER} (\text{ser}_i \cap \text{ser}_j = \emptyset) \land \\
  \forall \text{ser} \in \text{SER} (\text{OR(ser)} \cup \text{IR(ser)} = \emptyset).
  \]

Figure 3-7 and Figure 3-8 illustrate examples of PARSOS and PURSOS system types respectively. For example, ‘Academic Management System’ shown in Figure 3-8 is an example of a pure service-oriented system (PURSOS), where the system consists of nine fully independent services that communicate with one another strictly via service interfaces.
In contrast, the design shown in Figure 3-7 cannot be considered as PURSOS type, insofar as service elements are directly connected to the elements of other services (via OR or IR relationships), and some of the services share implementation elements. Nonetheless, this design can be considered as PARSOS since the entire system is partitioned into services.
3.4 Definitions Listing

Table 3-1 lists all definitions presented in this chapter. These definitions will be used in Chapters 4 and 5 when providing the formal definitions for the coupling and cohesion metrics derived in this research. To this end, Table 3-1 will be utilised in Chapters 3 and 4 for the cross-referencing purposes.

<table>
<thead>
<tr>
<th>NUMBER</th>
<th>SYMBOL</th>
<th>NAME</th>
<th>SECTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>SYS</td>
<td>structure of service-oriented system</td>
<td>3.3.1</td>
</tr>
<tr>
<td>D1.1</td>
<td>s</td>
<td>structure of a service of SYS</td>
<td>3.3.1</td>
</tr>
<tr>
<td>D2</td>
<td>op</td>
<td>operations of an element of SYS</td>
<td>3.3.1</td>
</tr>
<tr>
<td>D2.1</td>
<td>Param returnType Cond</td>
<td>input parameters, return type, and pre- and post-conditions of op</td>
<td>3.3.1</td>
</tr>
<tr>
<td>D2.2</td>
<td>atr</td>
<td>attributes of an element of SYS</td>
<td>3.3.1</td>
</tr>
<tr>
<td>D3</td>
<td>R</td>
<td>overall set of possible and common relationships</td>
<td>3.3.2</td>
</tr>
<tr>
<td>D3.1</td>
<td>Rs</td>
<td>generic relationships between service elements</td>
<td>3.3.2</td>
</tr>
<tr>
<td>D4.1</td>
<td>IIR(s)</td>
<td>service interface to implementation relationships</td>
<td>3.3.2.1</td>
</tr>
<tr>
<td>D4.2</td>
<td>ISR(s)</td>
<td>internal service relationships</td>
<td>3.3.2.1</td>
</tr>
<tr>
<td>D4.3</td>
<td>IR(s)</td>
<td>incoming relationships</td>
<td>3.3.2.1</td>
</tr>
<tr>
<td>D4.4</td>
<td>OR(s)</td>
<td>outgoing relationships</td>
<td>3.3.2.1</td>
</tr>
<tr>
<td>D4.5</td>
<td>SIR(s)</td>
<td>service incoming relationships</td>
<td>3.3.2.1</td>
</tr>
<tr>
<td>D4.6</td>
<td>SOR(s)</td>
<td>service outgoing relationships</td>
<td>3.3.2.1</td>
</tr>
<tr>
<td>D5</td>
<td>c</td>
<td>run-time collaboration</td>
<td>3.3.2.2</td>
</tr>
<tr>
<td>D5.1</td>
<td>cs</td>
<td>collaboration sequence</td>
<td>3.3.2.2</td>
</tr>
<tr>
<td>D5.2</td>
<td>ic</td>
<td>indirect run-time collaboration</td>
<td>3.3.2.2</td>
</tr>
<tr>
<td>D5.3</td>
<td>ics</td>
<td>indirect collaboration sequence</td>
<td>3.3.2.2</td>
</tr>
</tbody>
</table>
There are a number of other formalisms proposed to model various aspects of SOA and SOC. These include: i) approaches that propose formalisms for capturing the semantics of service interfaces (such as OWL-S [152]) in order to assist in dynamic discovery, binding, and orchestration of services: ii) models that cover communicational and collaborative aspects of services using formal representations based on Petri-Nets\textsuperscript{17} [155] or finite state automata [21]; and iii) approaches that model (web) services using formal behavioural contract relationships between service components [149]. Such models, being defined at the architectural level, are not concerned with the design and implementation of service-oriented software, treating services as "black boxes" or nodes in a workflow. As a result, these models cannot be readily used to capture the structural properties of SO designs.

In contrast, the model derived in this research is designed to capture the structure of SO system designs in order to support the evaluation of various structural design properties using software metrics. More specifically, the derived model has the following advantages:

i) it represents software designs using graph-based abstractions, which are highly accepted constructs for depicting software designs and are widely used in software engineering. The graph-based abstraction of software design allows the application of set-theoretic operations in order to theoretically validate the derived metrics.

ii) it is flexible since the model does not enforce any particular software implementation. That is, the model was designed to be as generic and technology agnostic as possible. For example, an OO class element is independent of a particular implementation language (such as Java or C++). Moreover, the service implementation element types included in the model can be used to represent other similar technological implementations. For example, the concept of a software component such as an Enterprise Java Bean (EJB) or CORBA component is not included in the model since it can be represented as a combination of OO interfaces and

\textsuperscript{17} Petri-Nets represent a widely-used graph-based formalism for specifying concurrent systems [13, 14].
classes. The same applies to other popular implementation technologies such as scripting languages, since they can be readily classified as OO or Procedural implementations.

iii) it can be readily customised to support specific technologies, as was shown by Perepletchikov et al. [193, 195] who tailored the model for the specific case of BPEL4WS. For example, a business process script \( bps \in BPS \) was redefined as: \( bps = <VAR, ACT, PL> \), where \( VAR \) is the set of XML variables, \( ACT \) is the set of business process activities, and \( PL \) is the set of partner links (treated as a specialised case of service outgoing relationships SOR (D4.6)).

iv) it captures the practical design aspects of SOC. More importantly in the context of this thesis, the model covers these design aspects that are deemed to influence the structural properties of SO designs investigated in this research (coupling and cohesion). For example, formal definitions of different types of design relationships (D4.1 - D5.3) support unambiguous and formal definition of SO coupling metrics in Chapter 4. To this end, the model can be treated as a supporting mechanism for maintaining the homomorphism between the empirical and formal relational systems (refer to Section 2.5.2 and/or Appendix B).

Finally, the model was successfully used to generate formal representations of the design structures of service-oriented systems as part of the empirical study conducted in this research. Such representations were measured using the derived metrics in order to assess the structural properties of the corresponding software designs. This shows the practical applicability of the model.
Chapter 4. Service-Oriented Coupling Metrics

This chapter presents a suite of metrics for quantifying the structural property of coupling in service-oriented software designs. These metrics, combined with the cohesion metrics presented in Chapter 5, constitute the main contribution of this thesis. The derived metrics are designed to predict the maintainability of final software products according to a number of proposed SO coupling assumptions that establish explicit links between the different aspects of service-oriented structural design coupling and the sub-characteristics of maintainability.

This chapter is organised as follows. Section 4.1 provides an overview of the derived coupling metrics and describes the purpose of the coupling assumptions presented in Section 4.2. The metrics themselves are defined and theoretically validated in Section 4.3, with the theoretical validation summarised in Section 4.4. Finally, Section 4.5 summarises the findings presented in this chapter.

4.1 Overview

The structural property of coupling can be quantified at different levels of product abstraction (refer to Section 3.4). For example, most of the existing OO metrics measure coupling at the design-level [32, 44, 102, 146] because the sooner problems in the software structure can be identified, the lesser the effort required to fix them [251]. Accordingly, the metrics presented in this chapter measure the coupling between artefacts of service-oriented software design as captured by the formal model presented in Chapter 3.

The formal model defined all possible coupling relationships present in service-oriented (SO) system designs, thereby providing basis for unambiguous and formal definition of coupling metrics. That is, the model covers all key structural design characteristics of SOC and formalises relationship types considered to be important in the context of the service-oriented paradigm. The metrics in turn quantify these relationship types, with each distinct relationship (Chapter 3. Table 3.1: D3 and D4.1 - D4.6) having an associated metric to measure it. More specifically, two distinct types of relationships present in SO designs are covered:

i) general (or common) relationships between any two design artefacts (for example, OO class (C) to OO class (C) CC relationship) as part of the overall set of possible and common relationships $R$ (Table 3.1 – D3) disregarding the specific design constraints introduced by SOC;

ii) service-oriented specific design relationships (Table 3.1 – D4.1-D4.6) that address various intra- and extra-service couples, thereby encapsulating fundamental design principles of service-orientation as described in Section 3.3.2.
In addition to measuring the coupling based on the type and number of relationships between artefacts in SO designs, the metrics presented in Section 4.3 cover the newly introduced dimension of service-oriented coupling, namely service-autonomy (defined in Section 3.3.4). Service autonomy reflects the conformance of the system design to the core principles of service-orientation in terms of structuring the software system as a collection of services where all implementation elements belong to one and only one service. Although the modularisation of a system into services cannot be considered as ‘traditional’ coupling (since coupling is typically measured based on the relationships between the elements of the system [33, 63]), the decision was made to include service-autonomy as specific type of SO design coupling. This is because the concept of service-autonomy is related to the communicational (or coupling) structural design aspects, and as such, can be used as a quantifiable indicator of the degree to which a system exhibits service-oriented properties.

In terms of conformance to measurement theory, as required to derive theoretically valid metrics [204], the model (Chapter 3) formalised our understanding of the problem domain (service-oriented design structure and associated coupling relationships), thus providing strong theoretical basis for the derivation of coupling metrics. That is, measuring all possible relationships and system types (related to service-autonomy) defined by the model, allows maintaining the mapping between the intuitive and formal relational models as prescribed by the rules of measurement theory (refer to Appendix B).

The limitation of the formal model of SO design is that it can provide support for the identification of metrics designed for assessment purposes only (as was described in Section 2.5.1, there are two common applications of metrics, assessment and prediction [72]), but it does not provide any support for the definition of predictive metrics. This is because different coupling relationships will have varying impact on the specific quality characteristic under study. To this end, there is a need to establish assumptions that can further constrain and enhance the definition of metrics based on the specific measurement objective [34]. For example, the coupling assumptions defined in Section 4.2 establish explicit links between the different aspects of the property being measured (coupling) and the quality characteristic to be predicted (maintainability). The assumptions will be formally redefined and used as experimental hypotheses in the empirical study presented in Chapter 6.

Additionally, the assumptions can provide support for the derivation of metric weights that reflect the relative strength of a particular coupling relationship in terms of its expected impact on the analysability, changeability, and stability of SO software. To illustrate the purpose of assumptions and weights when using predictive metrics, we can consider the simple example of measuring the height of a human discussed in Chapter 3 and Appendix B, where it was shown that measurement theory can assist in establishing that a proposed measure of height, such as a centimetre, is theoretically valid. Let us assume that there is a need to pre-
dict the ability of a given human to become a professional high jumper using height as one of the predictive factors. In which case, it would be advisable to treat differently the measures of height (or length) of the particular parts of the human body that intuitively have greater influence on jumping ability. For example, suppose that the length of human legs can be considered as the most important indicator of a person’s jumping ability\(^\text{18}\). To this end, establishing relative weights for different parts of the human body will allow deriving a useful measure of human height that can be directly used to predict jumping ability.

Note that some researchers in the software metrics area suggest that metric weights should be used with care since such weights can only be established objectively based on a large number of comprehensive empirical studies [27]. Nonetheless, in this research it was reasoned that without such weights it would be difficult to derive practical and accurate predictive metrics because the coupling relationships covered in Chapter 3 can have varying impact on different quality characteristics. For example, when predicting the maintainability of SO software, the relationships that undermine the rules of service autonomy and reusability should be avoided [67] (as described in Section 4.2). Whereas, when predicting the performance quality characteristic, the service interface-related relationships will have greater significance due to the extra processing introduced by XML marshalling when communicating via WSDL-based service interfaces [4].

### 4.2 Coupling Assumptions and Metric Weights

The coupling assumptions presented in this section assist in the metrics derivation and empirical evaluation processes by proposing connections between the structural property of coupling and the sub-characteristics of maintainability of service-oriented software, namely analysability, changeability, and stability (defined in Section 2.3). The assumptions are based on the review of the related literature in the areas of SOC and software maintainability (Section 2.2 and Section 2.3), covering the key concepts of service-oriented design as described and formalised in Chapter 3.

Note that the decision was made to also identify, where possible, the effect of coupling on the reusability of services so to provide a basis for future work on investigating the concept of service reusability. Reusability was chosen since it can be considered as one of the key design targets of SOC (refer to Section 2.2), and also because it was suggested previously that the reusability of individual software modules is highly related to the overall maintainability of software systems [151, p. 25].

\(^{18}\) This is a hypothetical example which is not necessary supported by any scientific observation.
The following assumptions are divided into two distinct types: service-oriented coupling assumptions (CSA1-CSA5 below) that relate system coupling to the maintainability and reusability of software based on the fundamental characteristics of service-orientation (service-oriented specific coupling relationships and service-autonomy coupling described in Section 4.1), and common coupling assumptions (CCA1 and CCA2) that relate the general notion of coupling (common coupling relationships) to the sub-characteristics of maintainability. Note that the service-oriented specific relationships defined by the formal model of SO design are shown in Figure 4-1. Such relationships can be categorised into:

i) intra-service relationships;

ii) indirect extra-service relationships (via service interfaces);

iii) direct extra-service relationships (between implementation elements belonging to different services).

**Service-Oriented Assumption CSA1 – Intra-service coupling.**

High intra-service coupling between design elements belonging to the same service (as captured by the ISR and IIR relationships defined in Chapter 3 and shown in Figure 4-1) should be avoided since services are intended to be independent components and thus can be maintained in isolation from the system. This type of coupling can be considered as the generic type of design coupling and can be linked to the notion of coupling in Procedural and OO designs. This is because an individual service can be considered as a Procedural or OO subsystem when investigated in isolation from the other services in the system; therefore, the impact of high-intra service coupling on maintainability is expected to be similar to that suggested for the Procedural/OO systems [28, 53, 85]. More specifically, high intra-service coupling will result in decreased analysability, changeability, and stability of a service.

**Service-Oriented Assumption CSA2 – Indirect extra-service coupling.**

Indirect extra-service coupling covers the relationships between services in the system through service interfaces only (as captured by the SIR and SOR relationships shown in Figure 4-1). This type of coupling supports the notion of service-autonomy, and as such, can be considered as the desirable form of (loose) coupling. That is, services in Service-Oriented Computing should communicate with one another via interfaces in order to achieve some desired functionality; therefore, this type of coupling is unavoidable in practice [4, 13, 184]. Nonetheless, indirect extra-service coupling should be weighted higher than intra-service coupling type (as per CSA1) in order to allow for more accurate prediction of maintainability.

---

19 This figure has been already shown in Section 3.3.2, but was also included here to improve the readability of this section and to assist in the description of the corresponding assumptions.
This is because the system functionality is encapsulated in different services, which can be situated across various logical and physical boundaries, making it harder to maintain the system. This is similar to the Procedural and OO systems where coupling between packages/classes is considered to be stronger than the coupling between procedures/methods within the packages/classes themselves [63, 82]. Also, the direction of communication (or locus of impact [33]) will influence the specific sub-characteristics of maintainability as follows:

**Service-Oriented Assumption CSA2.1.** High incoming indirect extra-service coupling (SIR relationships in Figure 4-1) to service interface $s_i$ belonging to service $s$ from the implementation element/s $e_1...e_n$ belonging to service/s $s_1...s_n$, will negatively influence changeability and stability of a system (service/s $s_1...s_n$) since $e_1...e_n$ will be (loosely) dependent on the operations exposed in $s_i$.

**Service-Oriented Assumption CSA2.2.** High outgoing indirect extra-service coupling (SOR relationships in Figure 4-1) from a given service implementation element $e$ of service $s$, to the service interfaces $s_{i1}...s_{in}$ belonging to services $s_1...s_n$ will negatively influence the analysability, changeability, and stability of element $e$ (and thus its encompassing service $s$) since there is a possibility that changes to the operations exposed in service interfaces $s_{i1}...s_{in}$ will influence the functioning of element $e$ (stability), and also more effort will be required to analyse and change this element.
Service-Oriented Assumption CSA3 – Direct extra-service coupling.

Direct extra-service coupling covers the (direct) relationships between implementation elements belonging to different services (as captured by the IR and OR relationships shown in Figure 4-1). This type of coupling can be considered as the strongest (worst) type of extra-service coupling and thus should be avoided. This is because the direct extra-service relationships result in explicit dependencies between implementation of services, thereby decreasing the reusability of such [219]. Additionally, such relationships break the principles of service autonomy [68]. To this end, this type of coupling should be weighted higher than both intra-service and indirect extra-service coupling. Furthermore, the direction of communication will influence some of the maintainability sub-characteristics as follows:

Service-Oriented Assumption CSA3.1. High incoming direct extra-service coupling from service implementation element/s e₁...eₙ belonging to service/s s₁...sₙ, to a given implementation element e of service s, will negatively influence changeability and stability of a system (service/s s₁...sₙ) since e₁...eₙ will be dependent on the internal implementation characteristics of service s.

Service-Oriented Assumption CSA3.2. High outgoing direct extra-service coupling from a given service implementation element e of service s, to the implementation elements e₁...eₙ belonging to the rest of the system will negatively influence the analysability, changeability, and stability of element e (and thus its encompassing service s) since there is a possibility that changes to the external elements e₁...eₙ will influence the functioning of an element e (stability), and also more effort will be required to analyse and change this element.

Note that the following Service-Oriented Assumptions CSA4 and CSA5 cover the service-autonomy related coupling, and as such, they do not introduce any weights given that they do not measure the actual relationships between design artefacts.

Service-Oriented Assumption CSA4 – System Partitioning.

A high number of implementation elements (e₁...eₙ) that do not belong to any of the services in the system (for example, legacy modules which are not used by any of the services) will result in decreased analysability of a system, and also undermine one of the core principles of service-orientation that a system should be constructed as a set of interacting services [12, 130]. This type of coupling reflects the conformance of a given system design to the Partitioned Service-Oriented System (PARSOS) defined in Section 3.3.4.

Service-Oriented Assumption CSA5 – System Purity.

A high number of implementation elements (e₁...eₙ) that belong to more than one service will increase the interdependencies between services [68], thereby decreasing their reusability and analysability. Also, the stability of a system will decrease since changes to elements e₁...eₙ can potentially influence more than one service. Note that it might appear that structuring a
system as a set of fully disjoint services which do not share any implementation elements can potentially introduce some degree of implementation-level duplication [67]. To avoid such duplication, it is advisable to place the implementation elements that cover some common system-level functionality into separate (utility) services [68]. This type of coupling reflects the conformance of a given system design to the Pure Service-Oriented System (PURSOS) defined in Section 3.3.4.

Common Assumption CCA1 – Elements belonging to different development paradigms. The common coupling between implementation elements belonging to different development paradigms (for example, OO class to Procedural package) is ‘stronger’ than coupling between elements of the same type (for example, OO class to OO class) and thus should be weighted higher, irrespective of whether the relationship is intra or extra-service. This is because such communication will require extra development and maintenance efforts due to implementation specific issues, thus negatively influencing the analysability and changeability of a system.

Common Assumption CCA2 – Dynamic collaborations. A high number of design elements interacting in order to achieve some desired functionality in response to the invocation of an operation will result in the decreased analysability of a system since the entire call chain needs to be analysed in order to understand the functioning of the operation. Also, the stability will be affected since an element to which a given operation belongs will be dependent on an increasing number of external elements. This is consistent with the understanding of the relationship between software maintainability and dynamic collaborations in other development paradigms (such as OO [44, 45]).

To support the derivation of metrics for predicting the maintainability of SO software products based on the proposed coupling assumptions (the impact of the assumptions on the maintainability and reusability is summarised in Table 4-1), a series of weights for different types of relationships are provided in Table 4-2. More specifically, the values in the WEIGHT column represent the relative strength of coupling as described by the associated assumptions. For example, SIR/SOR relationships (Assumption CSA2) are weighted higher than IIR/ISR relationships (Assumption CSA1), but lower than IR/OR relationships (Assumption CSA3). It is important to note that the proposed weights are preliminary and not fixed at this stage. That is, the chosen weight values (1, 2, and 3) represent a rough estimation of the particular coupling strength, and were provided in this section for illustrative purposes only. The concrete and objective weights could be established in future work based on comprehensive empirical studies that will investigate a large number of service-oriented systems as described further in Chapter 6.
### CHAPTER 4. SERVICE-ORIENTED COUPLING METRICS

#### Table 4-1. Relationships between coupling assumptions and software maintainability and reusability

<table>
<thead>
<tr>
<th>ASSUMPTIONS</th>
<th>MAINTAINABILITY SUB-CRTERISTICS</th>
<th>REUSABILITY (FUTURE WORK)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ANALYSABILITY</td>
<td>CHANGEABILITY</td>
</tr>
<tr>
<td>CSA1</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>CSA2 and CSA2.1-CSA2.2</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>CSA3 and CSA3.1-CSA3.2</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>CSA4</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>CSA5</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>CCA1</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>CCA2</td>
<td>✔</td>
<td></td>
</tr>
</tbody>
</table>

#### Table 4-2. Weighted Service-Oriented design relationships

<table>
<thead>
<tr>
<th>RELATIONSHIP TYPE (FROM CHAPTER 3)</th>
<th>ASSUMPTION</th>
<th>WEIGHT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service specific relationships (CHAPTER 3. TABLE 3.1: D4.1 - D4.6)</td>
<td>Service (SW)</td>
<td></td>
</tr>
<tr>
<td>IIR, ISR</td>
<td>CSA1</td>
<td>1</td>
</tr>
<tr>
<td>SIR, SOR</td>
<td>CSA2 (CSA2.1 and CSA2.2)</td>
<td>2</td>
</tr>
<tr>
<td>IR, OR</td>
<td>CSA3 (CSA3.1 and CSA3.2)</td>
<td>3</td>
</tr>
<tr>
<td>Common relationships between two design artefacts (CHAPTER 3. TABLE 3.1: D3)</td>
<td>General (GW)</td>
<td></td>
</tr>
<tr>
<td>CC, PP, BPSBPS, CI, IC, II, PH, HP, HH, CSI, SIC, PSI, SIP, BPSSI, SIBPS</td>
<td>CCA1</td>
<td>1</td>
</tr>
<tr>
<td>CP, PC, CH, CBPS, BPSC, PI, IP, IBPS, BPSI, PBPS, PSP</td>
<td>CCA1</td>
<td>2</td>
</tr>
</tbody>
</table>
CHAPTER 4. SERVICE-ORIENTED COUPLING METRICS

4.3 Metrics Definitions

The metrics presented in this section can be used to measure the static and dynamic coupling in service-oriented designs. Static metrics can be collected by examining documents related to the structure of a SO system (Figure 4-2 provides an example of such a structure). The information needed to measure the dynamic (run-time) coupling can be obtained from behavioural design documents such as: sequence or collaboration diagrams; flow charts or data flow diagrams; or by tracing the source code or executable binaries where available.

Note that in order to define useful and practical metrics, the decision was made to use elements (and not operations or attributes) as the basic measurement construct. Therefore, multiple couples to the same design element in the same direction are counted as one couple only. That is, the frequency of relationships between elements is not counted, as is the case with widely-used OO metrics such as CBO [44] that measure coupling between classes rather than methods. As an example consider: i) element A uses operation x and references attribute y of element B; ii) element A uses operations s and t of element C; and iii) element B uses operations l, m and n of element A. In this case, the total relationship count for element A will be 3 (1 for the connection to element B + 1 for the connection from element B + 1 for the connection to element C).

4.3.1 Metric Naming and Organisation

The proposed metrics are applicable on either a per system, service, service interface, or service implementation element basis. The metrics are presented individually in terms of:

i) Informal description of the metric that covers the motivation behind its derivation based on the coupling assumptions described in the previous section;

ii) A formal definition of the metric using the formalism captured by the model of service-oriented software designs presented in Chapter 3;

iii) Measurement procedure that summarises the process of metric collection (again based on the sets of relationships captured by a formal model), and uses the example design shown in Figure 4-2 to demonstrate the collected values for each of the proposed metrics. That is, the design shown in Figure 4-2 shows an example service-oriented system consisting of three services, which include different relationship types that will be quantified by the proposed metrics.

iv) Theoretical validation, which is based on the property-based software engineering measurement framework [30] described in Section 2.5.2, where a given metric can be deemed valid if it conforms to the prescribed mathematical characteristics of coupling (shown in Section 2.5.2, Table 2-4).
Note that in the process of the theoretical validation it was discovered that most of the derived metrics failed to satisfy one of the coupling characteristics due to the different granularity of measurement performed in this research compared to that prescribed by Briand et al. [30, 32] as explained further in Section 4.4.

The order of the metrics presentation is based on the order of associated coupling assumptions defined in Section 4.2. Additionally, to improve the readability of this section, the metrics are presented in two parts, with Section 4.3.2 describing the primary metrics that measure all possible relationships and system types from the formal model of SO design, and Section 4.3.3 covering metrics that represent aggregations of the primary metrics.

Finally, the metrics were named based on the following rules:

i) The relationship related metrics were named based on three different aspects: 1) the locality of the coupling relationship (intra-service or direct/indirect extra-service); 2) the direction of the relationship (incoming or outgoing), where applicable; and 3) the type/level of abstraction of the artefact to be measured (which can be system, service, service interface, or service implementation element).

ii) The service-autonomy related metrics were named based on the type of SO system design under study. For example, metric COUP-M10:SPURF is named System Purity Factor (SPURF) to indicate the conformance of a given system design to the Pure Service-Oriented System (PURSOS) defined in Section 3.3.4.
4.3.2 Primary Metrics

<table>
<thead>
<tr>
<th>METRIC NAME</th>
<th>WEIGHTED INTRA-SERVICE COUPLING BETWEEN ELEMENTS (WISCE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVERVIEW</td>
<td>WISCE for a given service implementation element ( e ) belonging to a service ( s ) is the <em>weighted</em> count of the number of other implementation elements of the same service ( s ) to which element ( e ) is coupled via incoming or outgoing relationships as part of the <em>intra-service coupling</em> covered by Assumption CSA1.</td>
</tr>
<tr>
<td>DESCRIPTION / MOTIVATION</td>
<td>Covers Assumptions CSA1 and CCA1.</td>
</tr>
<tr>
<td></td>
<td>Services are intended to be independent components and thus can be maintained in isolation from the system. Therefore, it is useful to measure the coupling within a single service as described by Assumption CSA1. More specifically, high intra-service coupling of a given service implementation element can indicate bad internal design structure of a service, which is expected to have a negative effect on its maintainability. For example, implementation element ( c7 ) belonging to service ‘Billing’ shown in Figure 4-2 has high coupling compared to the other implementation elements of ‘Billing’, thereby indicating a potential design problem that should be fixed prior to commencing the implementation of this service. This is consistent with the notion of general coupling in the Procedural and OO paradigms [28, 53, 85]. Therefore, WISCE can be considered as a weighted version of the widely-used Coupling Between Objects (CBO) metric from the CK suite of OO metrics [44], with weights being assigned based on the type of implementation elements involved in the coupling relationship according to Assumption CCA1.</td>
</tr>
<tr>
<td>FORMAL DEFINITION</td>
<td>WISCE (( e )) = (</td>
</tr>
<tr>
<td>MEASUREMENT PROCEDURE</td>
<td>Count the weighted number of occurrences of a particular element in the ISR(s) for a given service ( s ), where ISR(s) is the set of all internal service relationships of service ( s ) (refer to Table 3.1 - D4.2 and Figure 4-1). The weights are assigned according to Table 4-2 based on the types of elements involved in the communication (general weight (GW)). For example, WISCE (( c1 )) = (</td>
</tr>
</tbody>
</table>
| VALIDATION | Property COUPLING.1 (Non-negativity) is satisfied since WISCE for a given service implementation element can only equal zero or some positive value indicating the number of couples to/from the other elements belonging to the same service. It will never be negative under any cir-

(February, 2009)
Property COUPLING.2 (Null Value) is satisfied since WISCE for a given service implementation element will be null (or zero) in the case where this element does not have any intra-service relationships.

Property COUPLING.3 (Monotonicity) is satisfied since the coupling of a service implementation element cannot be decreased by adding more relationships between this element and the rest of the elements of a service.

Property COUPLING.4 (Merging of Modules) is satisfied since merging two service implementation elements \( e_m \) and \( e_n \) together will result in a decreased amount of coupling for the resultant element \( e_{mn} \) (or in the worst case the coupling of \( e_{mn} \) will equal to the sum of original coupling for \( e_m \) and \( e_n \)), but it will not increase.

Property COUPLING.5 (Disjoint Module Additivity) is not satisfied since the frequency of coupling relationships (multiple connections between two implementation elements) is not counted. For example, let us assume that an operation of element \( e_m \) belonging to service \( s \) is invoked by element \( e'' \) belonging to the same service \( s \) (one incoming couple), and element \( e_n \) (disjoint from \( e_m \)) belonging to the same service \( s \) also has an incoming relationship from element \( e'' \), then the combined element \( e_{mn} \) will be coupled to \( e'' \) through one (incoming) relationship only, which is not the sum of previous couples. As such, the coupling of disjoint elements will not be additive. Note that this metric (and all the other metrics that fail this property) is still believed to be valid for the reasons described in the validation summary presented in Section 4.4.

Note: Properties COUPLING.1 and COUPLING.2 are satisfied ‘by default’ given that all the metrics presented in this chapter are defined on ratio or absolute scales. These scale types imply that the obtained values cannot be negative, with zero representing the absence of quality being measured. As such, only the validation of the metrics in regards to the Properties COUPLING.3 - COUPLING.5 will be provided for the remaining metrics.

Table 4-3. COUP-M1: Weighted Intra-Service Coupling between Elements (WISCE)

<table>
<thead>
<tr>
<th>METRIC NAME</th>
<th>SERVICE INTERFACE TO INTRA ELEMENT COUPLING (SIIEC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVERVIEW</td>
<td>SIIEC for a given service ( s ) is a count of the relationships between its service interface ( si_s ) and the implementation elements belonging to service ( s ) that directly implement operations exposed in the interface ( si_s ) as part of the intra-service coupling covered by Assumption CSA1.</td>
</tr>
</tbody>
</table>
Covers Assumption CSA1.

This metric can be considered as a variation of the WISCE metric presented above. The decision was made to define a separate metric for quantifying the number of internal relationships between a service interface and service implementation elements that directly implement the operations exposed in this interface since such relationships should be kept to minimum in order to avoid unnecessary dependencies on the service implementation elements. That is, a large number of service implementation elements invoked from a service interface, can result in the decreased analyzability of this service due to an unnecessarily ‘tight’ linkage between its interface and implementation [68].

For example, a service interface $si_3$ belonging to service ‘Billing’ shown in Figure 4-2 has a high SIIEC value (SIIEC = 3), where three service operations are mapped to three implementation elements (i.e. there is a one-to-one mapping between service interface operations and implementation elements, which can be considered as the strongest Service Interface to Intra Element Coupling possible given that that each service interface operation can be implemented by at most one implementation element in the current implementation technologies). This suggests that three separate chain of calls originated from the interface $si_3$ will need to be analysed when performing maintenance tasks. As such, it is advisable to keep the SIIEC value as close to one as possible. This can be achieved by having a dedicated implementation element to which all service operations are initially mapped.

| FORMAL DEFINITION | $\text{SIIEC} (s) = |\text{IIR} (s)|$ |
|-------------------|----------------------------------|
| Scale: Ratio; Measurement Unit: Count |

| MEASUREMENT PROCEDURE | Count the number of elements in the IIR(s) for a given service $s$, where IIR(s) is the set of interface to implementation relationships (refer to Table 3.1 - D4.1 and Figure 4-1). Note that the possible types of service interface to implementation element relationships (SIC, SIP, SIBPS relationships from Section 3.3.1) are not weighted since specific types of elements implementing the service interface are not expected to have a different impact on the overall maintainability of a service because service interfaces should be technology agnostic. For example, $\text{SIIEC} (\text{Billing}) = |\{<si_3, c_4>, <si_3, c_5>, <si_3, c_6>\}| = 3$ in the design shown in Figure 4-2. |

| VALIDATION | Property COUPLING.3 (Monotonicity) is satisfied since coupling of a service interface cannot decrease when adding more relationships between this interface and the rest of the elements in the service. |

---

20 Such ‘implementation’ is typically done via middleware support in the case of WSDL-based interfaces
Property COUPLING.4 (Merging of Modules) is satisfied since merging two services together (merging of service interfaces implies merging of services since there is one to one relationship between services and their interfaces) can only decrease the amount of coupling (or in the worst case the coupling will remain the same), but it will not increase.

Property COUPLING.5 (Disjoint Module Additivity) is satisfied since when merging two unrelated services all the internal relationships between their interfaces and implementation elements will be preserved. As such, the coupling of disjoint services interfaces (services) will be additive.

Table 4-4. COUP-M2: Service Interface to Intra Element Coupling (SIEC)

<table>
<thead>
<tr>
<th>METRIC NAME</th>
<th>EXTRA-SERVICE INCOMING COUPLING OF SERVICE INTERFACE (ESICSI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVERVIEW</td>
<td>ESICSI for a given service ( s ) is a count of the number of system elements not belonging to service ( s ) that couple to this service through its interface ( s_1 ) as part of the indirect extra-service coupling covered by Assumption CSA2.</td>
</tr>
<tr>
<td>DESCRIPTION / MOTIVATION</td>
<td>Covers Assumption CSA2 and CSA2.1. Although indirect extra-service coupling can be considered as the desirable form of loose coupling as described in Section 4.2, it is advisable to avoid excessive coupling between services in the system. This is because system functionality, encapsulated in different services, is situated across various logical and physical boundaries, making it harder to maintain the system. More specifically, as the value of ESICSI increases so does the dependency of the rest of the system on this service, thereby resulting in decrease of system changeability and stability. For example, a service interface ( s_1 ) belonging to service ‘Timetabling’ shown in Figure 4-2 has a high ESICSI value compared to the other services in the system. That is, any changes to the operations exposed in service interface ( s_1 ) can potentially require changes to the ‘Enrolling’ and ‘Billing’ services. As such, this metric provides an indication of how critical the ‘Timetabling’ service is within a system wide context. Note that the high value of ESICSI might indicate that a service is too coarse-grained (as reflected by the large number of operations exposed in its service interface). To reduce ESICSI, a service could be separated into a number of finer-grained services, or some of the operations exposed in the service interface can be moved to the other services in the system [68].</td>
</tr>
<tr>
<td>FORMAL DEFINITION</td>
<td>( \text{ESICSI} (s) =</td>
</tr>
</tbody>
</table>
MEASUREMENT PROCEDURE

Count the number of occurrences of a service interface in the SIR(s) for a given service \( s \), where SIR(s) represents a set of service incoming relationships (refer to Table 3.1 - D4.5 and Figure 4-1). Note that the service interface related relationships are weighted according to the service weights (SW) only. They are not weighted according to the type of implementation element connecting to the service interface (general weights (GW)) since element types are not expected to impact upon maintainability because services should be technology agnostic.

For example, ESICSI (Timetabling) = \(|\{<\text{bps, si1}>, <\text{c4, si1}>\}|*2 = 4\) in the design shown in Figure 4-2.

VALIDATION

Property COUPLING.3 (Monotonicity) is satisfied since the coupling of a given service interface cannot be decreased by adding more incoming relationships to this service interface.

Property COUPLING.4 (Merging of Modules) is satisfied since merging two service interfaces \( \text{ser}_m \) and \( \text{ser}_n \) together should result in a decreased amount of outgoing extra-service coupling for the resultant interface \( \text{ser}_{mn} \) (or in the worst case the coupling of \( \text{ser}_{mn} \) will equal to the sum of original coupling for \( \text{ser}_m \) and \( \text{ser}_n \)), but it will not increase.

Property COUPLING.5 (Disjoint Module Additivity) is not satisfied since multiple connections to a given service interface from the same outside element are not counted. Refer to the explanation provided for metric COUP-M1.

Table 4-5. COUP-M3: Extra-Service Incoming Coupling of Service Interface (ESICSI)

<table>
<thead>
<tr>
<th>METRIC NAME</th>
<th>ELEMENT TO EXTRA SERVICE INTERFACE OUTGOING COUPLING (EESIOC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVERVIEW</td>
<td>EESIOC for a given service implementation element ( e ) is a count of the number of other service interfaces to that are used (coupled to) by implementation element ( e ) as part of the indirect extra-service coupling covered by Assumption CSA2.</td>
</tr>
<tr>
<td>DESCRIPTION / MOTIVATION</td>
<td>Covers Assumption CSA2 and CSA2.2. As described previously (refer to Assumption CSA2), the indirect extra-service coupling can be considered as a loose type of coupling. Nonetheless, it is advisable to avoid excessive and unnecessary indirect coupling between services in the system since services can be situated across various logical and physical boundaries. As the value of EESIOC for a given implementation element ( e ) increases so does the dependency of this element on the other services in the system. As such, the stability of element ( e ) (and therefore its encompassing service) will be decreased since there is a possibility that changes to the coupled service interfaces will influence the functioning of element ( e ). For example, the implemen-</td>
</tr>
</tbody>
</table>
CHAPTER 4. SERVICE-ORIENTED COUPLING METRICS

tation element \( bp \) belonging to the ‘Enrolling’ service (shown in Figure 4-2) might need to be modified when changes to the interfaces \( si_2 \) and \( si_3 \) are made. As such, this metric provides an indication of the degree of dependency of a given implementation element on the other services in the system. Moreover, the *analysability* and *changeability* of this element can also be effected given that greater effort will be needed to analyse (and change) the element which uses functionality provided by the outside artefacts.

Note that in practice, it is expected that business process scripts (BPS) will have relatively high values of EESIOC due to the orchestration-based nature of the business processes (refer to Section 2.2). This should be taken into consideration when analysing the obtained values of EESIOC. Nevertheless a high value of EESIOC can potentially indicate that some of the services accessed (invoked) by a given implementation element in order to achieve some business functionality are too fine-grained. To this end, the software engineer might consider merging some of the invoked services in order to reduce EESIOC.

### FORMAL DEFINITION

\[
\text{EESIOC} (e) = |\{<e, si> | e \in (C_s \cup I_s \cup P_s \cup H_s \cup \text{BPS}_s) \land si \in \text{SI} \land <e, si> \in \text{SOR(s)}\}| * \text{WeightFactor} (SW)
\]

*Scale*: Ratio; *Measurement Unit*: Count

### MEASUREMENT PROCEDURE

Count the number of occurrences of relationships involved implementation element to be measured in the SOR(s) for a given service \( s \), where SOR(s) represents a set of *service outgoing* relationships (refer to Table 3.1 - D4.6 and Figure 4-1). The relationships are weighted according to the service weights (SW) only (similar to metric COUP-M3).

For example, \( \text{EESIOC} (bp) = |\{<bp, si1>, <bp, si3>\}|*2 = 4 \) in the design shown in Figure 4-2.

### VALIDATION

Property COUPLING.3 (Monotonicity) is satisfied since the *coupling* of a given service implementation element *cannot be decreased* by adding more outgoing relationships to other service interfaces.

Property COUPLING.4 (Merging of Modules) is satisfied since merging two service implementation elements \( e_m \) and \( e_n \) together should result in a decreased amount of outgoing extra service coupling for the resultant element \( e_{mn} \) (or in the worst case the coupling of \( e_{mn} \) will equal to the sum of original coupling for \( e_m \) and \( e_n \)), but it *will not increase*.

Property COUPLING.5 (Disjoint Module Additivity) is *not* satisfied since multiple connections to a given service interface from the same element are not counted. Refer to the explanation provided for metric COUP-M1.

<table>
<thead>
<tr>
<th>Table 4-6. COUP-M4: Element to Extra Service Interface Outgoing Coupling (EESIOC)</th>
<th></th>
</tr>
</thead>
</table>

(February, 2009)
## METRIC NAME

**WEIGHTED EXTRA-SERVICE INCOMING COUPLING OF AN ELEMENT (WESICE)**

### OVERVIEW

WESICE for a given service implementation element $e_1$ of service $s$ is the *weighted* count of the number of system elements not belonging to service $s$ that couple to an element $e_1$ as part of the *direct extra-service coupling* covered by Assumption CSA3.

### DESCRIPTION / MOTIVATION

Covers Assumptions CSA3 and CSA3.1.  

*Direct extra-service coupling* introduces *tight* (implementation-dependent) coupling between services and should be avoided as much as possible. Firstly, it is generally advisable to avoid excessive incoming extra-service coupling between elements belonging to different services in the system since the system functionality is encapsulated in different services that can be situated across various logical and physical boundaries (as described in COUP-M3:ESICSI metric). Secondly, the direct extra-service relationships result in explicit dependencies between implementation of services, thereby decreasing their *reusability* [219] and also breaking the principles of *service autonomy* [68].

Moreover, the incoming coupling from service implementation elements belonging to different services to element $e_1$ is expected to negatively influence *changeability* and *stability* of these services since as WESICE increases, so does the number of external elements (and services) dependent upon the implementation characteristics of $e_1$. As such, the reuse of the services containing external elements will be limited. For example, the business process script $bp$ belonging to the ‘Enrolling’ service shown in Figure 4-2 is directly coupled to the implementation element (OO class) $c2$ belonging to the ‘Timetabling’ service, therefore any changes to the operations of element $c2$ will have to be propagated to the element $bp$.

Note that to reduce WESICE value for element $e_1$ belonging to service $s$, the operations of element $e_1$ accessed by elements $e_n...e_m$ belonging to the rest of the system could be exposed via a service interface $s_i$.

### FORMAL DEFINITION

$$\text{WESICE} (e_1) = |\{<e, e_1>\ast \text{WeightFactors}(SW \text{ and GW}) \mid e \in (C - C_s \cup I - I_s \cup P - P_s \cup H - H_s \cup \text{BPS} - \text{BPS}_s) \land e_1 \in (C_s \cup I_s \cup P_s \cup H_s \cup \text{BPS}_s) \land <e, e_1> \in \text{IR}(s)\}|$$

Scale: Ratio; Measurement Unit: Count

### MEASUREMENT PROCEDURE

Count the *weighted* number of occurrences of a particular element in the IR(s) for a given service $s$, where IR(s) is the set of all *incoming relationships* of the implementation element belonging to service $s$ from the implementation elements belonging to the rest of the system (Table 3.1 - D4.3 and Figure 4-1). The weights are assigned according to Table 4-2.
in two consecutive steps. Firstly, the general weights (GW) are assigned based on the types of elements involved in the communication. Secondly, the service weights (SW) are assigned based on the locality of communication.

For example, WESICE (ph) = |{<c2, bp> * 2}| * 3 = 6 in the design shown in Figure 4-2.

**VALIDATION**

Property COUPLING.3 (Monotonicity) is satisfied since the coupling of a given service implementation element cannot be decreased by adding more extra-service incoming relationships from the outside elements.

Property COUPLING.4 (Merging of Modules) is satisfied since merging two service implementation elements \( e_m \) and \( e_n \) together should result in a decreased amount of incoming extra-service coupling for the resultant element \( e_{mn} \) (or in the worst case the coupling of \( e_{mn} \) will equal to the sum of original coupling for \( e_m \) and \( e_n \)), but it will not increase.

Property COUPLING.5 (Disjoint Module Additivity) is not satisfied since multiple connections from the same element are not counted as described previously. Refer to the explanation provided for metric COUP-M1.

**Table 4-7. COUP-M5: Weighted Extra-Service Incoming Coupling of an Element (WESICE)**

<table>
<thead>
<tr>
<th>METRIC NAME</th>
<th>WEIGHTED EXTRA-SERVICE OUTGOING COUPLING OF AN ELEMENT (WESOCE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVERVIEW</td>
<td>WESOCE for a given service implementation element ( e ) of a particular service ( s ) is the weighted count of the number of system elements not belonging to the same service that are used (coupled to) by this element as part of the direct extra-service coupling covered by Assumption CSA3.</td>
</tr>
<tr>
<td>DESCRIPTION / MOTIVATION</td>
<td>Covers Assumptions CSA3 and CSA3.2. Direct extra-service coupling introduces tight coupling between services and should be avoided as much as possible. Firstly, it is generally advisable to avoid excessive outgoing extra-service coupling between elements belonging to different services in the system (as described in metrics COUP-M4). Secondly, the outgoing coupling from a given service implementation element ( e ) (belonging to service ( s )) to service implementation elements belonging to different services will negatively influence stability of ( e ) since as WESOCE increases so does the possibility that external changes will influence element ( e ) and thus service ( s ). For example, the implementation element (OO class) ( c2 ) shown in Figure 4-2 is coupled to the implementation element ( c3 ) via direct extra-service relationship; therefore, the stability of element ( c2 ) will be decreased since there is a possibility that changes to element ( c3 ) will influence the functioning of implementation element ( c2 ) and its encompassing service (‘Time- tabling’). Moreover, the analysability and changeability of this element can also be effected given that greater effort will be needed to analyse</td>
</tr>
</tbody>
</table>
(and change) element e2 because it uses functionality provided by the outside services.

Note that to reduce WESOCE, the operations of elements em...en accessed from element e1 could be exposed via service interfaces of services encompassing elements em...en and accessed through interface instead.

**FORMAL DEFINITION**

\[
\text{WESOCE}(e) = \{<e, e_1>\star \text{WeightFactors(SW and GW)} \mid e \in (C_s \cup I_s \cup P \cup H \cup BPS) \wedge e_1 \in (C - C_s \cup I - I_s \cup P - P_s \cup H - H_s \cup BPS - BPS_s) \wedge <e, e_1> \in \text{OR}(s)\}
\]

Scale: Ratio; Measurement Unit: Count

**MEASUREMENT PROCEDURE**

Count the *weighted* number of occurrences of a particular element in the OR(s) for a given service s, where OR(s) is the set of all outgoing relationships from the implementation element belonging to a service s to the implementation elements that belong to the rest of the system (Table 3.1 - D4.4 and Figure 4-1). The weights are assigned according to Table 4-2 in two consecutive steps. Firstly, the general weights (GW) are assigned based on the types of elements involved in the communication. Secondly, the service weights (SW) are assigned based on the locality of communication.

For example, WESOCE (c2) = |{<c2, c3>|}*3= 3 in the design shown in Figure 4-2.

**VALIDATION**

Property COUPLING.3 (Monotonicity) is satisfied since the coupling of a given service implementation element cannot be decreased by adding more extra-service outgoing relationships to the outside elements.

Property COUPLING.4 (Merging of Modules) is satisfied since merging two service implementation elements em and en together should result in a decreased amount of outgoing extra-service coupling for the resultant element emn (or in the worst case the coupling of emn will equal to the sum of original coupling for em and en), but it will not increase.

Property COUPLING.5 (Disjoint Module Additivity) is not satisfied since multiple connections to the same element are not counted. Refer to the explanation provided for metric COUP-M1.

<table>
<thead>
<tr>
<th>METRIC NAME</th>
<th>NUMBER OF COUPLED INCOMING SERVICES FOR A SERVICE (NCIS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVERVIEW</td>
<td>NCIS for a given service s is a distinct count of other services in the system having elements connecting to service s through either its implementation elements (direct extra-service coupling) or its service interface</td>
</tr>
</tbody>
</table>
CHAPTER 4. SERVICE-ORIENTED COUPLING METRICS

| DESCRIPTION / MOTIVATION | NCIS is based on the Assumptions CSA2.1 and CSA3.1. That is, as the value of NCIS for a given service \( s \) increases so does the dependency of the other services in the system on this service, which can decrease the changeability and stability of a system. This metric can be considered as the variation of the COUP-M3:ESICSI and COUP-M5:WESICE metrics defined previously, which are also used to measure the incoming extra-service coupling. However, there are two main differences between NCIS and the ESICSI/WESICE metrics: i) NCIS measures coupling at the service level without considering the individual couples between service interfaces or implementation elements as was the case with ESICSI and WESICE metrics; and ii) NCIS does not differentiate between the direct and indirect extra-service relationships. Such differences can be considered as the limitation of this metric since they can potentially result in the loss of measurement accuracy. For example, the values of NCIS for a given service \( s \) will be less precise and informative than the combined measures of ESICSI and WESICE for service \( s \) (such combined measure is defined in Section 4.3.3). Nonetheless, the decision was made to include this metric because NCIS can be calculated at the very early phases of the SDLC. More specifically, NCIS can be used to quantify the coupling between ‘black box’ services during the Analysis phase of SDLC after all major services in the system have been identified but not yet designed in terms of the concrete implementation elements.

| FORMAL DEFINITION | NCIS \((s)\) = \(|\text{CIS}(s)|\), where CIS represents the set of all services coupled to service \( s \) via direct or indirect relationships. Scale: Ratio; Measurement Unit: Count

| MEASUREMENT PROCEDURE | Count the number of services in the system, elements of which connect to the service \( s \) through either direct incoming extra-service relationships \( \text{IR}(s) \) or indirect incoming extra-service relationships \( \text{SIR}(s) \) (Table 3.1 - D4.3 and D4.5, and Figure 4-1). For example, NCIS (Timetabling) = 2 in the design shown in Figure 4-2.

| VALIDATION | Property COUPLING.3 (Monotonicity) is satisfied since the coupling of a service cannot be decreased by adding more incoming relationships between these services and the rest of the services in the system. Property COUPLING.4 (Merging of Modules) is satisfied since merging two services together will result in a decreased amount of coupling for each of these services (or in the worst case the coupling will remain the same), but it will not increase. Property COUPLING.5 (Disjoint Module Additivity) is not satisfied since
multiple connections from the same service are not counted as discussed previously.

Table 4-9. COUP-M7: Number of Coupled Incoming Services (NCIS)

<table>
<thead>
<tr>
<th>METRIC NAME</th>
<th>NUMBER OF COUPLED OUTGOING SERVICES FOR A SERVICE (NCOS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVERVIEW</td>
<td>NCOS for a given service ( s ) is a distinct count of other services in the system, to which service ( s ) is coupled either through an element (direct extra-service coupling) or a service interface (indirect extra-service coupling).</td>
</tr>
<tr>
<td>DESCRIPTION / MOTIVATION</td>
<td>This metric is based on the Assumptions CSA2.2 and CSA3.2. That is, as the value of NCOS for a given service ( s ) increases so does its dependency on the other parts of a system, which can negatively influence the analysability, changeability, and stability of service ( s ). This metric can be considered as the variation of the COUP-M4:EESIOC and COUP-M6:WESOCE metrics defined previously, which are also designed to measure the outgoing extra-service coupling. However, as was the case with the COUP-M7:NCIS metric defined above, there are two main differences between NCOS and EESIOC/ WESOCE metrics: i) NCOS measures coupling at the service level without considering the individual couples between service interfaces or implementation elements; and ii) NCOS does not differentiate between the direct and indirect extra-service relationships. Such differences can be considered as the limitation of this metric due to the potential loss of measurement accuracy. Nevertheless, NCOS was included in the overall suite of coupling metrics since it can be used to quantify extra-service coupling during the earlier phases of the SDLC after all major services in the system have been identified but not yet designed in terms of the concrete implementation elements.</td>
</tr>
<tr>
<td>FORMAL DEFINITION</td>
<td>NCOS ( (s) =</td>
</tr>
<tr>
<td>Scale: Ratio; Measurement Unit: Count</td>
<td></td>
</tr>
<tr>
<td>MEASUREMENT PROCEDURE</td>
<td>Count the number of services in the system to which this service connects through either direct outgoing extra-service relationships ( \text{OR(s)} ), or indirect outgoing extra-service relationships ( \text{SOR(s)} ) (Table 3.1 - D4.4 and D4.6, and Figure 4-1) For example, NCOS (Timetabling) = 1 in the design shown in Figure 4-2.</td>
</tr>
<tr>
<td>VALIDATION</td>
<td>Property COUPLING.3 (Monotonicity) is satisfied since the coupling of a service cannot be decreased by adding more outgoing relationships to</td>
</tr>
</tbody>
</table>

(February, 2009)
CHAPTER 4. SERVICE-ORIENTED COUPLING METRICS

Property COUPLING.4 (Merging of Modules) is satisfied since merging two services together will result in a decreased amount of coupling for each of these services (or in the worst case the coupling will remain the same), but it will not increase.

Property COUPLING.5 (Disjoint Module Additivity) is not satisfied since multiple connections to the same service are not counted as discussed previously.

Table 4-10. COUP-M8: Number of Coupled Outgoing Services (NCOS)

<table>
<thead>
<tr>
<th>METRIC NAME</th>
<th>SYSTEM PARTITIONING FACTOR (SPARF)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVERVIEW</td>
<td>SPARF for a given Service-Oriented system SOS measures the degree of partitioning of this system into services. More specifically, SPARF is the ratio of total elements in the system belonging to at least one service to the total number of the elements in the system.</td>
</tr>
<tr>
<td>DESCRIPTION / MOTIVATION</td>
<td>Covers Assumption CSA4.</td>
</tr>
<tr>
<td></td>
<td>A high number of elements that do not belong to any of the system services will result in decreased analysability of a system due to a lack of system modularisation [36], and will also undermine one of the core principles of service-orientation that a system should be constructed as a set of interacting services [13, 181].</td>
</tr>
<tr>
<td></td>
<td>This type of coupling is related to the notion of service-autonomy described in Section 4.1 since it reflects the conformance of a given system design to the Partitioned Service-Oriented System (PARSOS) defined in Section 3.3.4.</td>
</tr>
<tr>
<td>FORMAL DEFINITION</td>
<td>SPARF (SOS) =</td>
</tr>
<tr>
<td></td>
<td>Scale: Absolute; Measurement Unit: Count</td>
</tr>
<tr>
<td>MEASUREMENT PROCEDURE</td>
<td>Count the total number of design artefacts (service interfaces and service implementation elements) that belong to at least one service (ser) from the set of all the services SER (Table 3.1 – D6 and D6.1), and then divide this number by the total number of design artefacts in the system.</td>
</tr>
<tr>
<td></td>
<td>Values of SPARF will range from zero to one, where a value of one indicates that all the elements in the system belong to at least one service, and as such, the system can be considered as a PARSOS (Table 3.1 – D7). Conversely, a value of zero indicates the total absence of services in the rest of the services in the system.</td>
</tr>
</tbody>
</table>

106

(February, 2009)
the system. For example, SPARF (AMS) = 19/21 = 0.9 in the design shown in Figure 4-2.

**VALIDATION**

The coupling properties of the *property based software measurement* framework are not directly applicable to this metric (and the following metric COUP-M6: SPURF) since SPARF does not measure any of the communicational relationships from the formal model of Chapter 3. Nonetheless, given that this metric is based on the count of system elements according to the above-described rules, the decision was made to validate SPARF using the *number of system elements* as a validation construct instead of the actual *coupling relationships* as follows:

Property COUPLING.3 (Monotonicity) is satisfied since the *coupling* (as measured by SPARF) of a service-oriented system cannot be decreased by adding more implementation elements to this system.

Property COUPLING.4 (Merging of Modules) is satisfied since merging two systems together (that is combining their services and implementation elements) will not increase the overall partitioning factor of the resultant system.

Property COUPLING.5 (Disjoint Module Additivity) is satisfied since merging two disjoint systems will result in additive SPARF. This is because the number of elements not belonging to any of the services in the system/s under study will be summed up.

Table 4-11. COUP-M9: System Partitioning Factor (SPARF)

<table>
<thead>
<tr>
<th>METRIC NAME</th>
<th>SYSTEM PURITY FACTOR (SPURF)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVERVIEW</td>
<td>SPURF for a given Service-Oriented system <em>SOS</em> measures the <em>degree of purity</em> of this system in terms of all implementation elements belonging to <em>one and only one</em> service. More specifically, SPURF is the inverted ratio of the number of intersected services to the total number of the services in the system.</td>
</tr>
<tr>
<td>DESCRIPTION / MOTIVATION</td>
<td>Covers <em>Assumption CSA5</em>. A high number of service implementation elements that belong to more than one service can increase the interdependencies between different services, consequently decreasing their <em>reusability</em> and breaking the principle of <em>service-autonomy</em>, which in turn influences most sub-characteristics of maintainability. For example, the <em>stability</em> of a system will decrease since changes to the shared implementation elements can influence more than one service. Moreover, it will be difficult to physically ‘relocate’ a service <em>s</em> (that is, deploy it to different hardware or software environments) comprised of shared implementation elements $e_m...e_n$ without affecting the other services in the system that share ele-</td>
</tr>
</tbody>
</table>
ments $e_m, e_n$ with service $s$, thereby reducing system changeability.

This type of coupling is related to the notion of service-autonomy described in Section 4.1 since it reflects the conformance of a given system design to the Pure Service-Oriented System (PUSOS) defined in Section 3.3.4.

| FORMAL DEFINITION | SPURF (SOS) = 1 - $|IS (SOS)| / |SER|$, where $IS (SOS)$ is the set of all intersected services (the definition of intersection operation for services is provided in Section 3.3) in the system SOS, which can be formally expressed as: $IS (SOS) = \{\{ser1, ser2\} | ser1, ser2 \in SER \land ser1 \cap ser2 = \emptyset\}$; and $SER$ is a set of all the services ($ser$) of SOS (Table 3.1 – D6 and D6.1) |
|---|---|
| SCALE | Absolute; MEASUREMENT UNITS | Count |
| MEASUREMENT PROCEDURE | Divide the number of intersected services by the total number of services in the system, and subtract the resultant value from one in order to derive an inverted value of SPURF (purity factor). The obtained values will range from zero to one, where a value of one indicates that all the elements in the system belong to at most one service, and as such, the system can be considered as a PUSOS (Table 3.1 – D7.1). Conversely, a value of zero indicates the total absence of service autonomy in the system since all services are intersected with each other. For example, SPURF ($AMS$) = $1 - 2/3 = 0.33$ in the design shown in Figure 4-2. |
| VALIDATION | Note that similar to the SPARF metric defined above; SPURF is validated using the number of system elements as a validation construct instead of the actual coupling relationships. Property COUPLING.3 (Monotonicity) is satisfied since the coupling (as measured by SPURF) of a service-oriented system cannot be decreased by adding more implementation elements to the system disregarding of whether they are shared between different services or not. Property COUPLING.4 (Merging of Modules) is satisfied since merging two systems together (that is combining their services and implementation elements) will not increase the overall purity factor of the resultant system. In fact, SPURF will most likely decrease upon conducting a merge due to a potential reduction of shared elements. Property COUPLING.5 (Disjoint Module Additivity) is satisfied since merging two disjoint systems will result in additive SPURF. This is because the number of elements shared between the services in the system/s under study will be summed up. |

| Table 4-12. COUP-M10: System Purity Factor (SPURF) | 108 | (February, 2009) |
### OVERVIEW

RFO for a given operation \( o \) is the cardinality of the sets (total number of set elements) of service implementation elements and service interfaces that can be potentially invoked (or executed) in response to the invocations of operation \( o \) with all possible inputs.

Note that this metric can be considered as a generic measure of design coupling since it can be readily used to measure the dynamic coupling in any software design (for example, Procedural or OO designs). In fact, this metric is similar to the Response For Class (RFC) metric from the CK suite of OO metrics [44], but is defined at a lower level of granularity (at the operation level rather than class level of RFC).

### DESCRIPTION / MOTIVATION

Covers Assumption CGA2.

In contrast to all previously defined (static) metrics, RFO measures the *dynamic coupling* between service-oriented design artefacts. Although most existing coupling metrics measure the static aspects of coupling [33], dynamic aspects should also be considered given that it was previously shown that the length of the dynamic call chains (as measured by RFC) can influence the maintainability of software products [32, 45]. More specifically, a large number of the design elements interacting in order to achieve some desired functionality in response to the invocation of an operation will negatively impact the analysability sub-characteristic since the entire call chain needs to be analysed in order to understand the functioning of a given operation. Also, the stability will be affected since an element to which a given operation belongs will be dependent on an increasing number of other elements in the system.

### FORMAL DEFINITION

\[
\text{RFO} (o) = |\text{CS}(o)|, \\
\text{where } \text{CS}(o) \text{ is the set of all direct collaboration sequences } cs \text{ of operation } o.
\]

Scale: Ratio; Measurement Unit: Count

### MEASUREMENT PROCEDURE

Firstly, determine all collaboration sequences \( cs \) (refer to Table 3.1 – D5 and D5.1) that capture the set of interacting elements that achieve functionality exposed in operation \( o \) based on all possible inputs (parameter values). Secondly, combine all collaboration sequences into the overall set of collaboration sequences \( CS(o) \). Finally, count the number of distinct design artefacts (service implementation elements and service interfaces) in the \( CS(o) \) set.

To illustrate the process of calculating the RFO value for a given service interface operation \( so \) consider the following:

1) a service interface \( si2 \) (shown in Figure 4-2) has one service operation
enrollStudent (String studentID, String courseID) which will be achieved (or realised) differently based on the type of the student to be enrolled into the course (e.g. local or international student).

2) assume that the enrolment of a local student is achieved by the following collaboration sequence cs1:

   i) si2.enrollStudent(...) -> bp.enrollStudent(...);
   ii) bp.enrollStudent(...) -> si1.operationA(...);
   iii) si1.operationA(...) -> c1.operationA(...);
   iv) c1.operationA(...) -> i.operationB(...);
   v) i.operationB(...) -> c2.operationB(...);

   cs1.enrollStudent = {bp, si1, c1, i, c2}

3) assume that the enrolment of an international student is achieved by the following collaboration sequence cs2:

   i) si2.enrollStudent(...) -> bp.enrollStudent(...);
   ii) bp.enrollStudent(...) -> si1.operationA(...);
   iii) si1.operationA(...) -> p1.operationA(...);
   iv) p1.operationA(...) -> ph.operationB(...);
   v) ph.operationB(...) -> p2.operationB(...);

   cs2.enrollStudent = {bp, si1, p1, ph, p2}:

4) The set of sequences CS (si2.enrollStudent) will then consist of the elements {bp, si1, c1, i, c2, p1, ph, p2}, with RFO equalling the cardinality of this set (RFO = |CS(o)|).

That is, RFO(si2.enrollStudent) = 8 in the above example;

<table>
<thead>
<tr>
<th>VALIDATION</th>
<th>Property COUPLING.3 (Monotonicity) is satisfied since RFO cannot decrease by adding more elements to the call chain.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Property COUPLING.4 (Merging of Modules) is satisfied since merging two elements together can only decrease the amount of coupling given that the merged element will be removed from the collaboration sequence, but it will not increase.</td>
</tr>
<tr>
<td></td>
<td>Property COUPLING.5 (Disjoint Module Additivity) is satisfied since combining two unrelated elements cannot result in any of these elements being removed from the particular call chain/s of the original operations, and as such, the coupling of disjoint elements will be additive.</td>
</tr>
</tbody>
</table>

Table 4-13. COUP-M11: Response for Operation (RFO)

110

(February, 2009)
4.3.3 Aggregation Metrics

The purpose of the aggregation metrics is to combine together the relationship-based metrics defined in the previous section in order to support the quantification of coupling at a higher level of design abstraction. More specifically, the aggregation metrics are designed to measure coupling at the service level, rather than element (or service interface) level as was done in the previous sub-section. Such service-level aggregation metrics can be used to: i) compare and objectively select the alternative service structures; and ii) detect specific design problems at the service and element levels by examining the values of the individual constituent metrics as needed. Note that the service-autonomy related metrics (COUP-M9:SPARF and COUP-M10:SPURF) are not aggregated since they are already defined at the highest possible level of design abstraction (service-oriented system level).

The metrics are combined together based on their perceived influence on the particular sub-characteristics of maintainability as reflected by the coupling assumptions defined in Section 4.2. For example, the service-level metric COUP-AM3:TWOESC combines together the element-level metrics for measuring the indirect and direct outgoing extra-service coupling in order to derive the total measure of outgoing extra coupling for a service. Such measure can be used as the indicator of service stability since both indirect and direct coupling will influence the stability of services as described by the associated assumptions CSA2.2 and CSA 3.2. To this end, the impact of the aggregation metrics on the maintainability sub-characteristics is the amalgamation of the coupling assumptions covered by the individual constituent metrics. Note that the measurement procedure and validation process for the aggregation metrics are the same to those for the constituent metrics, and as such, they are not repeated in the metrics definitions.

<table>
<thead>
<tr>
<th>METRIC NAME</th>
<th>TOTAL WEIGHTED INTRA-SERVICE COUPLING OF A SERVICE (TWISC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVERVIEW</td>
<td>TWISC for a given service $s$ is a sum of all intra-service related measures for each of its implementation elements, combined with the measure of coupling between its service interface and implementation elements that directly implement this interface.</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>TWISC measures the total intra-service coupling of a service based on: 1) intra-service coupling of its implementation elements $e \in E_s$ (COUP-M1:WISCE metric), and 2) intra-service coupling of its interface $s_i$ (COUP-M2:SIEC metric). Services are intended to be independent components and thus be maintained in isolation from the system. Therefore, it is useful to measure the total coupling within a single service. A high value of TWISC can indicate bad internal design structure of a particular</td>
</tr>
</tbody>
</table>

(February, 2009)
service. For example, the ‘Billing’ service shown in Figure 4-2 has a high intra-service coupling and as such will be difficult to maintain. To this end, the internal structure of this service should be redesigned in order to decrease the intra-service coupling.

**Table 4-14. COUP-AM1: Total Weighted Intra-Service Coupling of a Service (TWISC)**

<table>
<thead>
<tr>
<th>METRIC NAME</th>
<th>TOTAL WEIGHTED INCOMING EXTRA-SERVICE COUPLING OF A SERVICE (TWIESC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVERVIEW</td>
<td>TWIESC for a given service s is a sum of all indirect (via service interface) and direct (between implementation elements) incoming extra-service measures for its constituent elements.</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>TWIESC measures the total incoming extra-service coupling of a service based on: 1) <em>indirect extra-service incoming coupling</em> of interface si_s (COUP-M3:ESICSI metric); and 2) <em>direct extra-service incoming coupling</em> of its implementation elements e (\in\ Es) (COUPM5:WESICE metric). TWIESC quantifies the dependency of the rest of the system on this service, thereby providing an indication of how critical the service is within a system wide context. To this end, this metric can be used as the indicator of service changeability. For example, the ‘Timetabling’ service shown in Figure 4-2 has a high incoming inter-service coupling. Therefore, any changes made to this service will have to be propagated to the other (two) services in the system. Note that TWIESC measures the indirect and direct incoming coupling types in combination since both types can influence the changeability of a service (refer to the descriptions of the ESICSI and WESICE metrics). Although the strength of coupling will depend on whether the incoming relationship was direct or indirect, the value of TWIESC can accommodate for the different coupling strengths since ESICSI and WESICE are weighted accordingly.</td>
</tr>
<tr>
<td>FORMAL DEFINITION</td>
<td>TWIESC(s) = (\sum_{e \in Es} WESICE(e) + ESICSI(s)),  where (Es = BPS_s \cup C_s \cup I_s \cup P_s \cup H_s) is the set of all implementation elements of service s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>METRIC NAME</th>
<th>TOTAL WEIGHTED INCOMING EXTRA-SERVICE COUPLING OF A SERVICE (TWIESC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVERVIEW</td>
<td>TWIESC for a given service s is a sum of all indirect (via service interface) and direct (between implementation elements) incoming extra-service measures for its constituent elements.</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>TWIESC measures the total incoming extra-service coupling of a service based on: 1) <em>indirect extra-service incoming coupling</em> of interface si_s (COUP-M3:ESICSI metric); and 2) <em>direct extra-service incoming coupling</em> of its implementation elements e (\in\ Es) (COUPM5:WESICE metric). TWIESC quantifies the dependency of the rest of the system on this service, thereby providing an indication of how critical the service is within a system wide context. To this end, this metric can be used as the indicator of service changeability. For example, the ‘Timetabling’ service shown in Figure 4-2 has a high incoming inter-service coupling. Therefore, any changes made to this service will have to be propagated to the other (two) services in the system. Note that TWIESC measures the indirect and direct incoming coupling types in combination since both types can influence the changeability of a service (refer to the descriptions of the ESICSI and WESICE metrics). Although the strength of coupling will depend on whether the incoming relationship was direct or indirect, the value of TWIESC can accommodate for the different coupling strengths since ESICSI and WESICE are weighted accordingly.</td>
</tr>
<tr>
<td>FORMAL DEFINITION</td>
<td>TWIESC(s) = (\sum_{e \in Es} WESICE(e) + ESICSI(s)),  where (Es = BPS_s \cup C_s \cup I_s \cup P_s \cup H_s) is the set of all implementation elements of service s</td>
</tr>
</tbody>
</table>
## METRIC NAME: 
**TOTAL WEIGHTED OUTGOING EXTRA-SERVICE COUPLING OF A SERVICE (TWOESC)**

### OVERVIEW
TWOESC for a given service $s$ is a sum of all indirect (via service interface) and direct (between implementation elements) outgoing extra-service measures for its constituent elements.

### DESCRIPTION
TWOESC measures the total outgoing extra-service coupling of a service based on: 1) *indirect extra-service outgoing coupling* of its implementation elements $e \in E_s$ (COUP-M4:EESIOC metric); and 2) *direct extra-service outgoing coupling* of its implementation elements $e \in E_s$ (COUP-M6:WESOCE metric).

TWOESC quantifies the dependency of a service on the other services in the system, and as such, it can be used as the indicator of service stability. For example, the ‘Enrolling’ service shown in Figure 4-2 has a high outgoing inter-service coupling. Therefore, there is a high possibility that changes made to the ‘Timetabling’ and ‘Billing’ services will influence the functioning of this service.

Note that TWOESC measures the indirect and direct incoming coupling types in combination since both types will influence the stability of a service (refer to the descriptions for the EESIOC and WESOCE metrics). Similarly to the previously-defined TWIESC metric, TWIESC can accommodate the different coupling strengths since the EESIOC and WESOCE metrics are weighted accordingly.

### FORMAL DEFINITION
\[
\text{TWOESC}(s) = \sum_{e \in E_s} (\text{EESIOC}(e) + \text{WESOCE}(e)),
\]
where $E_s = \text{BPS}_s \cup \text{C}_s \cup \text{I}_s \cup \text{P}_s \cup \text{H}_s$ is the set of all implementation elements of service $s$.

### Table 4-16. COUP-AM3: Total Weighted Outgoing Extra-Service Coupling of a Service (TWOESC)

<table>
<thead>
<tr>
<th>METRIC NAME</th>
<th>TOTAL WEIGHTED COUPLING OF A SERVICE (TWCS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVERVIEW</td>
<td>TWCS for a given service $s$ is a combination of all the service-level coupling measures for this service.</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>This metric quantifies the overall coupling of a service based on all possible types of coupling. In particular, the following coupling aspects are measured:</td>
</tr>
<tr>
<td></td>
<td>1) <em>intra-service</em> coupling (COUP-AM1: TWICS metric); 2) <em>incoming indirect/direct extra-service</em> coupling (COUP-AM2: TWIESC)</td>
</tr>
</tbody>
</table>
CHAPTER 4. SERVICE-ORIENTED COUPLING METRICS

metric); and 3) outgoing indirect/direct extra-service coupling (COUP-AM3: TWOESC metric)

TWCS can be used to quantify i) the quality of the internal design structure of a service (based on its intra-service measures); ii) the criticality of a service within a system wide context (based on its incoming extra-service measures); and iii) the degree of dependency of a service on the other services in the system (based on its outgoing extra-service measures).

| FORMAL DEFINITION | TWCS(s) = TWICS (s) + TWIESC (s) + TWOESC (s) |

Table 4-17. COUP-AM4: Total Weighted Coupling of a Service (TWCS)

<table>
<thead>
<tr>
<th>METRIC NAME</th>
<th>RESPONSE FOR SERVICE (RFS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVERVIEW</td>
<td>RFS for a given service s is the sum of the COUP-M11:RFO (Response for Operation) measures for each of the operations ( O(\text{si}_s) ) exposed in its service interface ( \text{si}_s ). Note that this metric is similar to the Response For Class (RFC) metric from the CK suite of OO metrics [44].</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>RFS quantifies the dynamic coupling of a service s as reflected by the number of service implementation elements and other service interfaces that can be potentially invoked (as part of collaboration sequences CS(s)) in response to all possible invocations of all operations exposed in the interface of this service. A large number of internal/external service implementation elements and other service interfaces included in the collaboration sequences of service s will have a negative effect on the analysability and stability of this service due to the strong dependency on a large number of other elements in the system. Note that the decision was made to quantify the dynamic coupling of a service based on the operations exposed in its interface since such operations will strongly influence the internal structure of a service. Moreover, service interfaces should be the primary ‘entry points’ of any service-oriented system [68], and as such, all collaboration sequences should originate from within the service interfaces.</td>
</tr>
<tr>
<td>FORMAL DEFINITION</td>
<td>( RFS (s) = \sum_{o \in O(\text{si}_s)} RFO(o), ) where ( O(\text{si}_s) ) is the set of all operations included in the interface ( \text{si}_s )</td>
</tr>
</tbody>
</table>

Table 4-18. COUP-AM5: Response for Service (RFS)
4.4 Theoretical Validation Summary

The theoretical validation of the coupling metrics derived in this research was based on the property-based software engineering measurement framework [30] described in Section 2.5.3. This framework proposes a number of axiomatic properties for the validation of the coupling metrics, where a given metric can be deemed valid if it satisfies all of the prescribed properties of coupling (defined in Section 2.5.3, Table 2-4).

In the process of the theoretical validation it was discovered that most of the derived metrics failed to satisfy the COUPLING.5 - Disjoint Module Additivity property. This is due to the different granularity of the derived metrics compared to the granularity prescribed by Briand et al. [30, 32]. More specifically, the proposed metrics only count the number of couples (relationships) between design elements without counting the frequency of such couples at the operation level. The same violation of property COUPLING.5 has been shown for the existing popular OO metrics such as, for example, Coupling Between Objects (CBO) [44]). Note however that Briand et al. [30, 32] themselves suggested that this property can be considered as overly-restrictive since it ‘forces’ the measurement of coupling at the lowest possible level of abstraction (operation level). Moreover, although CBO and other widely-used OO metrics fail this property for the reason described above, they are still considered to be useful predictors of quality characteristics of OO software based on the empirical results [32].

Table 4-19 lists the results of the validation for each of the derived metrics, where X indicates violation of a given property.

4.5 Summary

This chapter presented a set of coupling metrics for measuring the static and dynamic aspects of coupling in service-oriented designs for the purpose of predicting the quality characteristic of maintainability. The metrics, which are summarised in Table 4-20, measure various service-oriented specific and common relationships between different service-oriented design entities as captured by a formal model of service-oriented system presented in Chapter 3. Additionally, a specific type of coupling, service autonomy, is covered by measuring the conformance of a given system design to the principles of service-orientation, again based on the formalism described in Chapter 3.

The derived metrics were validated against the property-based software engineering measurement framework proposed by Briand et al. [30], and whilst some of the metrics failed to satisfy one of the five coupling properties, they are still believed to be theoretically valid since the failure of this property is due to the different granularity of some of the measurements performed in this work compared to that of Briand et al. [30], rather than a flaw in the underlying theoretical foundation of the derived metrics.
Moreover, the metrics were evaluated empirically in terms of their ability to predict the maintainability of SO software systems. The results of the empirical evaluation are presented and analysed in Chapter 6.

It is important to note that measuring other aspects of coupling (refer to Section 2.4) might result in different outcomes in terms of predicting maintainability, therefore factors like: i) the frequency of relationships between two elements; ii) the size and semantics of the data passed for a given communication [171] between design elements; iii) and the type of information flow (data or control) [63] could be investigated in future work.
CHAPTER 4. SERVICE-ORIENTED COUPLING METRICS

<table>
<thead>
<tr>
<th>METRIC NAME</th>
<th>TYPE OF COUPLING</th>
<th>DIRECTION</th>
<th>LEVEL OF ABSTRACTION</th>
<th>ASSUMPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUP-M1:WISCE</td>
<td>Intra-service</td>
<td>incoming/outgoing</td>
<td>element</td>
<td>CSA1, CCA1</td>
</tr>
<tr>
<td>COUP-M2:SIIEC</td>
<td>Intra-service</td>
<td>outgoing</td>
<td>interface</td>
<td>CSA1</td>
</tr>
<tr>
<td>COUP-M3:ESCSI</td>
<td>Indirect Extra-service</td>
<td>incoming</td>
<td>interface</td>
<td>CSA2.1</td>
</tr>
<tr>
<td>COUP-M4:ESIOC</td>
<td>Indirect Extra-service</td>
<td>outgoing</td>
<td>element</td>
<td>CSA2.2</td>
</tr>
<tr>
<td>COUP-M5:WESICE</td>
<td>Direct Extra-service</td>
<td>incoming</td>
<td>element</td>
<td>CSA3.1, CCA1</td>
</tr>
<tr>
<td>COUP-M6:WESOCE</td>
<td>Direct Extra-service</td>
<td>outgoing</td>
<td>element</td>
<td>CSA3.2, CCA1</td>
</tr>
<tr>
<td>COUP-M7:NCIS</td>
<td>Indirect/Direct Extra-service</td>
<td>incoming</td>
<td>service</td>
<td>CSA2.1, 3.1</td>
</tr>
<tr>
<td>COUP-M8:NCOS</td>
<td>Indirect/Direct Extra-service</td>
<td>outgoing</td>
<td>service</td>
<td>CSA2.2, 3.2</td>
</tr>
<tr>
<td>COUP-M9:SPARF</td>
<td>Service-autonomy</td>
<td>NA</td>
<td>system</td>
<td>CSA4</td>
</tr>
<tr>
<td>COUP-M10:SPURF</td>
<td>Service-autonomy</td>
<td>NA</td>
<td>system</td>
<td>CSA5</td>
</tr>
<tr>
<td>COUP-M11:RFO</td>
<td>Dynamic coupling</td>
<td>NA</td>
<td>operation</td>
<td>CCA2</td>
</tr>
</tbody>
</table>

**Aggregate Measures**

<table>
<thead>
<tr>
<th>METRIC NAME</th>
<th>TYPE/DIRECTION OF COUPLING</th>
<th>LEVEL OF ABSTRACTION</th>
<th>CONSTITUENT (BASE) METRICS</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUP-AM1:TWISC</td>
<td>Intra-service</td>
<td>service</td>
<td>WISCE, SIIEC</td>
</tr>
<tr>
<td>COUP-AM2:TWIESC</td>
<td>Incoming Indirect/Direct Extra-service</td>
<td>service</td>
<td>WESICE, ESICSI</td>
</tr>
<tr>
<td>COUP-AM3:TWOESC</td>
<td>Outgoing Indirect/Direct Extra-service</td>
<td>service</td>
<td>WESOCE, EESIOC</td>
</tr>
<tr>
<td>COUP-AM4:TWSC</td>
<td>Intra-service and Incoming/Outgoing Indirect/Direct Extra-service</td>
<td>service</td>
<td>TWISC, TWIESC, TWOESC</td>
</tr>
<tr>
<td>COUP-AM5:RFS</td>
<td>Dynamic coupling</td>
<td>service</td>
<td>RFO</td>
</tr>
</tbody>
</table>

Table 4-20. Summary of the coupling metrics

Finally, note that although SO and Procedural/OO systems exhibit different design characteristics as discussed in Chapter 3, the existing OO coupling metrics (such as the CBO and RFC metrics proposed by Chidamber and Kemerer [44]) can be adapted to measure the coupling of individual implementation elements as part of a common coupling (as was shown in Section 4.3.2 – metrics COUP-M1 and COUP-M11). Additionally, a number of metrics have been proposed for distributed and concurrent applications in general [168, 207, 228]. These
metrics can also be used to measure the common coupling in SO designs. However, although service-oriented applications could be considered as a subset of distributed applications [18], such metrics are not immediately applicable to service-oriented specific relationships since they were defined at an abstract level independent of specific implementation architecture or development paradigm. Consequently, existing metrics cannot be readily used to measure service-oriented specific coupling.
Chapter 5. Service-Oriented Cohesion Metrics

This chapter examines the structural property of cohesion in the context of SOC. More specifically, this chapter presents a taxonomy of qualitative service-oriented (SO) cohesion categories that can be used to characterise the inherently semantic property of cohesion. This in turn, supports the unambiguous derivation of metrics for quantifying cohesion in service-oriented software designs in order to predict the maintainability of software products according to a set of proposed cohesion assumptions that establish explicit links between the different aspects of service-oriented design cohesion and the maintainability of SO software. The service-oriented cohesion metrics, combined with the coupling metrics presented in Chapter 4, constitute the core contribution of this thesis.

This chapter is organised as follows. Section 5.1 provides a brief overview of existing semantic categories of cohesion, thereby providing a foundation for the material presented in this chapter. Section 5.2 presents eight proposed semantic categories of SO cohesion, with the metrics for quantifying the cohesion categories defined and theoretically validated in Section 5.3. Two cohesion assumptions that link the categories of cohesion, as measured by the derived metrics, to the sub-characteristics of maintainability are then presented in Section 5.4. Finally, Section 5.5 summarises the service-oriented cohesion categories and metrics presented in this chapter.

5.1 Overview

Cohesion is considered to be one of the most difficult to measure structural properties of software due to its inherently semantic nature [31, 75, 251]. Also, the current empirical understanding of the notion of cohesion is not as advanced as the understanding of other structural properties such as coupling and complexity [31]. Nevertheless, the concept of cohesion has been widely discussed in the context of the Procedural and OO paradigms with various qualitative classification schemes being proposed to describe different categories of software cohesion. For example, Stevens et al. [222] proposed six widely-accepted semantic categories of procedural module cohesion (classical cohesion [63]) that were later elaborated by Yourdon and Constantine [242]. They are: Coincidental, Logical, Temporal, Communicational, Sequential, and Functional. Additionally, the categories of classical cohesion have been later redefined and extended by Eder et al. [63] in order to cover the conceptual and technological aspects introduced by the OO paradigm (refer to Section 2.4.3 for the full description of Stevens’ et al. and Eder’s et al. categories).

The existing semantic categories of classical and OO cohesion are redefined and extended in this chapter (Section 5.2) in order to account for the distinguishing characteristics of SOC.
CHAPTER 5. SERVICE-ORIENTED COHESION METRICS

Defining such categories is important since they encapsulate the common understanding of the notion of software cohesion, thereby providing a strong conceptual foundation for the derivation of quantitative cohesion metrics. More specifically, the semantic categories of cohesion have been used in this research to drive the definition of service-oriented cohesion metrics (Section 5.3) using the formalism prescribed by the model of SO software designs (presented in Chapter 3) in an unambiguous and formal manner consistent with the principles of measurement theory. Additionally, the semantic categories of cohesion are used by the SO cohesion assumptions (Section 5.4) that link the proposed categories of cohesion, as measured by the cohesion metrics, to the sub-characteristics of software maintainability.

The derived cohesion metrics, which are theoretically validated against the required cohesion properties specified in the property-based software engineering measurement framework of Briand et al. [30], are needed since it was shown previously that the process of assigning design artefacts to different cohesion categories has a subjective nature, and thus cannot be fully automated [63]. Therefore, more recent research in the area of software cohesion (for OO software) has focussed on the definition of quantitative metrics that support automated quantification of cohesion [15, 31, 44, 50, 102, 146, 150] (refer to Section 2.4).

Note that as with the other structural properties of software (such as coupling), the structural property of cohesion can be quantified at different levels of product abstraction. The derived SO cohesion metrics measure cohesion at the design level so to be consistent with the main goal of this research, which is to predict the maintainability of SO software as early in the SDLC as possible. Measuring software cohesion at the design stage is generally considered to be a difficult task because data available during the design stage is limited [51]. Therefore, most existing OO metrics measure implementation-level cohesion, and as such, are not applicable at the design stage. Nonetheless, the importance of measuring cohesion prior to commencing the implementation of software products has been argued by a number of researchers [51, 179], with some of the newly-proposed OO metrics targeting the measurement of cohesion at the design level. For example, the Cohesion Among Methods in a Class (CAMC) metric measures the degree of correspondence between the parameter types across the methods in an OO class and can be calculated during the design phase [15].

5.2 Service-Oriented Cohesion - Definition and Taxonomy

This section describes the concept of service-oriented cohesion and also redefines the classical categories of software cohesion proposed for procedural software [222] using a service, as reflected by the operations exposed in its service interface, as the primary design construct for evaluating software cohesion. The classical categories of (procedural) cohesion provide a strong basis for the SO cohesion categories because conceptually SOC shares similarities with the procedural paradigm [68, 104].
In general, the cohesion of a service can be assessed using two different SO design constructs: *service implementation elements* and *service interfaces* as follows:

Firstly, the cohesion of a service can be assessed based on the *cohesiveness of its constituent implementation elements*. That is, the cohesion of individual service implementation elements can be quantified and combined together to approximate the overall cohesion of a service. The existing cohesion metrics (refer to Section 2.5.4) can be readily used to measure the cohesion of individual implementation elements, and as such, we do not introduce new element-level cohesion metrics in this research. More importantly, the element-level cohesion can only be fully measured when the software product was already implemented, which contradicts our goal of predicting the maintainability of SO software as early in the SDLC as possible.

Secondly, the cohesion of a service can be assessed based on the *conceptual relatedness of the operations exposed in its service interface* (Figure 5-1 provides the definition of service cohesion used in this thesis). This type of service cohesion cannot be readily measured using the existing metrics due to the conceptual and technological differences introduced by SOC (described in Sections 2.2 and 3.2), and as such, new metrics are proposed in this chapter. The decision was made to measure the cohesion of a service based on the relatedness of the operations exposed in its interface because the identification of service interfaces is considered to be one of the most important SO design activities given that interface granularity and cohesion of operations exposed in a service interface are expected to strongly influence the overall structure of a service [68, 104]. More specifically:

i) Service interfaces are the primary design constructs for determining service boundaries as was described in Chapter 3.

ii) Service interfaces should be the primary entry (or interaction) points of a system [183]. For example one of the main motivations for using SOAP-based Web Services, is that they are accessed through a language and location independent interface thus promoting loose technological coupling [4].

iii) Service interfaces should not be frequently changed so as to avoid potential problems on the service consumer (client) side [173, 220]. That is, service interfaces should be highly stable since changes to the interface can potentially influence a large number of consumers.

To provide a theoretical framework for analysing the newly-introduced concept of *service interface* cohesion, the existing categories of classical (procedural) cohesion are redefined in this section in terms of SO terminology. In addition to redefining the categories of classical cohesion, two new categories of software cohesion, External and Implementation, are introduced:
The cohesion of a service $s$ in a service-oriented system design is the measure of the degree to which the operations exposed in its service interface $s_{\text{is}}$ belong together conceptually.

**Figure 5-1. General definition of service cohesion**

- **External** cohesion reflects the cohesiveness of a service interface based on the usage of its operations with respect to (external) service consumers. The External cohesion category was introduced in order to capture the behavioural aspects of service interface cohesion that can indicate the relatedness of service operations.

- **Implementation** cohesion reflects the cohesiveness of a service interface based on the relatedness of the service implementation elements used to realise service operations. More specifically, the Implementation cohesion category covers the concepts encapsulated by the categories of OO cohesion [63] that capture the semantic notion of cohesion in OO designs.

In total, eight categories of service cohesion have been defined: Coincidental, Logical, Temporal, Communicational, External, Implementation, Sequential, and Conceptual. The categories are defined on a relaxed (or incomplete) ordinal scale [31]. That is, the first three cohesion categories (Coincidental, Logical, and Temporal) and the last category (Conceptual) are defined on an ordinal scale, being ranked based on their perceived strength of cohesion. The remaining four categories (Communicational, External, Implementation, and Sequential) are grouped together and ranked only in respect to the other categories, being deemed to represent stronger service cohesiveness than Temporal cohesion, but weaker cohesiveness than the Conceptual category. As such, these four categories are defined on a nominal scale in respect to one another. This is because it is difficult to objectively rank these four categories given that they are not mutually-exclusive, with each category positively influencing the overall cohesion of a service in a distinct manner.

Note however that the complete ordinal ranking could be established based on comprehensive empirical studies that involve the investigation of a large number of service-oriented systems where each category is evaluated independently from one another. Such empirical investigation was considered outside of the scope of this research and should be conducted in future work. Also note that the classical categories of cohesion are also subject to this incomplete ordinal scale constraint [31] due to the high degree of difficulty associated with testing and verifying the expected effect of a particular category on the cohesiveness of a service.

It is important to note that the above classification of cohesion, defined as the mixture of nominal and ordinal scales, is not particularly useful from the measurement perspective because the formal relations associated with the ordinal scale (for example, ‘>’ and ‘<’) cannot be applied directly to the nominal categories of cohesion. Nevertheless, the categories are
useful since they provide conceptual support for the derivation of the quantitative cohesion metrics presented in Section 5.3.

The eight proposed categories of service-oriented cohesion are described below. The descriptions of categories include original definitions of the classical and OO cohesion categories (where applicable) for comparison purposes, and also illustrate example services belonging to a particular category of cohesion.

**COH-CAT1: Coincidental Cohesion**

**ORIGINAL DEFINITION** [222]
There are no meaningful relationships between elements of a module.

**SERVICE-ORIENTED DEFINITION**
A service encapsulates unrelated functionality insofar as there are no semantically meaningful relationships between any of the operations exposed in its service interface.

Note that the Coincidental category of cohesion is entirely semantic in nature and is difficult to quantify by examining the structural properties of SO designs.

An example Coincidental service (‘Academic Management System’ (AMS)) is shown below. This service exhibits Coincidental cohesion since none of the operations exposed in the AMS interface are related with one another in any semantically meaningful way.

<table>
<thead>
<tr>
<th>AMS interface</th>
<th>AMS interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>createStudentTranscript (p1, p2)</td>
<td>createStudentTranscript (p1, p2)</td>
</tr>
<tr>
<td>p1: String stID</td>
<td>p1: String stID</td>
</tr>
<tr>
<td>p2: String pswd</td>
<td>p2: String pswd</td>
</tr>
<tr>
<td>changeCourseStructure (p3)</td>
<td>changeCourseStructure (p3)</td>
</tr>
<tr>
<td>p3: XML complex type Course</td>
<td>p3: XML complex type Course</td>
</tr>
<tr>
<td>arrangeAnnualLeave (p4, p5)</td>
<td>arrangeAnnualLeave (p4, p5)</td>
</tr>
<tr>
<td>p4: XML complex type Staff</td>
<td>p4: XML complex type Staff</td>
</tr>
<tr>
<td>p5: Date startDate</td>
<td>p5: Date startDate</td>
</tr>
</tbody>
</table>

**LEGEND:**
- Service
- Implementation Element
- Service Interface
- Service Operation
- Client/Consumer

Table 5-1. COH-CAT1: Coincidental Category of SO cohesion

(Febuary, 2009)
CHAPTER 5. SERVICE-ORIENTED COHESION METRICS

**COH-CAT2: LOGICAL COHESION**

**ORIGINAL DEFINITION [222]**
Elements with similar functionality such as input/output handling are collected in one module.

**SERVICE-ORIENTED DEFINITION**
Service operations provide common functionality such as, for example, data update or retrieval.

Note that the Logical category cannot be distinguished from Coincidental cohesion without semantic knowledge of the problem domain. Also, as with Coincidental cohesion, Logical cohesion is difficult to quantify by examining the structural properties of SO designs.

The interface for the ‘Reporting’ service shown below illustrates an example of Logical cohesion where the service interface operations are related only because they provide similar reporting functionality.

![Reporting Interface](generateTaxReport ()
generateStaffPerformanceReport ()
generateEnrollmentStatisticsReport ()

Table 5-2. COH-CAT2: Logical category of SO cohesion

**COH-CAT3: TEMPORAL COHESION**

**ORIGINAL DEFINITION [222]**
The elements of a module have logical cohesion, but they are also related in time (executed in the same, or a single defined, time period).

**SERVICE-ORIENTED DEFINITION**
Service operations provide common functionality (as captured by Logical cohesion) and are performed within a predefined time period.

System initialising/terminating and other ‘housekeeping’ operations are examples of Temporal cohesion.

Note that the Temporal category could be defined independently of the Logical category given that service operations can be performed in the same time period without exhibiting Logical cohesion. Nevertheless, in this thesis the decision was made to define Temporal cohesion as a more restricted version of Logical cohesion so as to be consistent with the original classical categories of cohesion. Additionally, as with the Logical category defined
above, Temporal cohesion cannot be distinguished from Coincidental cohesion without semantic knowledge of the problem domain, and thus is also difficult to quantify by examining the structural properties of SO designs.

The interface for the ‘Initialising’ service shown below illustrates an example of Temporal cohesion where the operations are related only because they provide similar initialising functionality performed within the same time period (for example, during system start-up).

<table>
<thead>
<tr>
<th>Initialising Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialiseStudentDB ()</td>
</tr>
<tr>
<td>initialiseStaffDB ()</td>
</tr>
<tr>
<td>initialiseAcademicDB ()</td>
</tr>
</tbody>
</table>

Table 5-3. COH-CAT3: Temporal category of SO cohesion

**COH-CAT4: COMMUNICATIONAL COHESION**

**ORIGINAL DEFINITION [222]**

The elements of a module are related by a reference to the same set of input and/or output data.

**SERVICE-ORIENTED DEFINITION**

*Service operations operate on the same shared data abstractions.*

Note that the cohesiveness of data abstractions themselves is important. For example, the Student data abstraction used in the example below might be composed of unrelated attributes, and as such, a service (Student Management) may be incorrectly deemed to exhibit Communicational cohesion. In this case, it may be difficult to achieve the highest level of cohesion (Conceptual level).

The interface for the ‘Student Management’ service shown below illustrates an example of Communicational cohesion where the operations are related to one another because they operate on the same data abstraction (they have a common input parameter type - Student).

<table>
<thead>
<tr>
<th>StudentManagement Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>calculateFees (Student student)</td>
</tr>
<tr>
<td>viewPersonalDetails (Student student)</td>
</tr>
<tr>
<td>createAcademicTranscript (Student student)</td>
</tr>
</tbody>
</table>

Table 5-4. COH-CAT4: Communicational category of SO cohesion
**COH-CAT5: EXTERNAL COHESION**

**ORIGINAL DEFINITION**
N.A. – This is a new category that was introduced in order to capture additional behavioural aspects of service interface cohesion.

**SERVICE-ORIENTED DEFINITION**
*Service operations are used in combination by service consumers (clients).*
Traditionally, cohesion is considered to be a property related to the internal structure of a software module [24]. However, it has been suggested in a more recent study [163] that cohesion can also be interpreted as an externally observed property without regard for the internal structure of a module. As such, cohesion can also be determined by examining the similarity of a module’s usage patterns from external clients [163]. Therefore, the External cohesion category has been introduced in this research in order to capture the behavioural properties of service interfaces.

The interface for the ‘Student Management’ service shown below illustrates an example of External cohesion where all service interface operations are invoked from a given client (Student).

![Diagram of Student Service and Student Interface](image)

Table 5-5. COH-CAT5: External category of SO cohesion

**COH-CAT6: IMPLEMENTATION COHESION**

**ORIGINAL DEFINITIONS [63]**
This category is related to the first three existing categories of OO cohesion defined by Eder et al. [63] and summarised below (refer to Section 2.4.3 for the complete definitions of the categories):

- **Separable**: the objects of a class represent multiple unrelated data abstractions.
- **Multifaceted**: the objects of a class represent multiple related data abstractions.
- **Non-delegated**: there exist attributes which do not describe the whole data abstraction represented by a class.
The Implementation category of cohesion was introduced in order to capture the aspects of service interface cohesion related to the underlying implementation of a service. More specifically, Implementation cohesion is conceptually related to the OO categories of cohesion that classify OO classes into cohesion categories based on the degree to which a given class represents some concrete single data abstraction as reflected by its class methods using the same class attributes. Such classification can be redefined, in order to cover for the extra level of abstraction introduced by SOC, in terms of the service interface operations being implemented by the same implementation elements. Note that the decision was made to cover the first three categories of OO cohesion in combination since they are all related to the same aspect of design cohesion. That is, the Separable, Multifaced, and Non-delegated categories can be reflected by the different levels of Implementation cohesion. Note that the remaining two categories of OO cohesion, Model and Concealed [63], capture different aspect of (semantic) cohesion and are covered separately as part of the Conceptual category presented later in this section.

**SERVICE-ORIENTED DEFINITION**

Service interface operations are implemented by the same implementation elements.

The ‘Enrollment’ service shown below exhibits Implementation cohesion since two operations exposed in its service interface are implemented by the same elements.

![Enrollment Interface](image)

Table 5-6. COH-CAT6: Implementation category of SO cohesion

---

### COH-CAT7: SEQUENTIAL COHESION

**ORIGINAL DEFINITION [222]**

The elements of a module are connected by a sequential control flow, where the output from an element is the input for the next element.

**SERVICE-ORIENTED DEFINITION**

Service operations are sequentially related insofar as either the output or post-condition from one operation serves as the input or pre-condition for the next operation. Note that although it is highly likely that service operations connected sequentially via out-
put and input values will be invoked from the same client (e.g. a business process script), in practice it is possible to have sequential dependencies from multiple clients as shown in the example below where two different clients invoke operations of the ‘Course Management’ service that are sequentially connected via both output/input values and post-/pre-conditions.

![Sequential dependencies diagram]

Table 5-7. COH-CAT7: Sequential category of SO cohesion

<table>
<thead>
<tr>
<th>COH-CAT8: CONCEPTUAL COHESION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORIGINAL DEFINITION (Procedural paradigm - Functional cohesion category [222]): All elements of the module are related to the performance of a single function (all elements contribute to accomplishing a single goal).</td>
</tr>
<tr>
<td>ORIGINAL DEFINITION (OO paradigm [63])</td>
</tr>
<tr>
<td>- Model: The class represents a semantic model of a problem domain entity.</td>
</tr>
<tr>
<td>- Concealed: There exists some useful data abstraction concealed in the data abstraction represented by the class.</td>
</tr>
<tr>
<td>Note that the Concealed category can be considered as a specialised case of Model cohesion, and as such, it is not covered explicitly by the Conceptual categories of cohesion.</td>
</tr>
<tr>
<td>SERVICE-ORIENTED DEFINITION</td>
</tr>
<tr>
<td><em>There is a meaningful semantic relationship between all operations of a service in terms of some identifiable domain level concept.</em> More specifically, the operations of a service contribute to either single business functionality or some other semantically meaningful concept such as an abstraction or data entity in the problem domain.</td>
</tr>
</tbody>
</table>
| This category represents the strongest type of service cohesion, given that it covers both the Functional and Model categories of classical and OO cohesion, which are considered to
be the strongest categories of cohesion in the Procedural and OO paradigms respectively. Note that although both types of cohesion (Functional/Model) are conceptually strong, OO or data entity type abstractions should arguably be avoided in pure SO designs [67] since a service should expose some aspect of a business function, rather than operations related to a particular domain-level abstraction.

Note that the Conceptual category of cohesion is semantic in nature and is difficult to quantify by examining the structural properties of SO designs. This is consistent with the original definitions of the Functional and Model cohesion categories that are inherently semantic. For example, Stevens et al. [222] suggested a non-automated approach for determining whether a module is functionally cohesive based on writing a textual description describing the purpose of the module, and then examining this description based on suggested heuristics for determining the Functional category of cohesion.

The ‘Enroll Student’ service shown below exhibits Conceptual cohesion since all the operations are related to a single domain-level concept, in this case a concrete business functionality (enrolling a student into a course).

![Enroll Student Service Diagram]

Table 5-8. COH-CAT8: Conceptual category of SO cohesion

5.2.1 Summary of the Cohesion Categories

This section presented eight categories of service cohesion which encapsulate the semantic understanding of the notion of software cohesion in SO software designs, thereby providing a conceptual foundation for the derivation of quantitative cohesion metrics in Section 5.3. Some of the proposed categories of service cohesion cannot be mapped directly to the structural constructs of service-oriented designs without examining service (and service invocation) semantics. Such purely semantic cohesion categories include the Coincidental, Logical, Temporal, and Conceptual categories of SO cohesion. In contrast, the Communicational, External, Implementation, and Sequential cohesion categories can be mapped to the constructs
of SO designs and quantified using the service-oriented cohesion metrics as is done in this chapter. As such, these categories represent quantifiable cohesion categories.

Note that the main goal of the cohesion metrics presented in this chapter is to quantify the overall cohesiveness of a service in SO system design; therefore, in addition to measuring the four quantifiable cohesion categories, we are also interested in determining the lack of cohesion (Coincidental cohesion), as well as the total existence of service cohesion (Conceptual cohesion). The Coincidental and Conceptual categories of cohesion are purely semantic categories, and thus difficult to quantify by examining the structural properties of SO designs. Nevertheless, we believe that the Coincidental and Conceptual could also be indirectly quantified according to the following two suppositions\(^\text{21}\), which are evaluated in the empirical study presented in Chapter 6.

**Cohesion Supposition CS1 – Coincidental cohesion.**
Coincidental cohesion could be reflected by the quantifiable cohesion categories as follows: i) there are no meaningful relationships between operations exposed in a service interface insofar as none of the operations work on the shared data abstractions (total lack of Communicational cohesion); ii) all service clients invoke one service operation at the most (total lack of External cohesion); iii) all operations are implemented by different implementation elements (total lack of Implementation cohesion); and iv) there are no sequential dependencies between any of the operations (total lack of Sequential cohesion).

**Cohesion Supposition CS2 – Conceptual cohesion.**
As with the Coincidental cohesion category described above, Conceptual cohesion could also be indirectly reflected by the quantifiable cohesion categories, with a service that possesses the best possible degrees of all quantifiable cohesion categories (Communicational, External, Implementation, and Sequential) could be classified as a Conceptually cohesive service.

Note that the Logical and Temporal categories of cohesion cannot be distinguished from Coincidental cohesion at the design stage without semantic knowledge of the problem domain, and as such, these categories are not covered by the suppositions.

Table 5-9 provides a summary of the service-oriented categories of cohesion, including the ordinal ranking of the categories. Note that the Communicational, External, Implementation, and Sequential cohesion categories are not ranked against each other as described previously. These categories are presented in the alphabetical order without any implied ordinal ranking.

\(^{21}\) We use the term *supposition* instead of *assumption* in order to differentiate between the main coupling and cohesion assumptions presented in this thesis, which are related to the influence of the metrics on software maintainability, and the (local) cohesion assumptions or *suppositions* specific to the categories of cohesion.
CHAPTER 5. SERVICE-ORIENTED COHESION METRICS

<table>
<thead>
<tr>
<th>SO COHESION CATEGORY</th>
<th>RANK (ORDINAL SCALE)</th>
<th>COHESION TYPE</th>
<th>MEASURABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>COINCIDENTAL</td>
<td>1</td>
<td>UNDESIRABLE</td>
<td>INDIRECTLY REFLECTED BY THE COMMUNICATIONAL, EXTERNAL, SEQUENTIAL, AND IMPLEMENTATION CATEGORIES</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>2</td>
<td>UNDESIRABLE</td>
<td>-</td>
</tr>
<tr>
<td>TEMPORAL</td>
<td>3</td>
<td>UNDESIRABLE</td>
<td>-</td>
</tr>
<tr>
<td>COMMUNICATIONAL</td>
<td></td>
<td></td>
<td>DIRECTLY</td>
</tr>
<tr>
<td>EXTERNAL</td>
<td>4</td>
<td>DESIRABLE</td>
<td>DIRECTLY</td>
</tr>
<tr>
<td>IMPLEMENTATION</td>
<td></td>
<td></td>
<td>DIRECTLY</td>
</tr>
<tr>
<td>SEQUENTIAL</td>
<td></td>
<td></td>
<td>DIRECTLY</td>
</tr>
<tr>
<td>CONCEPTUAL</td>
<td>5</td>
<td>DESIRABLE</td>
<td>INDIRECTLY REFLECTED BY THE COMMUNICATIONAL, EXTERNAL, SEQUENTIAL, AND IMPLEMENTATION CATEGORIES</td>
</tr>
</tbody>
</table>

Table 5-9. Categories of service-oriented cohesion – Summary

5.3 Metrics Definitions

The metrics presented in this section can be used to measure the cohesion of a service in the service-oriented design based on the operations exposed in its interface. More specifically, the metrics are designed to support the automatic allocation of services to the categories of service-oriented cohesion defined in Section 5.2. That is, the derived metrics describe which structural constructs of service-oriented designs can be measured when evaluating the cohe-
CHAPTER 5. SERVICE-ORIENTED COHESION METRICS

sion of a service according to the quantifiable cohesion categories, and also provide a con-
crete measurement mechanism describing how to measure these structural constructs.

Note that there are two technological constraints related to the structural properties of ser-
vice interfaces that were taken into consideration when defining the metrics: i) there are no data attributes encapsulated within service interfaces; and ii) the implementation of service interface operations is not provided in the actual service interface.

The proposed metrics are presented below in terms of:

i) Informal description of the metric that covers the motivation behind its derivation based on the associated cohesion categories of cohesion;

ii) A formal definition of the metric based on the formalism captured by the model of service-orientated software designs presented in Chapter 3;

iii) Measurement procedure that describes the process of metric collection;

iv) Theoretical validation, which is based on the property-based software engineering measurement framework [30], where a given metric can be deemed valid if it conforms to the prescribed mathematical characteristics of cohesion (shown in Section 2.5.2, Table 2-4).

<table>
<thead>
<tr>
<th>METRIC NAME</th>
<th>SERVICE INTERFACE DATA COHESION (SIDC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVERVIEW</td>
<td>The SIDC metric quantifies cohesion of a given service s based on the cohesiveness of the operations sop ∈ SOp(si) exposed in its service interface si, as reflected by all service operations:</td>
</tr>
<tr>
<td></td>
<td>i) having common parameter types Param(sop ∈ SOp(si))</td>
</tr>
<tr>
<td></td>
<td>ii) having the same return type returnType_{sop}</td>
</tr>
<tr>
<td></td>
<td>Refer to Chapter 3, Table 3.1–D2 and D2.1 for the formal definitions of the above constructs.</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>SIDC is designed to directly quantify the Communicational cohesion category, as well as indirectly reflect the Coincidental and Conceptual categories of cohesion.</td>
</tr>
<tr>
<td></td>
<td>Given that there are no attributes included in the service interfaces, the relatedness of the parameter and return types of service operations was considered as the only tangible mechanism for indicating the Communicational cohesion. Note that this metric has a limitation related to the usage of the standard data types (for example, Strings or Integers), which can result in artificially high Communicational cohesion. This is because the parameter/return types can be good indicators of Communicational cohesion only when such types represent some custom data abstractions (e.g. complex XML types used by Web Service interfaces), but this is</td>
</tr>
</tbody>
</table>
not necessarily the case for the standard data types. For example, consider two following service operations where both operations use the \textit{String} parameter type, but the actual String values represent unrelated data abstractions:

\begin{verbatim}
getPersonalInfo(String studentID)
arrangeAnnualLeave(String staffID)
\end{verbatim}

A service is deemed to be data cohesive when all possible pairs of service operations \( (SOp(s_i)) \) have at least one common parameter and return type. The values of SIDC will range from 0 to 1, with the value 1 representing the strongest possible data cohesion indicating that the service belongs to the \textit{Communicational} category of cohesion and also contributing to the decision of whether this service belongs to the \textit{Conceptual} category (refer to Section 5.2.1 - Supposition CS2); and the value 0 representing total lack of data cohesion, thereby suggesting that the service could potentially belong to the \textit{Coincidental} category of cohesion (refer to Section 5.2.1 - Supposition CS1). The values in between 0 and 1 suggest different levels of Communicational cohesion.

Finally note that SIDC is conceptually related to the OO parameter-based cohesion metrics (Cohesion Among Methods in a Class (CAMC) [15] and Normalised Hamming Distance (NHD) [51] metrics), which evaluate the cohesion of OO classes based on the degree of correspondence between the parameter types across each of the methods in an OO class. Therefore, the SIDC metric can be considered as the extended version of the CAMC and NHD metrics where the degree of correspondence between the return types of service operations is also covered.

### FORMAL DEFINITION

\[
\text{SIDC (s)} = \frac{\text{Common (Param(SOp(s_i)))}}{\text{Common (returnType(SOp(s_i)))}} + \frac{\text{Common (returnType(SOp(s_i)))}}{\text{Total (SOp(s_i))}} * 2,
\]

where

i) \(\text{Common (Param(SOp(s_i)))}\) is the function that calculates the number of service operation pairs that have at least one input parameter type in common; and \(\text{Common (returnType (SOp(s_i)))}\) is the function that calculates the number of service operation pairs that have the same return type;

ii) \(\text{Total(SOp(s_i))}\) is the function that returns the number of all possible combinations of operation pairs for the service interface \(s_i\)

Scale: Absolute; Measurement Unit: Count

### MEASUREMENT PROCEDURE

The measurement procedure involves the following steps:

1.A) Compare the sets of parameter types \(\text{Param(sop} \in \text{SOp(s_i))}\) for each service operation \(\text{sop} \in \text{SOp(s_i)}\) in a pair-wise manner, and then place the operation pairs that share at least one common parameter type into a set of common operation pairs \(\text{CP}\). The cardinality of the set \(\text{CP}\) is
then calculated and the resultant value returned by the function $Common(Param(SOp(si_i)))$.

1.B) Compare the return types $returnType(sop \in SOp(si_i))$ for each service operation $sop \in SOp(si_i)$ in a pair-wise manner, and then place the operation pairs that share a common return type into a set of common operation pairs $CR$. The cardinality of the set $CR$ is then calculated and the resultant value returned by the function $Common(returnType(SOp(si_i)))$.

2) Determine the number of all possible service operation pairs ($TP$) which will be returned by the function $Total(SOp(si_i))$. The value of $TP$ can be easily calculated as follows: Let $n$ be a total number of operations of a service. The number of possible service operation pairs ($TP$) can then be calculated as $TP = (n-1)*n/2$. For example, a service with three operations $a$, $b$, and $c$ has three possible combinations $(a, b)$, $(a, c)$, $(b, c)$.

3) Sum the values returned by the $Common(Param(SOp(si_i)))$ and $Common(returnType(SOp(si_i)))$ functions in steps 1.A and 1.B in order to determine the overall data commonality of the service operations based on both the parameter and return types.

4) Divide the value obtained in step 3 by (the number of all possible service operation pairs as calculated in step 2, multiplied by two) in order to derive the final value of SIDC.

Note that a service operation $sop1 \in SOp(si_i)$ will be excluded from the calculation performed by the $Common(Param(SOp(si_i)))$ and $Total(SOp(si_i))$ functions if this operation has no input parameters ($Param(sop1) = \emptyset$). Accordingly, a service operation $sop1 \in SOp(si_i)$ will be excluded from the calculation performed by the $Common(returnType(SOp(si_i)))$ and $Total(SOp(si_i))$ functions if this operation has no return types ($returnType(sop1) = \text{null}$).

Also note that the value of SIDC will be set to unknown in the case a service interface consists of one operation only ($|SOp(si_i)| = 1$) since SIDC is calculated based on a pair-wise comparison of service operations, and as such, it can only be calculated when the service interface has more than one service operation.

The following example illustrates the calculation of SIDC measure for the service interface of the sample Academic Management System (AMS) service shown below.

**AMS Interface**

```
int calculateFees (Student student)
String getStaffDetails (String staffID)
String createAcademicTranscript (Student student, String program, int fees)
```
CHAPTER 5. SERVICE-ORIENTED COHESION METRICS

| 1.A) $|\text{CP}| = |\{\text{calculateFees, createAcademicTranscript}\}, \{\text{getStaffDetails, createAcademicTranscript}\}| = 2$ |
| 1.B) $|\text{CR}| = |\{\text{getStaffDetails, createAcademicTranscript}\}| = 1$ |

2) number of service operations ($n = 3$; $TP = ((n-1)^*n)/2 = 3$)

3) $|\text{CP}| + |\text{CR}| = 2+1 = 3$

4) $\text{SIDC (AMS)} = 3 / (TP^*2) = 3 / (3^*2) = 0.5$

**VALIDATION**

Property COHESION.1 (Non-negativity and Normalisation) is satisfied since SIDC values for a given service interface will never be negative under any circumstances, and the values of SIDC will always belong to a specified interval $[0, \text{MAX}]$ as required by the normalisation characteristic, with MAX equalling one (1) for this and all the consequent metrics. Note that normalisation allows meaningful and direct comparisons between the cohesion values of different services since the values belong to the same interval.

Property COHESION.2 (Null Value) is satisfied since SIDC for a given service interface will be null (or zero) when the operations exposed in this interface have no common parameter or return types.

Property COHESION.3 (Monotonicity) is satisfied since adding a common parameter or return type to a pair of service operations will not decrease the overall cohesion of this service (it will most likely increase it).

Property COHESION.4 (Cohesive Modules) is satisfied since the cohesion of a service interface $\text{s}_{imn}$ obtained by joining together two unrelated service interfaces $\text{s}_i$ and $\text{s}_n$ (insofar there are no common parameter or return types in-between the operations belonging to these interfaces) will not be greater than the maximum cohesion of the original interfaces $\text{s}_i$ and $\text{s}_n$.

Table 5-10. COH-M1: Service Interface Data Cohesion (SIDC)

<table>
<thead>
<tr>
<th>METRIC NAME</th>
<th>SERVICE INTERFACE USAGE COHESION (SIUC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVERVIEW</td>
<td>The $SIUC$ metric quantifies cohesion of a given service $s$ based on the cohesiveness of the operations $\text{sop} \in SOp(\text{s}_i)$ exposed in its interface $\text{s}_i$, as reflected by all service operations being invoked (used) by every client of this service as captured by the set of service incoming relationships $SIR(s)$. Refer to Chapter 3, Table 3.1–D4.5 for the formal definition of $SIR$.</td>
</tr>
</tbody>
</table>
SIUC quantifies the usage patterns of service operations, thereby being directly related to the *External* category of cohesion, as well as indirectly influencing the *Coincidental* and *Conceptual* categories of cohesion.

A service is deemed to be Externally cohesive when *all* of its service operations \( (SOp(s_i)) \) being invoked by *all* the clients of this service. Clients can be other services in the system, but in theory, any piece of executable software can be considered as a client if it invokes operation of services under study. For example, a typical service client (consumer) would be a business process script (BPEL4WS script).

The information required to calculate this metric can be obtained from the behavioural design artefacts such as UML sequence diagrams, business process workflows, and procedural flowcharts.

As with the COH-M1:SIDC metric, the values of SIUC will range from 0 to 1, with the value 1 representing the strongest possible usage cohesion indicating that the service belongs to the *External* category of cohesion and also contributing to the decision of whether this service belongs to the *Conceptual* category; and the value 0 representing total lack of usage cohesion, thus suggesting that the service could be potentially assigned to the *Coincidental* category of cohesion. The values in between 0 and 1 suggest different levels of External cohesion.

Note that the interpretation of obtained values for the remaining three metrics follow the same principle, where metric values range from 0 to 1, with value 1 representing the strongest possible cohesion of a particular type (and also suggesting *Conceptual* cohesion); and value 0 indicating total lack of particular type of cohesion, thus suggesting *Coincidental* cohesion.

<table>
<thead>
<tr>
<th>FORMAL DEFINITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>( SIUC(s) = Invoked(\text{clients, } SOp(s_i)) / (\text{num_clients} *</td>
</tr>
<tr>
<td>i) \text{clients} is the set of all the clients of this service</td>
</tr>
<tr>
<td>ii) \text{num_clients} is the cardinality of the \text{clients} set (total number of clients invoking operations (SOp(s_i)));</td>
</tr>
<tr>
<td>iii) \text{Invoked}(\text{clients, } SOp(s_i)) is the function which computes the total number of all used operations calculated on a per client basis;</td>
</tr>
<tr>
<td>Scale: Absolute; Measurement Unit: Count</td>
</tr>
</tbody>
</table>

The measurement procedure involves the following steps:

1) Construct the set \(\text{inv}\) that consists of invoked service operations for a given \text{client} accessing an interface \(s_i\) of service \(s\), and calculate the cardinality of this set \(|\text{inv}|\).
1.1) enforce the Null Value constraint (as required by Property COHESION.2 shown in the Validation section below) by testing the following condition:

\[
\text{if } (|SOp(si)| > 1 \&\& |inv| == 1) \\
|inv| = |inv| - 1
\]

That is, if the total number of service operations is greater than one (\(|SOp(si)| > 1\)) and a given service client accesses only one service operation (\(|inv| == 1\)), then we need to subtract the value one (1) from the cardinality value of \(inv\). Otherwise, SIUC will never be null since at least one operation will always be used (default usage) by any given client.

2) Sum the \(|inv|\) values for each client (from the set of clients) of the service interface under study, with the resultant value being returned by the function \(Invoked\) (clients, \(SOp(si)\)).

3) Divide the value obtained in step 2 above by (the total number of clients \((\text{num}_\text{clients})\) multiplied by the total number of service operations \(|SOp(si)|\)) in order to derive the final value of SIUC.

The following example illustrates the calculation of the SIUC metric for the Academic Management System (AMS) service shown below:

1) \(|inv\ (\text{Admin})| = |\{\text{opA, opB, opC}\}| = 3; \\
|inv\ (\text{Student})| = |\{\text{opB}\}| = 1; *the condition described in point 1.1 is met, therefore value 1 needs to be deducted i.e. \(|inv\ (\text{Student})| = 0; \\
|inv\ (\text{Academic})| = |\{\text{opB, opC}\}| = 2;
CHAPTER 5. SERVICE-ORIENTED COHESION METRICS

<table>
<thead>
<tr>
<th><strong>VALIDATION</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Property COHESION.1 (Non-negativity and Normalisation) is satisfied since SIUC values for a given service interface will never be negative under any circumstances, and the values of SIUC will always belong to a specified interval [0, 1] as required by the normalisation characteristic.</td>
</tr>
<tr>
<td>Property COHESION.2 (Null Value) is satisfied since SIUC for a given service interface will be null (or zero) when all the clients of the corresponding service use only one operation exposed in the interface.</td>
</tr>
<tr>
<td>Property COHESION.3 (Monotonicity) is satisfied since adding an additional relationship between a service client and a given service operation will not decrease the overall cohesion of this service.</td>
</tr>
<tr>
<td>Property COHESION.4 (Cohesive Modules) is satisfied since the cohesion of a service interface $si_mn$ obtained by joining together two unrelated service interfaces $si_m$ and $si_n$ (insofar they do not have common clients) will not be greater than the maximum cohesion of the original interfaces $si_m$ and $si_n$.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Table 5-11. COH-M2: Service Interface Usage Cohesion (SIUC)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>METRIC NAME</strong></td>
</tr>
</tbody>
</table>
| **OVERVIEW** | The SIIC metric quantifies cohesion of a given service $s$ based on the cohesiveness of the operations $sop \in SOp(si_i)$ exposed in its interface $si_i$, as reflected by all service operations being implemented by the same service implementation elements as captured by the set of collaboration sequences $cs_{sop} \in SOp(si_i) \in CS(si_i)$.

Refer to Chapter 3, Table 3.1–D5.1 for the formal definitions of $cs_{sop} \in SOp(si_i)$ and $CS(si_i)$. |
| **DESCRIPTION** | SIIC is related to the implementation features of service operations, thereby being directly related to the Implementation category of cohesion, as well as indirectly influencing the Coincidental and Conceptual categories. |

As with the previous two metrics, the information required to calculate this metric can be obtained from behavioural design documents such as UML sequence diagrams or business process workflows. |

A service $s$ is deemed to be Implementation cohesive when all of its service operations ($SOp(si_i)$) are implemented by the same implementation
CHAPTER 5. SERVICE-ORIENTED COHESION METRICS

<table>
<thead>
<tr>
<th>FORMAL DEFINITION</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SIIC (s)</strong> =</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MEASUREMENT PROCEDURE</th>
</tr>
</thead>
<tbody>
<tr>
<td>The measurement procedure involves the following steps:</td>
</tr>
<tr>
<td>1) Derive the collaboration sequences <strong>c_sop</strong> (\in<strong>CS(si_s)</strong>) for each service interface operation <strong>sop</strong> (\in<strong>SOp(si_s)</strong>). The collaboration sequences will consist of the sets of implementation elements invoked in response to the invocation of a given service operation with all possible input parameters.</td>
</tr>
<tr>
<td>2) Construct the <strong>IC (s)</strong> set by intersecting all collaboration sequences (obtained in step 1) in a pair-wise manner, and then place all the intersected elements in the <strong>IC (s)</strong> set. Note that duplicate elements will be included in the <strong>IC (s)</strong> set in order to have a theoretically-valid measure which belongs to the specified interval ([0, 1]) as required by the COHESION.1 property (see Validation section below). Formally:</td>
</tr>
<tr>
<td>(\text{IC (s)} = {e \in (C_s ∪ I_s ∪ P_s ∪ H_s ∪ BPS_s) \land \forall \text{c_sop}_A, \text{c_sop}_B \in CS(si_s) } (e \in c_sop_A ∩ c_sop_B)}.</td>
</tr>
<tr>
<td>3) Calculate the cardinality (total number of elements) of the <strong>IC (s)</strong> set constructed in step 2, and then divide the cardinality value by the total number of service implementation elements (</td>
</tr>
<tr>
<td>Note that the value of SIIC will be set to <strong>unknown</strong> in the case a service interface consists of one operation only (</td>
</tr>
<tr>
<td>The following example illustrates the calculation of SIIC measure for the AMS service shown in the corresponding figure.</td>
</tr>
</tbody>
</table>
CHAPTER 5. SERVICE-ORIENTED COHESION METRICS

Table 5-12. COH-M3: Service Interface Implementation Cohesion (SIIC)

<table>
<thead>
<tr>
<th>VALIDATION</th>
<th>Property COHESION.1 (Non-negativity and Normalisation) is satisfied since SIIC values for a given service interface will never be negative under any circumstances, and the values of SIIC will always belong to a specified interval [0, 1] as required by the normalisation characteristic.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property COHESION.2 (Null Value) is satisfied since SIIC for a given service interface will be null (or zero) when the operations exposed in this interface do not have any common implementation elements.</td>
<td></td>
</tr>
<tr>
<td>Property COHESION.3 (Monotonicity) is satisfied since adding a common implementation element to a pair of service operations will not decrease the overall cohesion of this service.</td>
<td></td>
</tr>
<tr>
<td>Property COHESION.4 (Cohesive Modules) is satisfied since the cohesion of a service interface $s_{in}$ obtained by joining together two unrelated service interfaces $s_{im}$ and $s_{in}$ (they do not have any common implementation elements) will not be greater than the maximum cohesion of the original interfaces $s_{im}$ and $s_{in}$.</td>
<td></td>
</tr>
</tbody>
</table>
### OVERVIEW

The *SISC* metric quantifies cohesion of a given service based on the cohesiveness of the operations $sop \in SOp(si)$ exposed in its interface $si$, as reflected by all service operations having sequential dependencies, where a post-condition/output of a given operation satisfies a pre-condition/input of the next operation.

### DESCRIPTION

SISC quantifies the usage patterns of service operations, thereby being directly related to the **Sequential** category of cohesion, as well as indirectly influencing the **Coincidental** and **Conceptual** categories of cohesion.

As with the COH-M2:SIUC metric, the SISC metric is associated with the communication (usage) pattern of service operations. The difference is that in the case of SISC, the dependencies between service operations are also taken into consideration. That is, the communication is deemed to be sequential if, either the output from one operation serves as the input for the next operation, or the post condition of an operation satisfies the precondition of the next operation. More specifically, a service is deemed to be Sequentially cohesive when all of its service operations ($SOp(si)$) are connected via sequential dependencies. The values of SISC will range from 0 to 1, with the value 1 representing the strongest possible Sequential cohesion; and the value 0 representing total lack of Sequential cohesion.

Note that the above definition could be considered as overly-restrictive since it requires the existence of sequential dependencies in between all possible pairs of operations in order to achieve the strongest possible cohesion (the value 1). In practice, it might be sufficient for service operations to have chain-based dependencies in order to be considered as fully sequential. For example, a service interface containing three operations (opA, opB, and opC) could be considered as fully sequential if it exhibits sequential dependencies between [opA and opB], and [opB and opC] (in our definition, the sequential dependency between [opA and opC] is also needed in order for this interface to be considered as fully sequential). This issue should be investigated in future work.

Also note that the sequential usage can be potentially difficult to determine syntactically because:

1. the information related to the pre- and post-conditions of service operations might not be part of the actual design structure. Nevertheless, this information could be obtained from some of the existing behavioural design artefacts. For example, the UML sequence diagrams might utilise the Object Constraint Language (OCL) [178], which is a declarative modelling language included in the UML 2.0 standard that allows defining pre- and post-conditions as part of the standard sequence diagrams. Moreover, the pre- and post-conditions are included in the formal model
of SO system design (Chapter 3, Table 3.1-D2.1).

ii) the information needed to determine the input/output sequential dependencies can be collected only upon examining the implementation-level specifics of a calling element (client), therefore we need to have full access to all calling elements of the service under study.

<table>
<thead>
<tr>
<th>FORMAL DEFINITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>SISC (s) = SeqConnected (SOp(si_s)) / Total (SOp(si_s)), where</td>
</tr>
<tr>
<td>i) SeqConnected (SOp(si_s)) is the function that calculates the number of service operation pairs that have sequential dependencies;</td>
</tr>
<tr>
<td>ii) Total(SOp(si_s)) is the function that returns the number of all possible combinations of operation pairs for the service interface si_s</td>
</tr>
</tbody>
</table>

Scale: Absolute; Measurement Unit: Count

<table>
<thead>
<tr>
<th>MEASUREMENT PROCEDURE</th>
</tr>
</thead>
<tbody>
<tr>
<td>The measurement procedure involves the following steps:</td>
</tr>
<tr>
<td>1) Identify the sequential dependencies (based on the post-conditions/outputs and pre-conditions/inputs) of all service operations SOp(si_s) in a pair-wise manner, and then place the operation pairs that have sequential dependencies into a set of sequentially connected operation pairs SEQOP. The cardinality of the set SEQOP is then calculated and the resultant value returned by the function SeqConnected (SOp(si_s)).</td>
</tr>
<tr>
<td>2) Determine the number of all possible service operation pairs (TP) which will be returned by the function Total (SOp(si_s)). The value of TP can be calculated as TP = ((n-1)*n)/2, where n is the total number of service operations.</td>
</tr>
<tr>
<td>3) Divide the value obtained in step 1 by the number of all possible service operation pairs calculated in step 2 in order to derive the final value of SISC.</td>
</tr>
</tbody>
</table>

Note that the value of SISC will be set to unknown in the case a service interface consists of one operation only (|SOp(si_s)|==1) since SISC is calculated based on a pair-wise comparison of service operations, and as such, it can only be calculated when the service interface has more than one service operation.

The following example illustrates the calculation of the SISC metric for the AMS service interface shown below.
CHAPTER 5. SERVICE-ORIENTED COHESION METRICS

**AMS**

<table>
<thead>
<tr>
<th>int calculateFees (Student student)</th>
</tr>
</thead>
<tbody>
<tr>
<td>String createAcademicTranscript (Student student, String program, int fees)</td>
</tr>
<tr>
<td>String getStaffDetails (String staffID)</td>
</tr>
</tbody>
</table>

Sequential dependencies
- "output value" /
- int calculateFees (Student student)
- "input value" /
- String createAcademicTranscript (... int fees)

1) |SEQOP| = |\{calculateFees, createAcademicTranscript\}| = 1

2) number of service operations \( (n) = 3; \ TP = (n-1)*n/2 = 3 \)

3) **SISC** (**Student Management**) = 1 / 3 = 0.33

**VALIDATION**

Property COUPLING.1 (Non-negativity and Normalisation) is satisfied since SISC values for a given service interface will *never be negative* under any circumstances, and the values of SISC will always *belong to a specified interval [0, 1]* as required by the normalisation characteristic.

Property COUPLING.2 (Null Value) is satisfied since SISC for a given service interface will be *null* (or *zero*) when the operations exposed in this interface have no sequential dependencies.

Property COUPLING.3 (Monotonicity) is satisfied since adding an additional sequential dependency to a pair of service operations will not decrease the cohesion of this service.

Property COUPLING.4 (Cohesive Modules) is satisfied since the cohesion of a service interface \( s_{im} \) obtained by joining together two unrelated (insofar as there are no sequential dependencies) service interfaces \( s_{im} \) and \( s_{in} \) will not be greater than the maximum cohesion of the original interfaces \( s_{im} \) and \( s_{in} \).

Table 5-13. COH-M4: Service Interface Sequential Cohesion (SISC)

<table>
<thead>
<tr>
<th>METRIC NAME</th>
<th>TOTAL INTERFACE COHESION OF A SERVICE (TICS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVERVIEW</td>
<td>The TICS metric quantifies the total (overall) cohesion of a given service based on the cohesiveness of the operations exposed in its interface ( s_{i} ).</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>TICS covers all possible aspects of service interface cohesion as captured by the previously defined metrics COH-M1 - COH-M4. That is, TICS quantifies a cohesion of a service according to the cohesion suppo-</td>
</tr>
</tbody>
</table>
sitions CS1 and CS2 (refer to Section 5.2.1) based on the following characteristics of its service interface:

i) service operations having common parameters and return types (COH-M1: SIDC metric);

ii) service operations being invoked by every client of this service (COH-M2: SIUC metric);

iii) service operations being implemented by the same service implementation elements (COH-M3: SIIC metric);

iv) service operations having sequential dependencies (COH-M4: SISC metric);

To this end, TICS can potentially suggest the best possible cohesiveness of a service (the Conceptual cohesion category), or a total lack of cohesiveness (the Coincidental category). More specifically, the values of TICS will range from 0 to 1, with the value 1 representing the strongest possible cohesion of a service indicating that the service belongs to the Conceptual category of cohesion; and the value 0 representing total lack of cohesion, thereby suggesting that the service could potentially belong to the Coincidental category of cohesion. The values in between 0 and 1 could suggest different levels of Conceptual cohesion.

<table>
<thead>
<tr>
<th>FORMAL DEFINITION</th>
<th>TICS(s) = (SIDC(s) + SIUC(s) + SIIC(s) + SISC(s)) / 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scale: Absolute; Measurement Unit: Count</td>
<td></td>
</tr>
</tbody>
</table>

The measurement procedure consists of calculating the values of the constituent four metrics (SIDC, SIUC, SISC, SIIC) and dividing this number by the total number of constituent metrics (four) in order to derive the value of TICS that belongs to a specified interval [0, 1].

For example, the AMS service, which was used to illustrate the calculation process for the proposed metrics throughout this section, shown below will have the following cohesion values:

\[
\text{SIDC (AMS)} = 0.5 \\
\text{SIUC (AMS)} = 0.56; \\
\text{SISC (AMS)} = 0.33; \\
\text{SIIC (AMS)} = 0.53;
\]

The value of TICS (AMS) will then be equal to: 
\[
(0.5 + 0.56 + 0.33 + 0.53)/4 = 0.48
\]
indicating average total cohesion of the AMS service.
**VALIDATION**

Property COHESION.1 (Non-negativity and Normalisation) is satisfied since TICS values for a given service interface will *never be negative* under any circumstances, and the values of TICS will always *belong to a specified interval [0, 1]* as required by the normalisation characteristic.

Property COHESION.2 (Null Value) is satisfied since TICS for a given service interface will be *null* (or *zero*) when the operations exposed in this interface: i) do not have any common parameter or return types; ii) are not invoked in combination by all the service clients; iii) do not have sequential dependencies; and iv) are not implemented by the same elements.

Property COHESION.3 (Monotonicity) is satisfied since adding an additional cohesion-related construct (for example, a common parameter or return type) to a pair of service operations will not decrease the overall cohesion of this service.

Property COHESION.4 (Cohesive Modules) is satisfied since the cohesion of a service interface $s_{imn}$ obtained by joining together two unrelated service interfaces $s_i$ and $s_n$ will not be greater than the maximum cohesion of the original interfaces $s_i$ and $s_n$.

Table 5-14. COH-M5: Total Interface Cohesion of a Service (TICS)
CHAPTER 5. SERVICE-ORIENTED COHESION METRICS

5.4 Cohesion Assumptions

The following three service-oriented cohesion assumptions propose connections between the structural property of cohesion, as conceptually reflected by the directly and indirectly quantifiable cohesion categories, and the sub-characteristics of maintainability of service-oriented software (defined in Section 2.3). The assumptions are based on the review of the related literature in the areas of software maintainability and software cohesion (Sections 2.3 and 2.4). As with the coupling assumptions presented in Chapter 4, the cohesion assumptions will support the definition of the experimental hypotheses in the empirical evaluation of metrics presented in Chapter 6.

Service-Oriented Cohesion Assumption CA1 (Coincidental cohesion).
The Coincidental category of service cohesion, as reflected by the low value (the value of zero) of the TICS metric (and correspondingly, low values of its constituent metrics SIDC, SIUC, SISC, SIIC), should be avoided since the lack of service cohesion will negatively influence the analysability of a service due to this service performing semantically unrelated operations.

This assumption is consistent with the general understanding of the notion of software cohesion (in the Procedural and OO paradigms) where it has been shown empirically that cohesion of a software module influences its analysability presumably due to the additional effort required to comprehend the software under study [32, 50, 123].

Service-Oriented Cohesion Assumption CA2 (Communicational, External, Sequential, and Implementation cohesion).
The Communicational, External, Sequential, and Implementation categories of service cohesion, as reflected by the high values (the value of one) of the corresponding metrics SIDC, SIUC, SISC, and SIIC are recommended since it is expected that each of the above categories will positively influence service cohesion in a distinct way. This in turn should result in improvements to the analysability sub-characteristic of software maintainability for the reasons described in assumption CA1.

Service-Oriented Cohesion Assumption CA3 (Conceptual cohesion).
The Conceptual category of cohesion, as reflected by the high value (the value of one) of the TICS metric (and correspondingly, high values of its constituent metrics SIDC, SIUC, SISC, SIIC), represents the strongest possible type of cohesion, therefore this category of cohesion is highly recommended since it is expected to result in the improved analysability of a service. Additionally, Conceptual cohesion supports the principle of locality [172] (given that the conceptually related system functionality is located in the same place), which is shown to decrease the maintenance efforts [63].
5.5 Summary

This chapter investigated the concept of software cohesion in service-oriented design, using a *service* as the key design construct for evaluating cohesion based on the *relatedness of the operations exposed in its service interface*. Additionally, this chapter redefined and extended the existing qualitative *categories of cohesion* in order to provide a conceptual foundation for the definition of the quantitative *cohesion metrics*. The purpose of the metrics is twofold. Firstly, the metrics are intended to measure the *quantifiable cohesion categories* and also estimate the *purely semantic* Coincidental and Conceptual categories of cohesion according to the proposed *cohesion suppositions*. Secondly, and more importantly in the context of this research, the metrics are intended to be used as early predictors of the analysability and changeability of service-oriented software products as stated by the presented *cohesion assumptions*.

The metrics were defined in a formal and unambiguous manner using the definitions captured by a model of service-oriented software design presented in Chapter 3. All the derived metrics satisfy the mathematical characteristics of cohesion defined in the property-based software engineering measurement framework of Briand et al. [30] and thus can be deemed as *theoretically-valid measures of cohesion*. Also, the metrics fulfil the desirable pragmatic properties since they are simple to collect and technology independent. Moreover, the metrics were evaluated empirically in terms of i) their ability to accurately reflect the categories of cohesion, and ii) predict the analysability sub-characteristic of maintainability of SO software systems. The results of the empirical evaluation are presented and analysed in Chapter 6.

Finally, note that in future work a number of additional studies could be conducted in order to determine which of the desirable categories of cohesion (Communicational, External, Implementation, and Sequential) have a greater influence on the analysability of software, thereby allowing to establish a complete taxonomy of SO cohesion categories defined on a fully ordinal scale. This in turn, will allow establishing weights for the SIDC, SIUC, SIIC, and SISC metrics based on their perceived influence on service cohesion in order to derive more accurate (weighted) metric of total interface cohesion of a service (TICS).
Chapter 6. Empirical Evaluation of Metrics

This chapter describes the empirical study conducted in this research in order to evaluate the service-oriented coupling and cohesion metrics described in Chapters 4 and 5. The study investigates the influence of the structural design properties of coupling and cohesion, as measured by the derived metrics, on the analysability, changeability, and stability of service-oriented software.

This chapter is organised as follows. Section 6.1 provides an overview of the empirical study, the detailed description of which is presented in Sections 6.2-6.5 following an established template for reporting controlled experiments in software engineering [116, 117], which is separated into three major sections: Experimental planning (Section 6.2), Analysis (which is presented individually for the coupling and cohesion metrics in Sections 6.3-6.4), and Discussion (Section 6.5).

6.1 Overview

While the derived coupling and cohesion metrics have been validated theoretically, theoretical validation alone does not fully substantiate the overall validity of the predictive metrics since it is also necessary to establish empirically the relationship between the metrics and the quality characteristics they purport to predict [34, 98, 213].

In general, the common way to establish such an empirical relationship is to define and statistically test experimental hypotheses that formalise the relationships between the structural properties of software, as measured by the metrics, and the quality characteristic in question measured using already accepted (or standardised) metrics or other quantifiable constructs [75]. More specifically, the experimental hypotheses used in the empirical evaluation of metrics in this chapter are based on the coupling and cohesion assumptions (Sections 4.2 and 5.4), which established explicit links between the different aspects of SO design-level coupling and cohesion and the sub-characteristics of software maintainability.

According to Briand et al. [34], there are two main strategies that can be adopted in order to empirically evaluate software metrics:

Strategy 1: small-scale targeted experiments conducted under research settings.

Strategy 2: large-scale case studies of real-life industrial software projects and products.

Both strategies can have a contrasting impact on the internal and external validity [239] of the produced results. Where i) internal validity refers to the degree to which conclusions can be drawn about the causal effect of independent variables on the dependent variables; and ii) external validity refers to the degree to which the results of the research can be generalised to actual software engineering practice. More specifically, due to its smaller size and con-
controlled nature, Strategy 1 provides greater support for controlling the instrumentation effects [36] that can influence the internal validity of the study22, whilst negatively affecting the external validity given that the experiments are typically small and conducted in the constrained research setting which is not representative of actual software engineering practice. In contrast, Strategy 2 maximises the external validity, being related to real-life industrial studies, but makes it more difficult to control instrumentation effects [126].

The empirical study presented in this chapter employs Strategy 1 since it was suggested that the internal validity of the study should be established prior to establishing the external validity [239], and as such, Strategy 1 was considered suitable for the explorative and controlled experiments conducted as part of this thesis. Additionally, SOC is an emerging paradigm, and as such, it is difficult to conduct large-scale industrial investigation at this stage. Nevertheless, Strategy 2 could be utilised in future work in order to further evaluate the metrics and increase the external validity of the results as discussed in Chapter 7.

The empirical study involved a group of ten participants who were either industry practitioners or post-graduate students undertaking their study in the School of Computer Science and IT, RMIT University. The study adopted a within-subjects [217] experimental design, where all the experimental tasks were attempted by each of the participants. More specifically, the participants were asked to perform a number of maintenance activities on software systems that were developed for this study. The systems exhibited different structural characteristics as reflected by the SO coupling and cohesion software metrics (independent variables). The modification process was documented and measured using the ISO/IEC maintainability metrics (dependent variables). The participants were also asked to provide a subjective cohesion ranking (dependent variable) of the services under study. The correlation between the independent and dependent variables was then evaluated using standard significance testing [140] in order to establish connections between the variables. Moreover, correlation and univariate linear regression analysis was conducted in order to examine the direction and strength of the linear relationships between the variables.

The overall experimental design of the study presented in this chapter uses elements from existing study designs employed by other researchers in order to establish the empirical relationship between software metrics and maintainability using small-scale targeted experiments (described in Section 2.5.3). Note that although all effort was made to eliminate the threats to validity present in the existing studies, there are still a number of threats to the validity of the results obtained in this empirical study. For example, the size of the software systems was small, thereby limiting our ability to generalise the obtained results. The threats to validity are

---

22 For example, it is easier to manipulate the structural properties of interest, while keeping the other structural properties (controlled variables) as constant as possible, when conducting small-scale targeted experiments under research settings.
discussed throughout this chapter as appropriate, and summarised in Section 6.5.1 following the threats to validity classification of Wohlin [239].

Finally, the presentation of the empirical study follows a template for reporting controlled experiments in software engineering proposed by Jedlitschka et al. [116, 117] (Section 2.5.3) which is designed to support a systematic and well-structured description of the experimental studies. The template consists of four major sections (Experiment planning (or experimental protocol), Execution\(^{23}\), Analysis of the results, and Discussion) and associated sub-sections that serve as the foundation of the structure of the rest of this chapter.

## 6.2 Experimental Protocol

This section presents the overall protocol (or plan) that was used when performing the experimental tasks and analysing the obtained results. More specifically, the following aspects of the empirical study are described: i) goals of the study; ii) the participants involved in the study; iii) the experimental hypotheses and variables; iv) the experimental material and associated tasks; v) the overall experimental design; vi) the execution procedure; and vii) the data collection and analysis procedures. Note that an effort has been made to provide a detailed and comprehensive description of the experimental protocol so that the study can be replicated on a larger group of participants and a range of different service-oriented systems in future work.

### 6.2.1 Goals

The main goal of the study is to empirically evaluate the results of this thesis. More specifically, this study was conducted in order to evaluate the capability of the derived coupling and cohesion metrics to predict the specific sub-characteristics of maintainability (namely, analysability, changeability, and stability) according to the coupling and cohesion assumptions described in Chapters 4 and 5. The actual experimental goals, which were defined following the “Goal/Question/Metric” (GQM) [19] template described in Section 1.3.2, are presented below.

---

**GOAL-COUP1: COUPLING METRICS - MAINTAINABILITY PREDICTION**

**ANALYSE** the derived service-oriented coupling metrics, which are designed to measure the: *intra-service* relationships between implementation elements of a service; *indirect extra-service* relationships between services via service interfaces; and *direct extra-service* rela-

---

\(^{23}\) The Execution section was combined with the Experimental planning section to improve the readability of this chapter.
tionships between elements belonging to different services,

**FOR THE PURPOSE OF** evaluating their predictive capability,

**WITH RESPECT TO** early estimation of the specific sub-characteristics of software maintainability, namely *analysability, stability, and changeability*,

**FROM THE POINT OF VIEW OF** software engineers,

**IN THE CONTEXT OF** an experimental SO software system that contains software services, elements of which exhibit different levels of coupling.

---

**Table 6-1. Experimental Goal (GOAL-COUP1) – Coupling metrics**

**GOAL-COUP2: COUPLING METRICS - MAINTAINABILITY PREDICTION – STRENGTH OF COUPLING RELATIONSHIPS**

**ANALYSE** the derived service-oriented *coupling* metrics, designed to measure different types of coupling relationships (described in GOAL-COUP1),

**FOR THE PURPOSE OF** evaluating their relative strength (or impact), thereby:

i) Establishing relative weights for each coupling relationship type and corresponding metric based on their influence on maintainability;

ii) Comparing with the widely-accepted OO metric, Coupling Between Objects (CBO) [44]. Such a comparison is possible because CBO is structurally equivalent to the WISCE metric (refer to Section 4.3) intended to measure *intra-service* relationships,

**WITH RESPECT TO** early estimation of the specific sub-characteristics of software maintainability, namely *analysability, stability, and changeability*,

**FROM THE POINT OF VIEW OF** software engineers,

**IN THE CONTEXT OF** an experimental SO software system that contains software services, elements of which exhibit different types and levels of coupling.

---

**Table 6-2. Experimental Goal (GOAL-COUP2) – Coupling metrics**

**GOAL-COH1: COHESION METRICS – MAINTAINABILITY PREDICTION**

**ANALYSE** the derived service-oriented *cohesion* metrics, designed to quantify the semantic categories of service cohesion (refer to Section 5.2),

**FOR THE PURPOSE OF** evaluating their predictive capability,

**WITH RESPECT TO** early estimation of the *analysability* of service-oriented software,
FROM THE POINT OF VIEW OF software engineers,
IN THE CONTEXT OF an experimental SO software system that contains software services which exhibit qualitatively different levels of cohesion

<table>
<thead>
<tr>
<th>Table 6-3. Experimental Goal (GOAL-COH1) – Cohesion metrics</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>GOAL-COH2: COHESION METRICS – INDICATORS OF SERVICE COHESION</th>
</tr>
</thead>
</table>

ANALYSE the derived service-oriented cohesion metrics, designed to quantify the semantic categories of service cohesion (refer to Section 5.2),

FOR THE PURPOSE OF evaluating their predictive capability,

WITH RESPECT TO automated allocation of software services to the inherently semantic categories of service cohesion (Coincidental and Conceptual),

FROM THE POINT OF VIEW OF software engineers,

IN THE CONTEXT OF an experimental SO software system that contains software services which exhibit qualitatively different levels of cohesion

<table>
<thead>
<tr>
<th>Table 6-4. Experimental Goal (GOAL-COH2) – Cohesion metrics</th>
</tr>
</thead>
</table>

6.2.2 Participants

The experimental participants (subjects) were either industry practitioners or post-graduate students undertaking their study in the School of Computer Science and IT, RMIT University. Purposive sampling [217] was employed to select the participants. Purposive sampling is a participant selection technique frequently used in the behavioural sciences, where the participants are selectively chosen based on expert judgement according to pre-defined criteria [217]. For example, in this study an effort was made to select participants with comparable: i) practical experience with software development in the industrial, research, or academic settings; ii) knowledge of the various development paradigms, including SOC; and iii) knowledge and experience with developing (and maintaining) OO software products using the Java programming language, which was used to implement the experimental systems.

In total, 10 subjects participated in the study (5 industry practitioners and 5 post-graduate students). The level of experience and knowledge of each participant was then evaluated prior to the study using a short User Profile questionnaire shown in Appendix D. In addition, all participants completed a small pre-test task, the description of which can also be found in Appendix D. The purpose of the task was to provide a basis for comparing the programming/design skills of the participants and evaluate their knowledge of the technologies used in the study, namely UML and the Java programming language.
CHAPTER 6. EMPIRICAL EVALUATION OF METRICS

Table 6-5. Answers to the User Profile Questionnaire and results of the Pre-test task

Table 6-5 shows the summary of the responses to the User Profile questionnaire and the results of the pre-test task. The relatively small data dispersion, as indicated by low standard deviation (Std. Dev.) values, suggests that the participants had similar knowledge and skills in the required areas. Note that all participants:

i) were male in the 18-29 age group;

ii) had completed (or were in the process of completing) a postgraduate university degree in an IT related discipline;

<table>
<thead>
<tr>
<th>EXPERIENCE RELATED TO THE DEVELOPMENT PARADIGMS, LANGUAGES, AND TECHNOLOGIES USED IN THE EMPIRICAL STUDY (1 NEVER USED – 4 EXPERT)</th>
<th>MIN</th>
<th>MAX</th>
<th>MEAN</th>
<th>MEDIAN</th>
<th>STD. DEV.</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBJECT-ORIENTED DEVELOPMENT</td>
<td>3</td>
<td>4</td>
<td>3.4</td>
<td>3.5</td>
<td>0.53</td>
</tr>
<tr>
<td>SERVICE-ORIENTED COMPUTING</td>
<td>2</td>
<td>3</td>
<td>2.4</td>
<td>2</td>
<td>0.52</td>
</tr>
<tr>
<td>UML 1.1 OR 2.0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>WEB SERVICE RELATED TECHNOLOGIES</td>
<td>2</td>
<td>3</td>
<td>2.3</td>
<td>2</td>
<td>0.48</td>
</tr>
<tr>
<td>JAVA (J2SE OR/AND J2EE)</td>
<td>3</td>
<td>4</td>
<td>3.3</td>
<td>3</td>
<td>0.48</td>
</tr>
<tr>
<td>SOFTWARE MAINTENANCE</td>
<td>2</td>
<td>4</td>
<td>2.9</td>
<td>3</td>
<td>0.88</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>UNDERSTANDING OF THE KEY CONCEPTS RELATED TO THE EMPIRICAL STUDY (1 LOW – 5 HIGH)</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>UNDERSTANDING OF THE PRINCIPLES OF SOC</td>
<td>3</td>
<td>4</td>
<td>3.1</td>
<td>3</td>
<td>0.32</td>
</tr>
<tr>
<td>UNDERSTANDING OF THE MAINTAINABILITY QUALITY CHARACTERISTIC</td>
<td>3</td>
<td>5</td>
<td>3.8</td>
<td>4</td>
<td>0.63</td>
</tr>
<tr>
<td>UNDERSTANDING OF THE COUPLING AND COHESION PROPERTIES OF SOFTWARE</td>
<td>3</td>
<td>5</td>
<td>3.9</td>
<td>4</td>
<td>0.57</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PRETEST TASK</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TOTAL TIME SPENT ON TASK (MINS)</td>
<td>15</td>
<td>22</td>
<td>18.4</td>
<td>19</td>
<td>2.32</td>
</tr>
<tr>
<td>DEGREE OF TASK DIFFICULTY (1 EASY – 5 HARD)</td>
<td>1</td>
<td>2</td>
<td>1.2</td>
<td>1</td>
<td>0.42</td>
</tr>
<tr>
<td>QUALITY OF THE SOLUTION (1 LOW – 5 HIGH)</td>
<td>4</td>
<td>5</td>
<td>4.7</td>
<td>5</td>
<td>0.48</td>
</tr>
</tbody>
</table>

(February, 2009)
iii) possessed OO development experience ranging from six months to five years;
iv) had some basic experience and knowledge in developing service-oriented solutions.

Finally, all participants were unpaid volunteers who had personal and/or professional interest in Service-Oriented Computing.

6.2.3 Dependent Variables

This section presents the dependent variables used in the statistical analysis of the experimental hypotheses defined in Section 6.2.4.

6.2.3.1 Maintainability Metrics

The maintainability metrics are the subset of the ISO/IEC TR 9126-2:3:2003 [112, 113] maintainability metrics covering the analysability, changeability, and stability sub-characteristics of maintainability investigated in this research (refer to the research goals defined in Section 6.2.1). Note that all ISO/IEC metrics are defined on ratio or absolute scales, thereby being suitable for the parametric statistical analyses used in this study (Section 6.2.10).

- **Analysability**

  Failure Analysis Efficiency (FAE) = \( \frac{\text{Sum}(T)}{N} \), where
  \( T \) = time taken to analyse each cause of failure (or time taken to locate a software fault); \( N \) = number of failures of which causes are found. *The interpretation of possible values: 0 < FAE; the closer to 0, the better.*

- **Changeability**

  Modification Complexity (MC) = \( \frac{\text{Sum}(T)}{N} \), where
  \( T \) = work time spent on each change; and \( N \) = number of changes. Note that this metric is equivalent to a widely-referenced changeability metric Mean Time To Repair (MTTR) [75]. *The interpretation of possible values: 0 < MC; the closer to 0, the better.*

- **Stability**

  Modification Impact Localisation (MIL) = \( \frac{A}{B} \), where
  \( A \) = number of emerged adverse impacts (failures) in the system after modifications; and \( B \) = number of modifications made. *The interpretation of possible values: 0 <= MIL; the closer to 0, the better.*

---

24 A *failure* in the ISO/IEC TR 9126-2:3:2003 standard is defined as the termination of the ability of a product to perform a required function; whereas a *fault* refers to an incorrect step, process or data definition in a computer program which results in a failure.
6.2.3.2 Subjective Cohesion Ranks

To allow evaluating GOAL-COH2 (Section 6.2.1), the participants were asked to subjectively rank the cohesion levels of the services under study using a five-level Likert scale [56]. The available rank levels (1 to 5, with the value 1 indicating low cohesion, and the value 5 indicating high cohesion) were expected to correlate to five different levels of service cohesion exhibited by five experimental software services used in the study as described in Section 6.2.5.2. To this end, the Cohesion Rank (CR) measure was established to represent the cohesion ranks using integer values ranging from 1 (low cohesion) to 5 (high cohesion). CR is defined on an interval scale, thus being suitable for the parametric tests (such as one-way ANOVA) used in this research. The specific tests are described in detail in Section 6.2.10.

6.2.4 Hypotheses and Independent Variables

The service-oriented coupling (CSA1-CSA3, Section 4.2) and cohesion (CA1-CA3, Section 5.4) assumptions that established links between the proposed metrics and the sub-characteristics of maintainability, and cohesion suppositions (CS1 and CS2, Section 5.2) that related cohesion metrics to the semantic categories of cohesion, provide direct support for the unambiguous definition of the experimental hypotheses presented in this section. More specifically, the experimental hypotheses are explicitly mapped to the goals of this study (Section 6.2.1) and also utilise the proposed coupling and cohesion metrics (as independent variables), and the ISO/IEC TR 9126-2:3:2003 metrics and CR measure presented in the previous section (as dependent variables) in their definitions. This allows establishing formal links between the SO metrics and the sub-characteristics of maintainability according to the associated assumptions.

6.2.4.1 Scope of the Study (Independent Variables)

Note that not all of the coupling assumptions, and associated metrics, were empirically evaluated in this chapter. This is because this empirical study is considered to be an exploratory investigation rather than a large-scale industrial study, and as such, the decision was made to only evaluate the coupling assumptions and corresponding metrics that can be effectively collected from the small service-oriented systems developed for this study. Specifically, the following coupling assumptions and metrics were omitted from the study:

i) Common coupling assumptions CCA1 and CCA2 (Section 4.2) are not evaluated since they are related to the general notion of coupling which has been empirically shown to influence the maintainability of software in previous paradigms [7, 85, 151]. More specifically, the COUP-M11:Response for Operation (RFO) and COUP-AM5:Response for Service (RFS) metrics (Sections 4.3.2 and 4.3.3) were not evaluated since these metrics are conceptually and...
structurally similar to the existing OO metric, Response for Class (RFC) \[43, 44\], which was shown to be directly related to the maintainability of software in previous studies \[32, 45\].

ii) Service-oriented coupling assumptions CSA4 and CSA5 (Section 4.2) related to the system partitioning and purity characteristics are not evaluated due to the difficulties related to collection of the associated metrics COUP-M9:System Partitioning Factor (SPARF) and COUP-M10:System Purity Factor (SPURF) (defined in Section 4.3.2). The SPARF and SPURF metrics are system-level measures and developing multiple full system designs, as required to evaluate these metrics, was considered outside of the scope of this research. This can be considered as one of the limitations of this study as described further in Section 6.5.1.

iii) The service-level coupling metrics COUP-M7:Number of Coupled Incoming Services (NCIS) and COUP-M8:Number of Coupled Outgoing Services (NCOS) described in Section 4.3.2, and aggregate metrics COUP-AM1 – COUP-AM4 described in Section 4.3.3 are not evaluated since the evaluation of their constituent element-level metrics should be sufficient to indirectly establish the empirical validity of the aggregate metrics.

6.2.4.2 Coupling Hypotheses

The following coupling hypotheses, which are defined in terms of the associated independent and dependent variables, are based on the Service-Oriented Coupling Assumptions CSA1-CSA3 (Section 4.2) related to GOAL-COUP1 and GOAL-COUP2. Note that in order to allow testing whether the specific types of coupling relationships have varying impact on the maintainability sub-characteristics (GOAL-COUP2), the intra-service coupling metric (Weighted Intra-Service Coupling between Elements (WISCE)) was separated into incoming and outgoing WISCE types.

- **GOAL-COUP1: Coupling metrics - Maintainability Prediction**

**Hypothesis H\textsubscript{coup1}**- A highly-coupled service-oriented design element will have a significantly negative impact on the maintainability sub-characteristics compared to a lowly-coupled element.

This hypothesis is subdivided into a number of more concrete hypotheses to be used in this study as follows:

**Hypothesis H\textsubscript{coup1.1}**- An element with a high\textsuperscript{25} value of incoming Weighted Intra-Service Coupling between Elements (WISCE) will result in significantly lower (or worse) service change-ability (as reflected by a higher value of the Modification Complexity (MC) metric); and stability (as reflected by a higher value of the Modification Impact Localisation (MIL) metric); compared to an element with a low value of incoming WISCE.

\textsuperscript{25} The actual values used to represent high (and low) coupling in all coupling hypotheses are described in Section 6.2.5.1.
Hypothesis $H_{\text{coup1.2}}$ - An element with a high\textsuperscript{26} value of outgoing Weighted Intra-Service Coupling between Elements (WISCE) will exhibit significantly lower analysability (as reflected by a higher value of the Failure Analysis Efficiency (FAE) metric) and changeability (as reflected by a higher value of the Modification Complexity (MC) metric) compared to an element with a low value of outgoing WISCE.

- Assumption: CSA1

Hypothesis $H_{\text{coup1.3}}$ - A service interface with a high value of Extra-Service Incoming Coupling of Service Interface (ESICSI) will result in significantly lower system changeability (a higher value of MC) and stability (a higher value of MIL) compared to a service interface with a low value of ESICSI.

- Assumption: CSA2.1

Hypothesis $H_{\text{coup1.4}}$ - An element with a high value of Element to Extra Service Interface Outgoing Coupling (EESIOC) will exhibit significantly lower analysability (a higher value of FAE) and changeability (higher value of MC) compared to an element with a low value of EESIOC.

- Assumption: CSA2.2

Hypothesis $H_{\text{coup1.5}}$ - An element with a high value of Weighted Extra-Service Incoming Coupling of an Element (WESICE) will result in significantly lower system changeability (a higher value of MC) and stability (a higher value of MIL) compared to an element with a low value of WESICE.

- Assumption: CSA3.1

Hypothesis $H_{\text{coup1.6}}$ - An element with a high value of Weighted Extra-Service Outgoing Coupling of an Element (WESOCE) will exhibit significantly lower analysability (a higher value of FAE) and changeability (a higher value of MC) compared to an element with a low value of WESOCE.

- Assumption: CSA3.2

Note that the above six hypotheses ($H_{\text{coup1.1}} - H_{\text{coup1.6}}$) are defined in terms of multiple dependent variables. That is, they are designed to investigate more than one sub-characteristic of maintainability (e.g. changeability and stability) according to the corresponding coupling assumptions. These hypotheses are tested individually for each stated dependent variable (in Section 6.3.2.1), but were described here as a combined hypotheses in order to minimise the total number of presented hypotheses and thus improve the readability of this section. This is also the case with the following coupling hypotheses, where the dependent variables are again combined together to minimise the total number of hypotheses.

\textsuperscript{26} The actual values used to represent high (and low) coupling in all coupling hypotheses are described in Section 6.2.5.1.
• **GOAL-COUP2**: Coupling metrics - Maintainability Prediction (the strength of coupling relationships)

**Hypothesis** $H_{coup2}$ - Different coupling relationship types have varying impact on the maintainability sub-characteristics.

This hypothesis is subdivided into a number of more concrete hypotheses as follows:

**Hypothesis** $H_{coup2.1}$ - An element with a high number of incoming intra-service relationships (a high value of *Weighted Intra-Service Coupling between Elements (WISCE)*) will result in significantly higher (or better) system *changeability* (a lower value of *Modification Complexity (MC)*) and *stability* (a lower value of *Modification Impact Localisation (MIL)*) compared to an element with the same number of indirect extra-service incoming relationships (a high value of *Extra-Service Incoming Coupling of Service Interface (ESICSI)*).

- Assumption: CSA2.1

**Hypothesis** $H_{coup2.2}$ - An element with a high number of outgoing intra-service relationships (a high value of *WISCE*), will exhibit significantly higher *analysability* (a lower value of *Failure Analysis Efficiency (FAE)*) and *changeability* (a lower value of *MC*) compared to an element with the same number of indirect extra-service outgoing relationships (a high value of *Element to Extra Service Interface Outgoing Coupling (EESIOC)*).

- Assumption CSA2.2

**Hypothesis** $H_{coup2.3}$ - An element with a high number of intra-service incoming relationships (a high value of *WISCE*), will result in significantly higher system *changeability* (a lower value of *MC*) and *stability* (a lower value of *MIL*) compared to an element with the same number of direct extra-service incoming relationships (a high value of *Weighted Extra-Service Incoming Coupling of an Element (WESICE)*).

- Assumption CSA3.1

**Hypothesis** $H_{coup2.4}$ - An element with a high number of outgoing intra-service relationships (a high value of *WISCE*), will exhibit significantly higher *analysability* (a lower value of *FAE*) and *changeability* (a lower value of *MI*) compared to an element with the same number of direct extra-service outgoing relationships (a high value of *Weighted Extra-Service Outgoing Coupling of an Element (WESOCE)*).

- Assumption CSA3.2

**Hypothesis** $H_{coup2.5}$ - An element with a high number of indirect extra-service incoming relationships (a high value of *ESICSI*), will result in significantly higher system *changeability* (a lower value of *MI*) and *stability* (a lower value of *MIL*) compared to an element with the same number of direct extra-service incoming relationships (a high value of *WESICE*).

- Assumption CSA3.1
Hypothesis $H_{coup2.6}$ - An element with a high number of indirect extra-service outgoing relationships (a high value of $EE$), will exhibit significantly higher *analysability* (a lower value of $FAE$) and *changeability* (a lower value of $MI$) compared to an element with the same number of direct extra-service outgoing relationships (a high value of $WESOC$).

- Assumption CSA3.2

6.2.4.3 Cohesion Hypotheses

The following cohesion hypotheses are based on: i) Service-Oriented Cohesion Assumptions CA1-CA3 (Section 5.4) which are related to GOAL-COH1; and ii) Cohesion Suppositions CS1 and CS2 (Section 5.2) which are related to GOAL-COH2.

- **GOAL-COH1**: *Cohesion metrics – Maintainability Prediction*

**Hypothesis $H_{coh1}$** - A service with a high value (the closer to the value 1) of the *Total Interface Cohesion of a Service (TICS)* metric, as reflected by the constituent metrics of TICS [$Service Interface Data Cohesion (SIDC)$, $Service Interface Usage Cohesion (SIUC)$, $Service Interface Sequential Cohesion (SISC)$, and $Service Interface Implementation Cohesion (SIIC)$], will exhibit significantly higher *analysability* (low Failure Analysis Efficiency (FAE)) compared to a service with a low value of TICS.

- Assumption CA1–CA3

- **GOAL-COH2**: *Cohesion metrics – Indicators of Service Cohesion*

**Hypothesis $H_{coh2}$** - A service with a high value (the closer to the value 1) of the *Total Interface Cohesion of a Service (TICS)* metric, as reflected by the constituent metrics of TICS [$Service Interface Data Cohesion (SIDC)$, $Service Interface Usage Cohesion (SIUC)$, $Service Interface Sequential Cohesion (SISC)$, and $Service Interface Implementation Cohesion (SIIC)$], will have a significantly higher *level of conceptual cohesion* (high Cohesion Rank (CR) values) compared to a service with a low value of TICS.

- Supposition CS1 and CS2

Note that all coupling and cohesion hypotheses were stated as *alternative hypotheses*. That is, they describe the expected influence of the structural property in question on the specific sub-characteristics of maintainability. In contrast, the corresponding *null hypotheses* would be defined in terms of the absence of examined effects.

6.2.5 Experimental Material

The experimental systems were based on an existing prototypical service-oriented Academic Management System (AMS), which was loosely modelled on the business rules and procedures common to RMIT University (although some of the rules were simplified or changed to
support the experimental scenarios). The original system was intended to provide a foundation for the practical assessments used in the E-Commerce and Enterprise Systems and Mobile Application Development (J2ME) courses run by the School of the Computer Science and IT, RMIT University. Note that the chosen application domain (educational organisation) has the advantage of being easily comprehensible by the participants, thereby ensuring that system requirements could be easily interpreted.

Selected parts of the original system have been modified and redesigned in order to support the evaluation of the hypotheses investigated in this study. More specifically, the experimental material consisted of two controlled service-oriented software systems developed from the same AMS-related requirements document (shown in Appendix E, Section 1). The first system, SYS-COUP (described in Section 6.2.5.1), was designed in a way that allows testing the coupling hypotheses $H_{coup1.1} - H_{coup 1.6}$ and $H_{coup2.1} - H_{coup 2.6}$. The second system, SYS_COH (described in Section 6.2.5.2), was designed to test the cohesion hypotheses $H_{coh1}$ and $H_{coh2}$. Both systems were constructed as a collection of services exposed via local and WSDL-based interfaces. The services themselves were implemented using the Java programming language (Java EE 5).

Appendix E, Sections 2 and 3 provide UML design diagrams that demonstrate the structure of experimental systems. Note that both experimental systems (SYS-COH and SYS-COUP) included a number of utility services that provided basic support for data access and manipulation; and communication with external partners (prototype web services) running on a different application server. Such utility services were required in order to remove any potential influence of the specific technologies used to develop the experimental systems (for example EJB3 and Java Persistence API (JPA)) on the maintenance efforts. Additionally, SYS-COUP included a number of support services that provided means for evaluating the extra-service coupling metrics. The participants were notified that all utility and support services were fault free.

The following sub-sections describe the specifics of both experimental systems. Note that given our aim to conduct a fully controlled experiment, as was discussed in Section 6.1, all effort was made to only manipulate the structural properties investigated in the particular experimental system, while keeping the other structural properties (or controlled variables) as constant as possible in order to prevent their influence on the experimental results. The controlled variables are described individually for both systems.

6.2.5.1 Coupling System (SYS-COUP)
The modified version of AMS designed to investigate the coupling-related hypotheses was implemented as a collection of cooperating services, each located in a dedicated package as
CHAPTER 6. EMPIRICAL EVALUATION OF METRICS

shown in Appendix E, Section 2. In total three services were required in order to support the controlled investigation of different types of coupling as follows:

- service SER-COUP1 (ams.services.academic-management service shown in Appendix E, Section 2) was designed to support the investigation of *intra-service* coupling (hypotheses $H_{coup1.1}, H_{coup1.2},$ and $H_{coup2.1} - H_{coup2.4}$);

- service SER-COUP2 (ams.services.student-management service shown in Appendix E, Section 2) was designed to support the investigation of *indirect extra-service* coupling (hypotheses $H_{coup1.3}, H_{coup1.4}, H_{coup2.1}, H_{coup2.2}, H_{coup2.5},$ and $H_{coup2.6}$);

- service SER-COUP3 (ams.services.enrollment-support service shown in Appendix E, Section 2) was designed to support the investigation of *direct extra-service* coupling (hypotheses $H_{coup1.5}, H_{coup1.6},$ and $H_{coup2.3} - H_{coup2.6}$).

Moreover, the functionality provided by the experimental services was further (logically) divided into two sub-services in order to support the investigation of the incoming (SER-COUP1a, SER-COUP2a, and SER-COUP3a) and *outgoing* (SER-COUP1b, SER-COUP2b, and SER-COUP3b) coupling relationships in isolation from one another.

The following characteristics are common to all three services:

- Each service aims to evaluate the coupling-related hypotheses using four designated implementation elements/or service interfaces: two *lowly* [incoming and outgoing]-coupled elements/or interfaces, and two *highly* [incoming or outgoing]-coupled elements/or interfaces, where i) an implementation element/or service interface is considered to be *lowly-coupled* if it is coupled to/from one implementation element or service interface only; and ii) an implementation element/or service interface is considered to be *highly-coupled* if it is coupled to/from five other implementation elements or service interfaces. The number 5 was chosen because it allowed developing sub-systems of manageable size, and also because it represents the maximum number of couples for a given implementation element in the original AMS.

- Each element designed to investigate low and high *outgoing coupling* contains one introduced software fault. This allows evaluating the analysability of elements under study. Note that all faults were considered as conceptually similar since they were related to the same problem domain. Also, the algorithmic and structural complexity of all faults was approximately equivalent as was confirmed in the pilot study (described in Section 6.2.8.1).

The following *controlled variables* (or factors) were kept as constant as possible in order to prevent their influence on the experimental results:

---

27 Note that to assist in easier presentation of the results in Section 6.3, the three experimental coupling services (SER-COUP1 - SER-COUP3) will be described in terms of six sub-services (SER-COUP1a, SER-COUP1b, SER-COUP2a, SER-COUP2b, SER-COUP3a, and SER-COUP3b) in the remainder of this chapter.
i) **Interface Size and Complexity**: all three experimental services expose equal number (4) of service operations in order to ensure their comparability in terms of an interface size and complexity. The number 4 was chosen because it allowed developing services of manageable and (comparable) size, whilst maximising the amount of experimental data. Additionally, it represents the mean number of service operations exposed in the interfaces of services included in the original AMS.

ii) **Implementation Size**: the lines of code (LOC) measures for each of the experimental services were similar (300-350 LOC per service). Note that the total size of the SYS-COUP system, including the supporting services (as shown in Appendix E, Section 3.1), was approximately 3000 LOC;

iii) **Cohesion**: the cohesion of experimental services, as measured by the Total Interface Cohesion of a Service (TICS) metric, was kept at similar average (≈0.5) level, with the mean values of TICS ranging from 0.47 to 0.52.

### 6.2.5.2 Cohesion System (SYS-COH)

The modified version of AMS intended to support the investigation of the cohesion-related hypotheses was composed of five distinct services, each located in a dedicated package as shown in Appendix E, Section 3. The cohesion of the services was manipulated, independently of the other structural properties, across five specifically chosen levels of the Total Interface Cohesion of a Service (TICS) metric. More specifically, the TICS values for the experimental services ranged from the value 0 (total lack of cohesion) to the value 1 (best possible cohesion) in the 0.25 increments as shown in Table 6-6. This was done in order to allow establishing the regression model (Section 6.4.3) for evaluating the projected linear correlation between the different values of cohesion, as measured by TICS, and the measures of analysability (FAE) and subjective cohesion ranking (CR) according to hypotheses $H_{coh1}$ and $H_{coh2}$. Note that the decision was made to only examine the overall cohesion of services (using TICS), without investigating the individual aspects of service cohesion reflected by the constituent metrics of TICS (SIDC, SIUC, SISC, and SIIC) because:

i) TICS indicates the overall (total) cohesion of a service, and as such, it can be considered as the most important and complete measure of cohesion;

ii) Evaluating TICS allows indirect evaluation of its four constituent metrics since there is a direct dependency between the values of TICS and the values of the constituent metrics;

iii) It would be impractical to manipulate the values of the constituent metrics of TICS independently from one another. For example, manipulating a value of any given constituent metric while keeping the values of other constituent metrics at the constant level of 0 will produce services with low overall cohesion, thereby limiting our ability to establish any statistically significant correlation between the cohesion of a service and its analysability.
CHAPTER 6. EMPIRICAL EVALUATION OF METRICS

<table>
<thead>
<tr>
<th>SERVICE NAME</th>
<th>COMMUNICATIONAL METRIC: SIDC</th>
<th>EXTERNAL SIUC</th>
<th>SEQUENTIAL SISC</th>
<th>IMPLEMENTATION SIIC</th>
<th>TOTAL COHESION TICS</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMS1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>AMS2</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
</tr>
<tr>
<td>AMS3</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>AMS4</td>
<td>0.75</td>
<td>0.75</td>
<td>0.75</td>
<td>0.75</td>
<td>0.75</td>
</tr>
<tr>
<td>AMS5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6-6. Experimental System (SYS-COH) overview – cohesion values per service

The following characteristics are common to all (five) services of SYS-COH:
- Each service contains 4 artificially introduced software faults (one fault per service operation) in order to support the evaluation of the analysability of services under study. As with the coupling system, all faults were considered as conceptually equivalent since they were related to the same problem domain.
- All individual implementation elements (OO classes) are highly cohesive as indicated by high values of OO cohesion metric, Lack of Cohesion in Objects (LCOM) [44].

The following controlled variables were kept as constant as possible in order to prevent their influence on the experimental results:

i) Interface Size and Complexity: all five service interfaces expose equal number (4) of service operations in order to ensure their comparability in terms of the interface size and complexity. The number 4 was chosen because it represents the minimum possible number that can be used to generate the required variations of TICS.

ii) Implementation Size: the lines of code (LOC) measures for each of the experimental services were similar (170-200 LOC per service). Note that the total size of the SYS-COH system, including the supporting services, was approximately 2300 LOC.

iii) Coupling: the coupling of services was kept at constant levels as follows:
- All implementation elements in the system belong to at least one service, resulting in a best possible value (the value 1) of the System Partitioning Factor (SPARF) metric;
- Each implementation element in the system belongs to one and only one service resulting in the best possible value (the value 1) of the System Purity Factor (SPURF) metric;
- The values of the Response for Operation (RFO) metric (described in Section 4.4) were kept at a constant number (2) for all operations exposed in the interfaces of the experimental services, thereby ensuring that the same number of elements needs to be analysed when determining the cause of failures (described in Section 6.2.6.2).

Also note that the experimental services were named uniformly, following the AMSx format as shown in Table 6-6. This was done in order to remove any potential influence of a particular service name on the cohesion ranking task described in Section 6.2.6.2. More spe-
cifically, it was decided that naming the experimental services according to the provided functionality could influence the participants’ perception of service cohesion, and thus compromise the internal validity of the experimental results.

6.2.6 Experimental Tasks

The experimental tasks performed in this study can be classified into corrective and perfective maintenance activities (refer to Section 2.3). The tasks were intended to support the evaluation of the analysability, changeability, and stability sub-characteristics of maintainability in order to test the experimental hypotheses of this study. The tasks, which were documented on three printed pages and distributed to the participants at the beginning of the experiments, are summarised below individually for coupling and cohesion-related experimental systems. The complete description of all the tasks can be found in Appendix F.

Note that an approximate duration of each experimental task was established based on the judgment of the author. This was done in order to estimate the total time required to complete all the tasks (the participants were asked to perform all the tasks within a fixed time-period as described further in Section 6.2.7). The approximated task duration times were confirmed in the pilot study described in Section 6.2.8.1.

6.2.6.1 Coupling (SYS-COUP) Related Tasks

TASK-COUP1: The participants were asked to determine the causes of six failures related to the core functionality of the system (one failure per outgoing lowly- and highly-coupled element under study, refer to Section 6.2.5.1). This task can be considered as a corrective maintenance activity, being designed to evaluate the analysability of the implementation elements of SYS-COUP using the Failure Analysis Efficiency (FAE) metric shown in Section 6.2.3.1. As described previously, an effort was made to ensure that all causes of failures (or software faults) were comparable in terms of their conceptual complexity. Also, it was expected that it will take approximately 6-8 minutes to analyse each failure. Note that participants were required only to identify the faults, by noting the name of the faulty implementation element/s and operation/s, but not fix them. Approximate task duration: 40 minutes.

TASK-COUP2: The participants were asked to implement 12 changes to the business rules and logic - one change per designated [incoming/outgoing] lowly- and highly-coupled element in three experimental services of SYS-COUP. This task can be considered as perfective maintenance, and was intended to support the investigation of the changeability and stability sub-characteristics. It was expected that it will take approximately 8-10 minutes per required change. Note that the changes were related to the different functional requirements of AMS and internal implementation characteristics of each service were different, and as such, the
knowledge of the internals of any particular experimental service was unlikely to be directly transferred to the other sub-systems, thereby minimising any potential learning effects [36]. Also, an effort was made to ensure that all changes are comparable in terms of their conceptual complexity. Approximate task duration: 110 minutes.

6.2.6.2 Cohesion (SYS-COH) Related Tasks

**TASK-COH1**: Given that the cohesion-related hypotheses presented in Section 6.2.4 propose a link between the cohesion of services and their analysability only, the maintenance-related experimental task was related only to the identification (not repair) of the causes of failures as part of corrective maintenance. More specifically, the participants were asked to identify the causes of 20 failures related to the core functionality of the system (one failure per service operation, refer to Section 6.2.5.2). As with the coupling task TASK-COUP1 described above, an effort was made to ensure that all failures were comparable in terms of their conceptual complexity. Also, all failures can be considered as simple failures because they were local to the investigated services and required a (limited) analysis of a small sub-set of the service implementation. It was expected that it would take approximately 5-7 minutes to analyse each failure.

Note that the analysability of a given service was evaluated, using the FAE metric described in Section 6.2.3.1, based on the combined time taken to complete the failure analysis for all four failures located in this service, rather than for each individual failure. This is because we are aiming to investigate the combined cohesiveness of a service (using the TICS metric), and also because it has been suggested previously that performance across a set of similar maintenance tasks for a given software module is more important than performance related to individual tasks [88]. Approximate task duration: 120 minutes.

**TASK-COH2**: After completing the failure analysis as part of TASK-COH1, the participants were asked to carefully investigate the internal and external structure of five experimental services included in SYS-COH, and then subjectively rank the cohesion of each service on a five-point Likert scale (Section 6.2.3.2). Although this task is not related to the actual maintenance process, it was included to test the experimental hypothesis $H_{coh2}$ (Section 6.2.4). Approximate task duration: 15 minutes.

6.2.7 Experimental Design

This study employed a within-subjects design [165] where all the experimental tasks described in the previous section were attempted by each of the ten participants. A within-subjects experimental design requires fewer participants to maintain the statistical power of the experiments compared to a between-subjects (or independent-groups) design [48] where
CHAPTER 6. EMPIRICAL EVALUATION OF METRICS

each participant is exposed to one level of treatment only. Therefore, a within-subjects design was considered suitable for this study given the small number of participants. Moreover, within-subjects designs are said to be more sensitive\textsuperscript{28} compared to independent-groups designs due to the lower variability in the participants’ skills and experience [217]. That is, the error variance due to differences among participants is reduced in within-subjects designs. This is especially significant in the area of empirical software engineering where the strong variation in participant capabilities is a major concern [36].

Nevertheless, one of the limitations of a within-subjects design is that the independent variables can become confounded with the order of presentation because of practice effects [217]. Practice effects can arise due to improvements in the participants’ skills and knowledge as the experimentation progresses (this is commonly referred to as learning effects in the existing work on the empirical evaluation of metrics [36]), and degradation of participants’ abilities to perform experimental tasks due to fatigue effects. To deal with the practice effects in this study, thereby enforcing its internal validity [240], a number of techniques for balancing the practice effects were employed. More specifically: i) the participants were exposed to the various levels of treatment using the selected orders approach [217], which counterbalances learning and fatigue effects as explained in the following sub-section; and ii) the study was conducted in two separate 3-hour experimental runs (one experimental run for each structural property under investigation) in order to reduce the fatigue effects.

6.2.7.1 Counterbalancing Approach

The experimental material and tasks related to both experimental systems represent an incomplete within-subjects design, where each condition is administered to each subject only once, and the order of administering the conditions is varied systematically across participants. In contrast, in a complete (or repeated) within-subjects design, the same conditions are administered several times to each subject using a different order. To counterbalance any potential learning and fatigue effects in the incomplete within-subjects designs, it is recommended to use the selected orders approach [217] where only a sub-set of all possible orders is used, with the number of selected orders equalling some multiple of the number of conditions in the experiments.

For example, random starting order with rotation technique was employed in this study, which is an effective technique for balancing practice effects because it ensures that each condition appears in each ordinal position equally often. The technique involves choosing a random initial order (or sequence) of the conditions, which is then systematically rotated with each condition moving one position to the left on each rotation [217].

\textsuperscript{28} The sensitivity of an experiment refers to the extent to which an experiment is able to detect differences in the dependent variable that can be contributed to the independent variable.
CHAPTER 6. EMPIRICAL EVALUATION OF METRICS

<table>
<thead>
<tr>
<th>Participants</th>
<th>Experimental services allocation order (sequence)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-3</td>
<td>SER-COUP1, SER-COUP2, SER-COUP 3;</td>
</tr>
<tr>
<td>4-6</td>
<td>SER-COUP2, SER-COUP3, SER-COUP1;</td>
</tr>
<tr>
<td>7-10</td>
<td>SER-COUP3, SER-COUP1, SER-COUP2;</td>
</tr>
</tbody>
</table>

Table 6-7. The selective orders of the experimental tasks - coupling

<table>
<thead>
<tr>
<th>Participants</th>
<th>Experimental services allocation order (sequence)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 and 2</td>
<td>AMS1, AMS2, AMS3, AMS4, AMS5;</td>
</tr>
<tr>
<td>3 and 4</td>
<td>AMS2, AMS3, AMS4, AMS5, AMS1;</td>
</tr>
<tr>
<td>5 and 6</td>
<td>AMS3, AMS4, AMS5, AMS1, AMS2;</td>
</tr>
<tr>
<td>7 and 8</td>
<td>AMS4, AMS5, AMS1, AMS2, AMS3;</td>
</tr>
<tr>
<td>9 and 10</td>
<td>AMS5, AMS1, AMS2, AMS3, AMS4;</td>
</tr>
</tbody>
</table>

Table 6-8. The selective orders of the experimental tasks – cohesion

Table 6-7 and Table 6-8 show the allocation orders used in the coupling and cohesion experimental runs respectively. Note that each participant was assigned a random number (1 to 10) that was used in the random starting order with rotation procedure. Also note that four participants were allocated to the last order of the coupling experimental run (in contrast to the first two orders that had three participants) since the number of participants (ten) was not a multiple of three.

Finally note that to minimise the *anticipation effect* [217], which can arise in within-subjects design when the participants’ expectations about which condition should occur next can influence their responses, the participants were not told about the expected study outcomes or the structural specifics of sub-systems included in SYS-COUP and services included in SYS-COH.

**6.2.8 Experimental Procedure**

This section presents the specifics related to the execution of the study, including the description of pre-test training provided to the participants, and the process of executing the study proper. Additionally, a preliminary pilot study was conducted to emulate, and thus validate, different aspects of the experiment as described in sub-section 6.2.8.1. The study was conducted following the guidelines of RMIT’s Human Research Ethics Committee. That is, all participants were explicitly informed that their individual participation in the study would remain anonymous and confidential.

Prior to commencing the experiment, the overall goals of this study and the nature of the experimental tasks were explained to the participants. More specifically, the participants were asked to read a background document that provided a brief description of the topics related to
this research, including the core principles of SOC, and the general definitions of coupling and cohesion properties of software. This was done in order to ensure that all participants had sufficient and comparable knowledge needed to complete the experimental tasks. Furthermore, the participants were asked to conduct a thorough examination of the provided system-related documentation (approximately twenty pages in total) including: i) the Software Requirements Specification (SRS) document for the original AMS; and ii) the UML [210] class diagrams for the SYS-COUP and SYS-COH experimental systems. The participants were given one week to familiarise themselves with the provided documents.

At the end of the documentation familiarisation period, each participant received a short 15-minute face-to-face tutorial session from the present author, where the topics covered in the background document were described further, and the participants had a chance to ask questions related to the SRS and analysis documents of AMS. Furthermore, at the end of the tutorial session, the participants were asked to rank (anonymously) their understanding of the: i) functional requirements of the AMS system; ii) use-cases included in the analysis documents; and iii) business domain of AMS (educational institution) on a scale of 1 (low) to 5 (high). All participants indicated that they have a good understanding of these concepts (as reflected by the uniformly high rankings of 5).

The two experimental runs were conducted over two consecutive days in a controlled lab setting under the supervision of the present author. A controlled lab environment was chosen in order to eliminate any potential confounding factors, such as for example, unplanned distractions that could affect the performance of the participants. The same computer lab was used in both experimental runs. The PCs located in the lab had identical hardware and software characteristics. Eclipse 3.3 Integrated Development Environment (IDE) was used by the participants when performing the experimental tasks. All participants had prior experience with Eclipse.

At the beginning of both experimental runs, the experimental procedure was explained to the participants. The participants were then given a document describing the tasks to be performed in a particular experimental run, and asked to move to the allocated PCs which had an Eclipse project containing the UML class diagrams and source code for the experimental systems. Participants were required to perform the tasks in the order they appeared on the provided document. Note that participants were instructed not to talk between themselves, but to direct any questions to the monitor (present author). Also, the questions directed towards the monitor were not answered if thought to influence the performance of participants. All participants managed to complete all required tasks within the allocated time-frame (3 hours per experimental run).
CHAPTER 6. EMPIRICAL EVALUATION OF METRICS

6.2.8.1 Pilot Study

A pilot study using two experienced software engineers, who were not participating in the experimental runs of the main study, was conducted in the controlled lab environment under the supervision of the present author. This study was conducted in order to evaluate the similarity of the introduced failures (and associated faults), and required system changes based on the rankings assigned by software engineers, as well as the approximations of the completion times for each task.

The software engineers attempted all experimental tasks presented in Section 6.2.6 in two separate sessions. This allowed emulating the experimental runs of the main study, thereby determining whether the time allocated for each experimental run was sufficient. Additionally, the software engineers were asked to rank the conceptual and algorithmic complexity of the faults included in both experimental systems on the Likert scale of 1 (low) to 5 (high).

Both software engineers managed to successfully complete all the experimental tasks within the allocated time limit of 3 hours per experimental run. Also, the complexity of all faults and changes was ranked uniformly as low (ranks 1 or 2) by both software engineers. This was expected given the relative low complexity of the introduced faults and small size and complexity of required changes (the tasks were designed to have low complexity in order to allow completing the experimental runs within the allocated time-frame, and reducing fatigue effects).

6.2.9 Data Collection Procedure

The duration of individual tasks was collected in real time. That is, the commencement time for each maintenance task was noted (on a per participant basis), and as soon as a given participant finished one of the tasks he would indicate this to the monitor who would then record the task completion time on a PDA device. Also, after finishing all cohesion-related maintenance tasks in the second experimental run, participants completed the cohesion ranking questionnaire as their final task. The completed questionnaires were then examined by the present author, with the provided ranking data extracted and recorded as Cohesion Rank (CR) values.

In addition to recording the duration of the maintenance tasks during the experimental runs, the modified systems were later examined and tested (using a suite of unit and integration tests) by the present author in order to collect the data needed to calculate the values of the ISO/IEC maintainability metrics (Section 6.2.3.1). This was necessary since the participants were asked to implement the changes (fix the faults or implement a new use-case), but were not asked to test the experimental systems after the changes were made.

29 All participants were male as described in Section 6.2.2.
The following data was collected during/after the experimental runs:

- **Experimental Run 1 (SYS-COUP):** The data was collected individually for [incoming/outgoing] lowly- and highly-coupled implementation elements in each of the three experimental services (or five sub-services) of SYS-COUP on a per task basis as follows:

  **TASK-COUP1**
  
  *Failure Analysis Efficiency (FAE)*
  - Time (in minutes) taken to analyse the cause of failure in three experimental services for each pair of [outgoing] lowly- and highly-coupled elements for every participant (1 failure x 2 elements x 3 services x 10 participants = 60 data points);
  - The correctness (number of failures of which causes are found) of the failure analysis described above (60 data points). The correctness was evaluated in terms of a Boolean value [true/false]. That is, the failure analysis was considered correct only when all causes of a given failure were found.

  **TASK-COUP2**
  
  *Modification Complexity (MC)*
  - Time (in minutes) taken to implement a change to the functional requirement in three experimental services for each pair of [incoming/outgoing] lowly- and highly-coupled elements for every participant (1 change x 3 services x 4 elements x 10 participants = 120 data points);

  *Modification Impact Localisation (MIL)*
  - Number of detected faults in the system after changes to the functionality described above for each pair of [incoming] lowly- and highly-coupled elements for every participant (1 change x 3 services x 2 elements x 10 participants = 60 data points);

- **Experimental Run 2 (SYS-COH):** The data was collected individually for all five experimental services of SYS-COH on a per task basis as follows:

  **TASK-COH1**
  
  *Failure Analysis Efficiency (FAE)*
  - *Combined* time (in minutes) taken to determine the causes of all four failures for each of the five experimental services for every participant (5 services x 10 participants = 50 data points);
  - The correctness (number of failures of which causes are found) of the failure analysis described above (50 data points).

  **TASK-COH2**
  
  *Cohesion Rank (CR)*
  - The provided Cohesion Rank (CR) values for each of the five experimental services for every participant were extracted from the cohesion ranking questionnaire (5 services x 10 participants = 50 data points).
6.2.10 Analysis Procedure

The collected experimental data was analysed using the available statistical formulas included in the Microsoft Office Excel 2007 software package [59]. Additionally, the Analyse-it 3.0 [6] software tool was used for the statistical tests not covered in Excel 2007.

The first step of any analysis procedure is to check the normality of the data [36]. If the data is sufficiently normal, parametric tests can be applied. If the data is not sufficiently normal, non-parametric tests are more appropriate. The Shapiro-Wilk test, which is a recommended normality test for smaller (up to 1000) sample sizes [140], was employed in order to check for the normal distribution of the dependent variables. The results of the tests, which were performed under the 95% Confidence Interval (CI), showed that the distribution of all dependent variables collected in both experimental runs did not deviate significantly from normality as reflected by high Shapiro-Wilk (W) values (all W values were in a range of 0.75-0.95). This suggested that parametric tests were suitable for this study. Nevertheless, the significance probability values (p) for the tests were high (all p > 0.27), most likely due to a relatively small number of data points, thereby suggesting that it is probable that the observed results of the normality tests are incorrect. As such, the decision was made to also use an alternative non-parametric test for each selected parametric test. In every case, the non-parametric test supported the findings of the parametric one.

Additionally, a level of significance \( \alpha \) must be set, which reflects the probability of making a decision to reject a null hypothesis when the null hypothesis is actually true (also known as a Type I error). The decision itself is made using the p value. More specifically, if the p value is less than the level of significance \( \alpha \), then the null hypothesis is rejected. The smaller the p value, the more significant the result is said to be. In this study, \( \alpha \) was set to the commonly used (scientific) level of 0.05. Note that it is not uncommon to use higher levels of \( \alpha \) in software engineering experiments. For example, Briand et al. [36] suggest that an \( \alpha \) value as high as 0.2 might be considered good enough to make an informed decision regarding practical utility of software metrics, even though the empirical evidence is not strong enough to make a scientific statement with a high degree of confidence.

Finally, a power analysis is useful for determining that the experiment possesses sufficient statistical power (expressed as 1 - \( \beta \)), thereby reducing the chance of committing a Type II error (failing to reject a null hypothesis when the null hypothesis is actually false) [48]. Conventionally, a test with a power value of 0.8 (that is, \( \beta <=0.2 \)) is considered statistically powerful [162], meaning that if an experiment is run ten times, an existing effect will be discovered at least eight times out of the ten experimental runs. The power analysis itself is based on three factors: i) the number of participants used in the experiments; ii) the level of significance \( \alpha \); and iii) the effect size. Note that the procedure of calculating the effect size
values is dependent on the experimental design (e.g. within- or between-subjects) and specific statistical tests in use [140].

The G*Power 3 [69] software tool was used to conduct the power analysis in order to estimate the required effect sizes for the statistical tests described later in this section. The analysis showed that given our: i) small (and fixed) sample size of 10 participants used in the within-subjects design; ii) chosen level of $\alpha = 0.05$; and iii) a commonly-used power value of 0.8; the effect sizes required to achieve the statistically significant results for both coupling and cohesion-related tests can be classified as large according to the categorisation of Cohen [48]. For example, Cohen defined the conventional effect sizes ($f$) for one-way ANOVA as: small $f = 0.1$; medium $f = 0.3$; and large $f \geq 0.5$ [48]; and the actual calculated required effect sizes for one-way ANOVA tests employed in this study (described below) range from 0.5 to 0.6. This suggests that the statistical tests performed in this study could fail to detect a meaningful relationships between the independent and dependent variables when such relationships do not exhibit large effect sizes. To this end, the results of the statistical analyses presented in Sections 6.3 and 6.4 could be susceptible to Type II error.

### 6.2.10.1 Statistical Tests

A number of parametric and non-parametric statistical approaches and tests have been employed to evaluate the experimental hypotheses. The approaches were chosen based on their suitability. For example, the within-subjects experimental design adopted in the study requires the selection of statistical tests that cover data related to repeated measures. Also, correlation and regression analysis was applied to the data collected for the cohesion system SYS-COH given the sufficient number of levels of the independent variable (five levels of TICS). In contrast, such analysis was not applicable to the coupling-related data comprising only two levels (lowly- and highly-coupled) of independent variables. Table 6-9 shows the statistical approaches used in this study, the complete description of which can be found in [48, 58, 61, 140].

### 6.3 Analysis of the Results – Coupling Hypotheses

This section presents and analyses the results of the statistical tests used to evaluate the coupling hypotheses $H_{\text{coup1.1}} \sim H_{\text{coup1.6}}$ and $H_{\text{coup2.1}} \sim H_{\text{coup2.6}}$ defined in Section 6.2.4. Note that the selected statistics shown for different tests presented in this section, and the following section dedicated to the cohesion hypotheses, include the recommended statistical data for a particular test in use (for example, Sum squares, Mean square, F statistic, F critical value, and p-value are provided for one-way ANOVA tests [117]).
### Table 6-9. Summary of the statistical analysis techniques used in the study

<table>
<thead>
<tr>
<th>GOAL/HYPOTHESIS</th>
<th>STATISTICAL TESTS</th>
</tr>
</thead>
</table>
| GOAL-COUP1 / $H_{coup1.1} - H_{coup1.6}$ | - A standard **paired t-test for dependent samples** was used to evaluate the relationship between the maintainability values collected for lowly- and highly-coupled elements in each experimental sub-system (one test per hypothesis). More specifically, t-test allowed testing the null hypothesis that the means of a group of dependent variables collected for lowly-coupled elements are equal to a group of dependent variables collected for the highly-coupled elements. **Type**: Parametric;  
- The **Wilcoxon matched pairs** test was used as a **non-parametric** alternative to the paired t-test for dependent samples. |
| GOAL-COUP2 / $H_{coup2.1} - H_{coup2.6}$ | - A **one-way ANOVA for repeated measures** was used to evaluate the relationships between the different types of coupling and the maintainability values collected for all highly-coupled elements in each experimental sub-system. The one-way ANOVA has the same theoretical foundation and purpose as the paired t-test, except that it is used to test for differences among at least three groups (for example, intra-service, indirect extra-service, and direct extra-service maintainability values). The null hypothesis is that all the groups have the same mean, and the alternate hypothesis is that at least one of the means is different from the others. **Type**: Parametric;  
- The **Kruskall-Wallis** test was used as a **non-parametric** alternative to one-way ANOVA for repeated measures approach. |
| GOAL-COH1 and GOAL-COH2 / $H_{coh1}$ and $H_{coh2}$ | - A **one-way ANOVA for repeated measures** technique was used to evaluate the relationships between the different levels of service cohesion, and the analysability values and cohesion ranks collected for each experimental service. **Type**: Parametric;  
- The **Kruskall-Wallis** test was used as a **non-parametric** alternative to one-way ANOVA for repeated measures approach.  
- A **simple (univariate) linear regression** analysis was conducted to evaluate, using the least-squares method, the strength of the linear regressive effect ($R^2$) of the values of TICS on i) the values of Failure Analysis Efficiency (FAE) metric; and ii) the values of Cohesion Ranks (CR). **Type**: Parametric; |
6.3.1 Descriptive Statistics

The following descriptive summaries provide values for the dependent variables designed to quantify the maintainability sub-characteristics under study, thereby supporting the interpretation of the analytical results in the remainder of this section. Moreover, descriptive statistics can facilitate the comparison of results from future studies.

The descriptive statistics are presented in Table 6-10 - Table 6-12 individually for each dependent variable, with the exception of the stability metric, Modification Impact Localisation (MIL). The values of MIL are not included because all participants managed to perform TASK-COUP2 (described in Section 6.2.6.1), in all experimental services, without introducing any new system faults, and as such, all MIL values were equal to zero. Therefore, it is evident prima facie that there is no significant difference between the mean values of MIL obtained for the investigated lowly- and highly coupled elements, resulting in rejection of the alternative hypothesis $H_{coup1.4}$. We believe that this absence of failures occurred due to the small size of the experimental services, and also because the description of the tasks designed to evaluate the stability of a system (Appendix F, Task 1.b) had explicitly asked the participants to make sure that any changes to the existing functionality do not result in a negative effect on the system. To this end, additional experimental studies should be conducted in future work to conduct more thorough examination of stability.

The values for the remaining two dependent variables, Failure Analysis Efficiency (FAE) and Modification Complexity (MC) metrics, are shown for each of the lowly (Element1) and highly (Element2) coupled elements in each experimental service, where a lower value of a given dependent variable indicates better maintainability. The provided values cover the minimum and maximum levels, as well as the values of central tendency (mean and median) and dispersion (standard deviation and variance). Additionally, Figure 6-1 - Figure 6-3 provide a graphical overview of the results, showing mean values for the (dependent) maintainability metrics collected in each task.

Finally, note that the Modification Complexity (MC) changeability metric is used to evaluate both incoming and outgoing coupling types given that: i) the changeability of a system was hypothesized to be related to the number of incoming relationships (Hypothesis $H_{coup1.2}$); and ii) the changeability of an element was hypothesized to be related to the number of outgoing relationships (Hypothesis $H_{coup1.4}$).

---

30 Note that the raw data can be obtained from the present author on request.
Table 6-10. Descriptive Statistics: TASK-COUP1 (FAE values – outgoing coupling)

<table>
<thead>
<tr>
<th>SUB-SYSTEM / ELEMENT</th>
<th>MIN</th>
<th>MAX</th>
<th>MEAN</th>
<th>MEDIAN</th>
<th>STD. DEV.</th>
<th>VARIANCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SER-COUP1b (designed to investigate intra-service outgoing coupling effects) – refer to Section 6.2.5.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Element1 (WISCE =1)</td>
<td>2</td>
<td>5</td>
<td>3.1</td>
<td>3</td>
<td>1.2</td>
<td>1.43</td>
</tr>
<tr>
<td>Element2 (WISCE =5)</td>
<td>4</td>
<td>10</td>
<td>6</td>
<td>5.5</td>
<td>1.76</td>
<td>3.11</td>
</tr>
<tr>
<td>SER-COUP2b (covers indirect extra-service outgoing coupling) – refer to Section 6.2.5.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Element1 (EESIOC =1)</td>
<td>3</td>
<td>8</td>
<td>4.7</td>
<td>4</td>
<td>1.89</td>
<td>3.57</td>
</tr>
<tr>
<td>Element2 (EESIOC =5)</td>
<td>5</td>
<td>12</td>
<td>7.9</td>
<td>7</td>
<td>2.56</td>
<td>6.54</td>
</tr>
<tr>
<td>SER-COUP3b (covers direct extra-service outgoing coupling) – refer to Section 6.2.5.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Element1 (WESOCE =1)</td>
<td>2</td>
<td>8</td>
<td>4.7</td>
<td>4</td>
<td>2.06</td>
<td>4.23</td>
</tr>
<tr>
<td>Element2 (WESOCE =5)</td>
<td>4</td>
<td>11</td>
<td>7.6</td>
<td>8</td>
<td>2.91</td>
<td>8.49</td>
</tr>
</tbody>
</table>

Figure 6-1. TASK-COUP1 - Failure Analysis Efficiency (FAE)
Table 6-11. Descriptive Statistics: TASK-COUP2 (MC values - incoming coupling)

<table>
<thead>
<tr>
<th>SUB-SYSTEM / ELEMENT</th>
<th>MIN</th>
<th>MAX</th>
<th>MEAN</th>
<th>MEDIAN</th>
<th>STD. DEV.</th>
<th>VARIANCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SER-COUP1b (intra-service coupling)</td>
<td>4</td>
<td>8</td>
<td>5.6</td>
<td>5.5</td>
<td>1.51</td>
<td>2.27</td>
</tr>
<tr>
<td>Element1 (WISCE =1)</td>
<td>6</td>
<td>15</td>
<td>8.8</td>
<td>8.5</td>
<td>2.49</td>
<td>6.18</td>
</tr>
<tr>
<td>Element2 (WISCE =5)</td>
<td>3</td>
<td>8</td>
<td>5.2</td>
<td>5</td>
<td>1.4</td>
<td>1.96</td>
</tr>
<tr>
<td>SER-COUP2b (indirect extra –service outgoing coupling)</td>
<td>7</td>
<td>20</td>
<td>12.2</td>
<td>12</td>
<td>3.77</td>
<td>14.18</td>
</tr>
<tr>
<td>Element1 (EESIOC =1)</td>
<td>2</td>
<td>10</td>
<td>5.1</td>
<td>5</td>
<td>2.23</td>
<td>4.99</td>
</tr>
<tr>
<td>Element2 (WESOCE =5)</td>
<td>5</td>
<td>20</td>
<td>12.4</td>
<td>12.5</td>
<td>3.89</td>
<td>15.16</td>
</tr>
</tbody>
</table>

Table 6-12. Descriptive Statistics: TASK-COUP2 (MC values - outgoing coupling)
6.3.2 Hypothesis Testing

Eight paired t-tests for dependent samples were conducted in order to examine the impact of low and high design-level coupling on the analysability and changeability of service-oriented software according to hypotheses $H_{coup1.1} - H_{coup1.6}$ defined in Section 6.2.4. Additionally, three one-way ANOVA for repeated measures tests were conducted to determine whether there is a statistically significant difference in the analysability and changeability values for the highly-coupled elements belonging to six experimental (sub-)services of SYS-COUP according to hypotheses $H_{coup2.1} - H_{coup2.6}$ (Section 6.2.4).

6.3.2.1 Hypotheses $H_{coup1.1} - H_{coup1.6}$: Maintainability Impact

There is a statistically significant relationship between the selected service-oriented coupling metrics and the ISO/IEC analysability (FAE) and changeability (MC) metrics as was confirmed by a series of paired t-tests. This suggests that the evaluated coupling metrics can be used as early indicators of analysability and changeability of service-oriented software.

More specifically, a two-tailed paired t-test was used to determine whether the population means\(^{32}\) ($\mu$) of the groups of maintainability values sampled for each lowly- ($e1$) and highly-($e2$) coupled element in each of the experimental services are not equal (that is, $\mu (e_{1DV}) \neq \mu (e_{2DV})$, where DV stands for a dependent variable). For example, hypothesis $H_{coup1.2}$ states

---

\(^{31}\) Note that the stability sub-characteristic was not statistically analysed given that all values of the corresponding metric MIL were equal to zero as described in Section 6.3.1.

\(^{32}\) Population mean ($\mu$) is the mean of all members of a population (or all possible values from which a sample can be taken).
### Table 6-13. Paired t-test results (two-tailed p-values) for Hypothesis $H_{coup1.1} - H_{coup1.6}$ (n=10)

<table>
<thead>
<tr>
<th>Hypothesis</th>
<th>$H_{coup1.1}$</th>
<th>$H_{coup1.2}$</th>
<th>$H_{coup1.3}$</th>
<th>$H_{coup1.4}$</th>
<th>$H_{coup1.5}$</th>
<th>$H_{coup1.6}$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Service</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SER-COUP1a</strong></td>
<td>e1 (WISCE=1)</td>
<td>e2 (WISCE=5)</td>
<td>e1 (WISCE=1)</td>
<td>e2 (WISCE=5)</td>
<td>e1 (WISCE=1)</td>
<td>e2 (WISCE=5)</td>
</tr>
<tr>
<td><strong>SER-COUP1b</strong></td>
<td>e1 (WISCE=1)</td>
<td>e2 (WISCE=5)</td>
<td>e1 (WISCE=1)</td>
<td>e2 (WISCE=5)</td>
<td>e1 (WISCE=1)</td>
<td>e2 (WISCE=5)</td>
</tr>
<tr>
<td><strong>SER-COUP2a</strong></td>
<td>e1 (ESICSI=1)</td>
<td>e2 (ESICSI=5)</td>
<td>e1 (ESICSI=1)</td>
<td>e2 (ESICSI=5)</td>
<td>e1 (ESICSI=1)</td>
<td>e2 (ESICSI=5)</td>
</tr>
<tr>
<td><strong>SER-COUP2b</strong></td>
<td>e1 (ESICSI=1)</td>
<td>e2 (ESICSI=5)</td>
<td>e1 (ESICSI=1)</td>
<td>e2 (ESICSI=5)</td>
<td>e1 (ESICSI=1)</td>
<td>e2 (ESICSI=5)</td>
</tr>
<tr>
<td><strong>SER-COUP3a</strong></td>
<td>e1 (WESICE=1)</td>
<td>e2 (WESICE=5)</td>
<td>e1 (WESICE=1)</td>
<td>e2 (WESICE=5)</td>
<td>e1 (WESICE=1)</td>
<td>e2 (WESICE=5)</td>
</tr>
<tr>
<td><strong>SER-COUP3b</strong></td>
<td>e1 (WESICSI=1)</td>
<td>e2 (WESICSI=5)</td>
<td>e1 (WESICSI=1)</td>
<td>e2 (WESICSI=5)</td>
<td>e1 (WESICSI=1)</td>
<td>e2 (WESICSI=5)</td>
</tr>
<tr>
<td><strong>Test Groups</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\mu (e1) \neq \mu (e2)$</td>
<td>not related</td>
<td>&lt;0.0001</td>
<td>not related</td>
<td>0.0009</td>
<td>not related</td>
<td>0.012</td>
</tr>
<tr>
<td><strong>FAE (Task 1)</strong></td>
<td>0.0003</td>
<td>0.0046</td>
<td>0.0004</td>
<td>&lt;0.0001</td>
<td>&lt;0.0001</td>
<td>0.0001</td>
</tr>
<tr>
<td><strong>MC (Task 2)</strong></td>
<td>not related</td>
<td>&lt;0.0001</td>
<td>not related</td>
<td>0.0009</td>
<td>not related</td>
<td>0.012</td>
</tr>
</tbody>
</table>

that an element with a high value of indirect extra-service incoming coupling metric (ESICSI) will result in significantly lower system changeability (Modification Complexity (MC) metric) compared to an element with a low value of ESICSI, where the corresponding test condition can be expressed as $\mu (e1_{MC}) \neq \mu (e2_{MC})$.

The statistical analysis suggests that for both corrective and perfective maintenance activities, there is a significant variance in the maintainability values in each pair of lowly- and highly-coupled elements. That is, all p-values are below a statistically significant level of 0.05 as shown in Table 6-13. Moreover, the Wilcoxon matched pairs ($Z$) test was used as a non-parametric alternative to the paired t-test. The results of Wilcoxon tests also indicate a significant variance in the groups of FAE and MC values for all examined lowly- and highly-coupled elements (all $Z$-values were above $Z$ critical value at the significance levels $< 0.05$). Therefore, we accept alternative experimental hypotheses $H_{coup1.1} - H_{coup1.6}$.

Note that this outcome is not surprising since it was shown in previous paradigms that the structural property of coupling is correlated to the maintainability of software. Nevertheless, accepting hypotheses $H_{coup1.1} - H_{coup1.6}$ gives us confidence that the derived SO coupling metrics are empirically valid (or useful) measures of coupling.

#### 6.3.2.2 Hypotheses $H_{coup2.1} - H_{coup2.6}$: Strength of Coupling Relationships

The general impact of low and high coupling, measured by the SO coupling metrics, on the maintainability of software was evaluated in the previous section. The aim of the analysis presented in this section was to determine whether the specific types of coupling relationships have a significantly different impact on the analysability and changeability of SO software according to the hypotheses $H_{coup2.1} - H_{coup2.6}$ defined in Section 6.2.4. Note that as with the general maintainability evaluation described in the previous sub-section, the stability sub-characteristic was not tested given that all values of the corresponding metric (MIL) were equal to zero as described in Section 6.3.1.
The results of one-way ANOVA tests indicate:

i) a statistically significant variance in the three groups of changeability metric (Modification Complexity (MC)) values collected for different types of coupling relationships (intra-service incoming/outgoing coupling, indirect/direct extra-service incoming coupling, and indirect/direct extra-service outgoing coupling). More specifically, the pair-wise comparison tests conducted as part of ANOVA indicate a significant difference between MC values collected for both incoming and outgoing extra-service relationships and the intra-service relationship. This suggests that all evaluated direct/indirect and incoming/outgoing extra-service coupling metrics (ESICSI, EESIOC, WESICE, and WESOCE) should be weighted higher than the intra-service metric (WISCE) in order to reflect their relatively stronger impact on the changeability of SO software. This in turn suggests that the proposed metrics can potentially be more accurate indicators of changeability of service-oriented software compared to the existing OO metric CBO (which is considered to be structurally equivalent to the WISCE metric, as described in Section 4.3).

ii) an absence of statistically significant variance in the three groups of analysability metric (Failure Analysis Efficiency (FAE)) values collected for different types of coupling relationships (intra-service outgoing coupling and indirect/direct extra-service outgoing coupling). More specifically, although the obtained values of FAE were in the hypothesised direction, the current statistical evidence is not significant enough to establish that extra-service coupling has greater effect on the analysability of SO software compared to the intra-service coupling.

The following describes the specifics of the one-way ANOVA for repeated measures tests employed to evaluate hypotheses $H_{coup2.1}$ - $H_{coup2.6}$, and thus determine whether there is a significant difference in the maintainability values for the (highly-coupled) elements belonging to the different services of SYS-COUP.

- The maintainability of six different elements was investigated in order to test the experimental hypotheses:
  - Element1 ($e_1$) belongs to service SER-COUP1a and has a high value (5) of (incoming) WISCE (intra-service coupling);
  - Element2 ($e_2$) belongs to service SER-COUP1b and has a high value (5) of (outgoing) WISCE (intra-service coupling);
  - Element3 ($e_3$) belongs to SER-COUP2a and has a high value (5) of ESICSI (indirect extra-service incoming coupling);
  - Element4 ($e_4$) belongs to SER-COUP2b and has a high value (5) of EESIOC (indirect extra-service outgoing coupling);
- Element 5 (e5) belongs to SER-COUP3a and has a high value (5) of WESICE (direct extra-service incoming coupling);
- Element 6 (e6) belongs to SER-COUP3b and has a high value (5) of WESOCE (direct extra-service outgoing coupling);

• The individual test conditions related to the experimental hypotheses $H_{coup2.1}$ - $H_{coup2.6}$ were then established for two groups of dependent variables (analysability (FAE metric) and changeability (MC metric)) values as follows:

  - $H_{coup2.1}$ - an element with a high value of incoming WISCE will result in significantly higher (better) system changeability (MC) compared to an element with the same value of ESICSI. Test condition: $\mu (e1_{MC}) \neq \mu (e3_{MC})$;
  - $H_{coup2.2}$ - an element with a high value of outgoing WISCE will exhibit significantly higher (better) analysability (FAE) and changeability (MC) compared to an element with the same value of EESIOC. Test condition: $\mu (e2_{FAE}) \neq \mu (e4_{FAE}) \land \mu (e2_{MC}) \neq \mu (e4_{MC})$;
  - $H_{coup2.3}$ - an element with a high value of incoming WISCE will result in significantly higher system changeability (MC) compared to an element with the same value of WESICE. Test condition: $\mu (e1_{MC}) \neq \mu (e5_{MC})$;
  - $H_{coup2.4}$ - an element with a high value of outgoing WISCE will exhibit a significantly higher analysability (FAE) and changeability (MC) compared to an element with the same high value of WESOCE. Test condition: $\mu (e2_{FAE}) \neq \mu (e6_{FAE}) \land \mu (e2_{MC}) \neq \mu (e6_{MC})$;
  - $H_{coup2.5}$ - an element with a high value of ESICSI will result in significantly higher system changeability (MC) compared to an element with the same high value of WESICE. Test condition: $\mu (e3_{MC}) \neq \mu (e5_{MC})$;
  - $H_{coup2.6}$ - an element with a high value of EESIOC will exhibit significantly higher analysability (FAE) and changeability (MC) compared to an element with the same high value of WESOCE. Test condition: $\mu (e4_{FAE}) \neq \mu (e6_{FAE}) \land \mu (e4_{MC}) \neq \mu (e6_{MC})$;

• Finally, the individual test conditions were aggregated together into three one-way ANOVA tests shown below:

  - **Test 1** ($H_{coup2.1}, H_{coup2.3}, \text{and } H_{coup2.5}$): $\mu (e1_{MC}) \neq \mu (e3_{MC}) \neq \mu (e5_{MC})$;
  - **Test 2** ($H_{coup2.2}, H_{coup2.4}, \text{and } H_{coup2.6}$): $\mu (e2_{FAE}) \neq \mu (e4_{FAE}) \neq \mu (e6_{FAE})$;
  - **Test 3** ($H_{coup2.2}, H_{coup2.4}, \text{and } H_{coup2.6}$): $\mu (e2_{MC}) \neq \mu (e4_{MC}) \neq \mu (e6_{MC})$.

Note that the individual test conditions were aggregated and evaluated using (3) one-way ANOVA tests, rather than a collection of (9) t-tests, in order to minimise the likelihood of committing a Type I error which is sensitive to a number of conducted tests [140]. Also, two separate tests (Test 2 and Test 3) were conducted to examine the conjunctive conditions stated in the hypotheses $H_{coup2.2}, H_{coup2.4}, \text{and } H_{coup2.6}$ (Section 6.2.4).

(Feburary, 2009)
CHAPTER 6. EMPIRICAL EVALUATION OF METRICS

<table>
<thead>
<tr>
<th>SOURCE OF VARIATION</th>
<th>SUM SQUARES</th>
<th>MEAN SQUARE</th>
<th>F STATISTIC</th>
<th>F CRITICAL</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Groups (between)</td>
<td>53.1</td>
<td>26.5</td>
<td>3.68</td>
<td>3.35</td>
<td>0.038</td>
</tr>
<tr>
<td>Residual (within groups)</td>
<td>194.8</td>
<td>7.2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>247.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6-14. one-way ANOVA results for MC values (TASK-COUP2) – incoming coupling (n=30)

The results of the ANOVA tests are presented below. Note that there are two important ANOVA-related statistics used to interpret the results of the tests: i) $F$ statistic that provides an indication of the lack of fit of the data to the estimated values of the regression; and ii) $F$ critical value which is used to evaluate the outcome of the test. If the $F$ statistic equals or exceeds the $F$ critical value, the null hypothesis is rejected (and the alternative hypothesis is accepted), thereby suggesting that the population means $\mu$ of the groups sampled are not equal. Also, the probability $p$ indicates whether the observed effect is statistically significant.

- **Test 1 ($H_{coup2.1}$, $H_{coup2.3}$, and $H_{coup2.5}$):** ($e1_{MC}$) != ($e3_{MC}$) != $\mu$ ($e5_{MC}$)

The results of the ANOVA test conducted to test the inequality of variance in the groups of Modification Complexity (MC) values collected for three experimental elements $e1$, $e3$, and $e5$ are presented in Table 6-14. The results indicate that the impact of different types of investigated coupling relationships (intra-service, indirect extra-service incoming, and direct extra-service incoming) on the changeability of service-oriented software is statistically significant when performing perfective (TASK-COUP2) maintenance as measured by MC.

More specifically, the population means of the three sampled groups of MC values are not equal, with $F$ statistic (3.68) exceeding the $F$ critical value (3.35) at the statistically significant level of 0.05 ($p$-value = 0.038). Additionally, the Kruskal-Wallis test was conducted as a non-parametric alternative to ANOVA. The results of this test also show a significant variance between the groups of MC values (the Kruskal-Wallis’ statistic ($H$) = 6.63, with $p$ = 0.036 (corrected for ties)).

At this point, we are also interested in doing pair-wise comparisons of the means (which can only be performed if the alternative hypothesis was accepted by ANOVA). That is, the statistically significant ANOVA results described above showed that there is more variation between tested groups than would be expected by chance, but they did not identify which specific group pairs are significantly different from one another. Therefore, given our goal to evaluate the relative strength of different coupling relationship types (GOAL-COUP2, Section 6.2.1), the Fisher’s Least-Significant Difference (LSD)\(^{33}\) mean comparison test was applied for the purpose of comparing differences between the individual groups.

---

\(^{33}\) LSD is a widely-used method for comparing treatment group means in a pair-wise manner as part of ANOVA.
The results of the LSD comparison test are shown in Table 6-15. Note that LSD tests do not provide the significance value ($p$), but rather produce a Boolean value that reflects the significance of investigated group pairs. The LSD test indicates that there is a significant difference in population means of the MC values for: i) intra-service incoming coupling ($e_1$) and indirect extra-service incoming coupling ($e_3$); and ii) intra-service coupling ($e_1$) and direct extra-service incoming coupling ($e_5$); but the relationship between indirect ($e_3$) and direct ($e_5$) extra-service incoming coupling types is not significant at 0.05 level.

Therefore, we accept the changeability-related component of the combined (refer to Section 6.2.4.2) alternative hypotheses $H_{coup2.1}$ and $H_{coup2.3}$, but reject the changeability-related component of the alternative hypothesis $H_{coup2.5}$.

This leads to the following conclusions:

- Both indirect and direct extra-service incoming coupling types, as measured by the ESICSI and WESICE metrics respectively, should be weighted higher than intra-service coupling (measured using the WISCE metric) because they have a stronger influence on the changeability of SO software. Note that the actual weight values are not defined in this thesis due to a lack of empirical data. The weights could be established and validated in future work as discussed further in Chapter 7.

- The ESICSI and WESICE metrics are more accurate indicators of service-oriented software changeability compared to the CBO metric which is considered to be syntactically and structurally similar to the WISCE metric used to quantify the intra-service coupling of an element. This is because WISCE, and correspondingly CBO, was designed to quantify the general coupling between elements (OO classes in a case of CBO) disregarding the strength of specific types of coupling relationships which are shown to have varying impact on the stability of SO software.

- The current statistical evidence is not significant enough to establish that the direct extra-service coupling should be ranked (and thus weighted) higher than indirect coupling as was hypothesised in $H_{coup2.5}$. More studies could be conducted in future work to further test this hypothesis.
CHAPTER 6. EMPIRICAL EVALUATION OF METRICS

<table>
<thead>
<tr>
<th>SOURCE OF VARIATION</th>
<th>SUM SQUARES</th>
<th>MEAN SQUARE</th>
<th>F STATISTIC</th>
<th>F CRITICAL</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Groups (between)</td>
<td>20.9</td>
<td>10.4</td>
<td>1.73</td>
<td>3.35</td>
<td>0.197</td>
</tr>
<tr>
<td>Residual (within groups)</td>
<td>163.3</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>184.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6-16. one-way ANOVA results for FAE values (TASK-COUP1) – outgoing coupling (n=30)

- **Test 2 (H_{coup2.2}, H_{coup2.4}, and H_{coup2.6}): (e2_{FAE}) != (e4_{FAE}) != \mu (e6_{FAE})**

The results of the ANOVA conducted to test the inequality of variance in the groups of Failure Analysis Efficiency (FAE) values collected for three experimental elements e2, e4, and e6 are presented in Table 6-16.

The results indicate that different types of investigated coupling relationships (intra-service outgoing, indirect extra-service outgoing, and direct extra-service outgoing) do not have a significantly different impact on the *analysability* of service-oriented products, as reflected by F statistic (1.73) being below the F critical value (3.35) with p-value = 0.197. As such, we reject the analysability-related component of the alternative hypotheses H_{coup2.2}, H_{coup2.4}, and H_{coup2.6}. Consequently, the Fisher’s Least-Significant Difference (LSD) mean comparison test was not conducted as part of Test 2.

- **Test 3 (H_{coup2.2}, H_{coup2.4}, and H_{coup2.6}): (e2_{MC}) != (e4_{MC}) != \mu (e6_{MC})**

The results of the ANOVA conducted to test the inequality of variance in the groups of MC values collected for three experimental elements e2, e4, and e6 are presented in Table 6-17.

The results indicate that different types of investigated coupling relationships (intra-service outgoing, indirect extra-service outgoing, and direct extra-service outgoing) have statistically different impact on the *changeability* of service-oriented software. That is, the F statistic (3.46) exceeds the F critical value (3.35) at the statistically significant level of 0.05. This is further supported by the Kruskal-Wallis tests (H = 7, with p = 0.03 (corrected for ties)).

Given that the ANOVA result was statistically significant, the Fisher’s Least-Significant Difference (LSD) test was again applied for the purpose of comparing differences between the individual groups. The results of the LSD test, shown in Table 6-18, are similar to the results obtained during the evaluation of the (incoming) changeability sub-characteristic in Test 1. More specifically, the results indicate statistically significant difference in population means of the MC values for the: i) (outgoing) intra-service coupling (e2) and indirect extra-service outgoing coupling (e4); and ii) intra-service coupling (e2) and direct extra-service outgoing coupling (e6); but the relationship between indirect (e4) and direct (e6) extra-service outgoing coupling is not significant at 0.05 level. As such, we accept the changeability-related component of the alternative hypothesis H_{coup2.2} and H_{coup2.4}, and reject the changeability-related component of the alternative hypothesis H_{coup2.6}.  

(February, 2009)
This leads to a conclusion that both indirect and direct extra-service outgoing coupling should be ranked (and weighted) higher than intra-service coupling, but the current statistical evidence is not strong enough to establish the relative weights for the direct and indirect extra-service outgoing coupling covered in $H_{coup2.6}$. Moreover, as was the case with the incoming coupling metrics evaluated in Test 1, the metrics for measuring the indirect and direct extra-service outgoing relationships appear to be more accurate indicators of changeability of service-oriented systems compared to CBO.

### 6.4 Analysis of the Results – Cohesion Hypotheses

This section presents and discusses the results of the statistical tests (described in Section 6.2.10, Table 6-9) used to evaluate the cohesion hypotheses $H_{coh1}$ and $H_{coh2}$ defined in Section 6.2.4.

#### 6.4.1 Descriptive Statistics

The descriptive statistics provide the values for Failure Analysis Efficiency (FAE) and Cohesion Rank (CR) measures for each of the experimental services, thereby supporting the interpretation of the analysis results in the remainder of this section. The descriptive summaries, which are presented in Table 6-19 and Table 6-20 individually for each dependent variable, cover the minimum and maximum values, as well as the values of central tendency (mean and median) and dispersion (standard deviation and variance). The raw data can be obtained from the present author on request.
It can be observed that the values for both dependent variables are in the hypothesised direction. That is, the mean FAE value decreases (indicating better analysability of a service), and the mean CR value increases (indicating better subjective cohesion ranking) with each controlled increase of service cohesion level (TICS value). Additionally, it can be observed that the variance in FAE values obtained for the experimental services is systematically decreasing for services exhibiting better cohesion. This suggests that the failure analysis procedure for more cohesive services was performed in a more uniform manner across participants, and as such, the prediction of analysability can be performed more accurately and consistently for highly-cohesive services.

### 6.4.2 Hypothesis Testing

Two one-way ANOVA for repeated measures tests were conducted to determine whether there is a significant difference in the analysability (FAE), and subjective cohesion rankings (CR) for each of the five experimental services of experimental system SYS-COH. This was done in order to test the cohesion hypotheses $H_{coh1}$ and $H_{coh2}$. The results of the tests are discussed below individually for each of the dependent metrics.
Table 6-21. one-way ANOVA results for the Hypothesis H_{coh1} (n=50)

<table>
<thead>
<tr>
<th>SOURCE OF VARIATION</th>
<th>SUM SQUARES</th>
<th>MEAN SQUARE</th>
<th>F STATISTIC</th>
<th>F CRITICAL</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Groups (between)</td>
<td>26.56</td>
<td>6.64</td>
<td>2.74</td>
<td>2.58</td>
<td>0.04</td>
</tr>
<tr>
<td>Residual (within groups)</td>
<td>109.07</td>
<td>2.42</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>135.63</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6.4.2.1 Hypothesis H_{coh1}: Analysability Impact

The one-way ANOVA test indicated that there is a statistically significant relationship between the Total Cohesion of a Service (TICS) metric (and thus, its constituent metrics SIDC, SIUC, SIIC, and SISC) and the ISO/IEE analysability metric Failure Analysis Efficiency (FAE). This suggests that the derived cohesion metrics can be used as early indicators of analysability of service-oriented software.

The statistics related to the ANOVA conducted to test the inequality of variance in the groups of FAE values for five experimental cohesion services (μ (AMS1_{FAE}) ≠ μ (AMS2_{FAE}) ≠ μ (AMS3_{FAE}) ≠ μ (AMS4_{FAE}) ≠ μ (AMS5_{FAE})), are shown in Table 6-21 where it can be seen that the F statistic (2.74) exceeds the F critical value (2.58) at the statistically significant level of 0.05 (p-value = 0.04). This suggests that the population means of the five groups sampled are not equal. Moreover, the Kruskal-Wallis test was conducted, the results of which also shows a significant variance between the groups of FAE values collected for all experimental services (the Kruskal-Wallis statistic (H) = 9.75, with p = 0.044 (corrected for ties)). Therefore, we accept the alternative hypotheses H_{coh1}, and can conclude that TICS can be used to indicate the analysability sub-characteristic of maintainability of software services.

Note that the Fisher’s Least-Significant Difference (LSD) mean comparison test was not applied as part of the ANOVA tests for the cohesion hypotheses H_{coh1} and H_{coh2} given that a more powerful correlation and regression analysis was conducted in Section 6.4.3.

6.4.2.2 Hypothesis H_{coh2}: Cohesion Indication

The one-way ANOVA test indicated that there is a statistically significant relationship between the Total Cohesion of a Service (TICS) metric (and thus, its constituent metrics SIDC, SIUC, SIIC, and SISC) and the subjective ranks (CR) of the conceptual cohesiveness of services. This suggests that the derived cohesion metrics can be used to support the automated allocation of software services to the inherently semantic Coincidental and Conceptual categories of service cohesion.
CHAPTER 6. EMPIRICAL EVALUATION OF METRICS

<table>
<thead>
<tr>
<th>SOURCE OF VARIATION</th>
<th>SUM SQUARES</th>
<th>MEAN SQUARE</th>
<th>F STATISTIC</th>
<th>F CRITICAL</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Groups (between)</td>
<td>85.4</td>
<td>21.35</td>
<td>45.53</td>
<td>2.58</td>
<td>&lt;0.0001</td>
</tr>
<tr>
<td>Residual (within groups)</td>
<td>21.1</td>
<td>0.47</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>106.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.22. one-way ANOVA results for the Hypothesis $H_{coh2}$ (n=50)

The statistics related to the ANOVA conducted to test the inequality of variance in the groups of CR values for five experimental cohesion services ($\mu (AMS1_{CR}) \neq \mu (AMS2_{CR}) \neq \mu (AMS3_{CR}) \neq \mu (AMS4_{CR}) \neq \mu (AMS5_{CR})$) are shown in Table 6.22, where it can be seen that the F statistic (45.53) exceeds the F critical value (2.58) at the statistically significant level of 0.05 ($p$-value = <0.0001). This suggests that the population means of the five groups sampled are not equal. Moreover, the results of the Kruskal-Wallis test also show a significant variance between the groups of CR values ($H = 39.22$, with $p = <0.0001$ (corrected for ties)). Therefore, we accept the alternative hypotheses $H_{coh2}$, and can conclude that $TICS$ can be used to indicate the conceptual cohesiveness of services in SO systems.

6.4.3 Linear Regression Analysis

To further evaluate the experimental hypotheses $H_{coh1}$ and $H_{coh2}$, and to examine the direction and strength of the linear relationships between the values of TICS, and the values of Failure Analysis Efficiency (FAE) and Cohesion Rank (CR) measures, a univariate linear regression analysis [61] was conducted. Such analysis allows finding the best line that predicts one dependent variable (FAE or CR values in our case) from one independent variable (TICS) by minimising the sum of the square of the vertical distances of the points from the regression line. The following statistics are produced by the linear regression analysis:

- The coefficient of determination ($R^2$): is a measure of goodness-of-fit of linear regression. The values of $R^2$ range from 0 to 1, with the value 0 indicating the lack of fit (that is, the independent variable $x$ cannot be used to predict the dependent variable $y$); and the value 1 suggesting the perfect fit meaning that knowing $x$ lets us perfectly predict $y$.

- Slope ($\beta_1$) and intercept ($\beta_0$): The slope represents the regression coefficient which reflects the steepness of the regression line. More specifically, it shows the amount of increase of $y$ when $x$ is increased by one unit. If the slope is positive, $y$ increases as $x$ increases. If the slope is negative, $y$ decreases as $x$ increases. The intercept represents the line point on the y-axis when the position of the x-axis is zero (it defines the elevation of the line). Note that the significance (probability $p$) of $\beta$ coefficients must also be determined.

(February, 2009)
The obtained linear regression results\(^\text{34}\) (described below) indicate that:
- the behaviour of participants analysing the experimental services (as measured by FAE) has changed in a systematic linear manner with the controlled increase of the level of treatment (different levels of TICS). This suggests that the derived cohesion metrics can be used as early predictors of analysability of service-oriented software.
- the participants’ perception of service cohesiveness (as measured by CR) has changed in a systematic linear manner with the controlled increase of the level of treatment (different levels of TICS). This suggests that the derived cohesion metrics can be used to predict the conceptual cohesiveness of services in service-oriented software.

\(^{34}\)The linear regression analyses were conducted under the 95% Confidence Interval.
### Table 6-23. Regression statistics for TICS and FAE values (n = 5)

<table>
<thead>
<tr>
<th>$\beta_0$ (intercept) coefficient</th>
<th>$\beta_0$ (intercept) $p$ value</th>
<th>$\beta_1$ (slope) coefficient</th>
<th>$\beta_1$ (slope) $p$ value</th>
<th>Standard Error (SE)</th>
<th>$R^2$</th>
<th>Adjusted $R^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.814</td>
<td>&lt;-0.0001</td>
<td>-2.016</td>
<td>0.0033</td>
<td>0.186</td>
<td>0.96</td>
<td>0.95</td>
</tr>
</tbody>
</table>

Table 6-23. Regression statistics for TICS and FAE values (n = 5)

The statistics related to the regression analysis are shown in Table 6-23. The high value of goodness-of-fit of linear regression ($R^2 = 0.96$) and the large regression coefficient ($\beta_1 = -2.016$) with the statistical significance $p = 0.0033$, provide statistical evidence that the values of TICS can be used to predict the values of FAE. This in turn, provides an additional basis for accepting the experimental hypothesis $H_{coh1}$.

**6.4.3.2 Hypothesis $H_{coh2}$: Cohesion Indication**

The graphical representation of the regression results related to the ability of TICS to predict the conceptual cohesiveness of services (CR values) is shown in Figure 6-5, where it can be observed that the trendline of the plotted experimental data closely matches the linear regression trendline, as was the case with the FAE-related results. Furthermore, the actual values of CR also closely match the expected values (the five CR levels were expected to directly correlate to five levels of TICS as explained in Section 6.2.3.2).
### Table 6-24. Regression statistics for TICS and CR values (n = 5)

<table>
<thead>
<tr>
<th>β₀ (intercept) coefficient</th>
<th>β₀ (intercept) p value</th>
<th>β₁ (slope) coefficient</th>
<th>β₁ (slope) p value</th>
<th>Standard Error (SE)</th>
<th>R²</th>
<th>Adjusted R²</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.26</td>
<td>0.002</td>
<td>3.68</td>
<td>0.0004</td>
<td>0.16</td>
<td>0.991</td>
<td>0.99</td>
</tr>
</tbody>
</table>

The statistics related to the regression analysis are shown in Table 6-24. The high value of goodness-of-fit of linear regression (R² = 0.991) and the large regression coefficient (β₁ = 3.68) with the statistical significance p = 0.0004, provide statistical evidence that the values of TICS can be used to predict the values of CR. This in turn, provides an additional basis for accepting the experimental hypothesis H₁coh2.

### 6.5 Summary

This study investigated the impact of structural properties (coupling and cohesion) of SO software, measured by the metrics derived in this research, on the specific sub-characteristics of software maintainability (namely analysability, changeability, and stability). Overall, we can conclude that:

i) there is empirical evidence of the relationship between the investigated service-oriented cohesion metrics, and the dependent variables used in the study;

ii) there is empirical evidence that the highly-coupled elements have a negative influence on the changeability and analysability of SO software compared to the lowly-coupled elements;

iii) extra-service incoming and outgoing coupling relationships have a greater impact on the changeability sub-characteristic of SO software compared to intra-service relationships, but the results obtained when comparing the impact of extra- and intra-service relationships on the analysability of software did not indicate any statistically significant differences.

iv) the results obtained when comparing the impact of indirect and direct extra-service incoming and outgoing relationships did not indicate any statistically significant differences. Nevertheless, we believe that performing system changes of a structural nature (e.g. merging or removing services) could result in different statistical outcomes since it would be harder to perform structural changes on a system containing elements coupled via direct extra-service relationships. Moreover, the reusability of services containing direct extra-service relationships could also be decreased. To this end, additional studies could be conducted in future work to further evaluate the associated hypotheses.

Note that although this study is exploratory in nature, insofar as the size of the experimental systems was small and the number of participants low, it can be considered as a controlled
experiment since: i) the independent variables were controlled, and assigned to the service-oriented design artefacts based on specific criteria; ii) potential influencing factors (controlled variables) were kept as constant as possible; and iii) the experiments were conducted in a controlled environment. This in turn, allows for a stronger case to be made for causation between the independent and dependent variables. Nevertheless, we still consider this study as explorative, and as such, all significant relationships presented in Sections 6.3 and 6.4 and summarised above should be accepted as preliminary findings. To this end, this study should be replicated and extended in future work in order to confirm the validity of the experimental results, thereby establishing that the derived metrics could be used as early indicators and predictors of SO software maintainability in an industrial setting.

A summarised view of the results is shown in Table 6-25, where all evaluated metrics are mapped to the experimental hypotheses and associated maintainability sub-characteristics. Finally, the measures taken to address the possible threats to validity of this study are discussed in Section 6.5.1, and a detailed outline of future work directions related to the empirical evaluation of metrics is presented in Chapter 7.

6.5.1 Threats to Validity

This section summarises measures taken to alleviate any potential threats to validity of the experimental results. The threats are described according to the classification of Wohlin et al. [239], which prescribes three main categories of experimental validity threats: i) construct validity; ii) internal validity; and iii) external validity.

6.5.1.1 Construct Validity

Construct validity refers to the degree to which the independent and dependent variables accurately measure the concepts they purport to measure.

- The derived service-oriented coupling and cohesion metrics (independent variables) were defined in a formal and precise manner and also validated theoretically. This gives us confidence that the independent variables satisfy construct validity.
- The dependent variables used to measure the different sub-characteristics of maintainability were taken from the suite of standard ISO/IEE metrics, and thus, can be considered as constructively valid measures of maintainability.

6.5.1.2 Internal Validity

Internal validity refers to the degree to which conclusions can be drawn about the causal effect of independent variables on dependent variables. The following types of internal validity threats have been identified by other researchers performing similar studies in the software metrics area [29, 36, 53, 86]:

(February, 2009)
## Chapter 6. Empirical Evaluation of Metrics

<table>
<thead>
<tr>
<th>METRIC/S NAME</th>
<th>TESTED HYPOTHESIS</th>
<th>TEST CONDITION</th>
<th>ANALYSABILITY (FAE)</th>
<th>CHANGEABILITY (MC)</th>
<th>STABILITY (MIL)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INVESTIGATED COUPLING METRICS</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>WISCE</strong></td>
<td>$H_{coup1.1}$ and $H_{coup1.2}$</td>
<td>low vs. high intra-service incoming and outgoing coupling</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td><strong>ESICSI</strong></td>
<td>$H_{coup1.3}$</td>
<td>low vs. high indirect extra-service incoming coupling</td>
<td>NR</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td><strong>EESIOC</strong></td>
<td>$H_{coup1.4}$</td>
<td>low vs. high indirect extra-service outgoing coupling</td>
<td>✓</td>
<td>✓</td>
<td>NR</td>
</tr>
<tr>
<td><strong>WESICE</strong></td>
<td>$H_{coup1.5}$</td>
<td>low vs. high direct extra-service incoming coupling</td>
<td>NR</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td><strong>WESOCE</strong></td>
<td>$H_{coup1.6}$</td>
<td>low vs. high direct extra-service outgoing coupling</td>
<td>✓</td>
<td>✓</td>
<td>NR</td>
</tr>
<tr>
<td><strong>WISCE, ESICSI</strong></td>
<td>$H_{coup2.1}$</td>
<td>high intra-service vs. high indirect extra-service incoming coupling</td>
<td>NR</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td><strong>WISCE, EESIOC</strong></td>
<td>$H_{coup2.2}$</td>
<td>high intra-service vs. high indirect extra-service outgoing coupling</td>
<td>×</td>
<td>✓</td>
<td>NR</td>
</tr>
<tr>
<td><strong>WISCE, WESICE</strong></td>
<td>$H_{coup2.3}$</td>
<td>high intra-service vs. high direct extra-service incoming coupling</td>
<td>NR</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td><strong>WISCE, WESOCE</strong></td>
<td>$H_{coup2.4}$</td>
<td>high intra-service vs. high indirect extra-service outgoing coupling</td>
<td>×</td>
<td>✓</td>
<td>NR</td>
</tr>
<tr>
<td><strong>ESICSI, WESICE</strong></td>
<td>$H_{coup2.5}$</td>
<td>high indirect extra-service incoming coupling vs. high direct extra-service incoming coupling</td>
<td>NR</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td><strong>EESIOC, WESOCE</strong></td>
<td>$H_{coup2.6}$</td>
<td>high indirect extra-service outgoing coupling vs. high direct extra-service outgoing coupling</td>
<td>×</td>
<td>×</td>
<td>NR</td>
</tr>
</tbody>
</table>

| **INVESTIGATED COHESION METRICS** | | | | | |
| **TICS (and [SIDC, SIUC, SIIC, SISC])** | $H_{coh1}$ | - testing five different levels of service cohesion | ✓ | NR | NR |

(Febuary, 2009)
CHAPTER 6. EMPIRICAL EVALUATION OF METRICS

<table>
<thead>
<tr>
<th>TICS (and [SIDC, SIUC, SIIC, SISC])</th>
<th>$H_{coh2}$ - testing five different levels of service cohesion in terms of the Cohesion Rank (CR)</th>
<th>NR</th>
<th>NR</th>
<th>NR</th>
</tr>
</thead>
</table>

Table 6-25. Summary of the empirical evaluation results (Legend: statistically significant (✓), not significant (✘), not related (NR))

i) Differences among participants.
- The study used a within-subjects design; therefore error variance due to differences among participants was reduced (refer to Section 6.2.7).
- Purposive sampling [217] was employed to select participants with comparable knowledge and experience in the areas related to the study (refer to Section 6.2.2).

ii) Practice [learning and fatigue] effects (where independent variables can become confounded with the order of presentation, refer to Section 6.2.7).
- The random starting order with rotation [217] technique was used to assign the participants to the experimental systems in a systematic counterbalancing manner, thereby reducing potential learning and fatigue effects (refer to Section 6.2.7.1).
- The experiments were conducted in two separate experimental runs in order to further minimise fatigue effects.

iii) Instrumentation effects (where independent variables can be affected due to the differences in the experimental material, refer Section 6.1).
- Both experimental systems, and their associated sub-systems and services, were designed for the same universe of discourse (Academic Management System); therefore, the knowledge of the application domain did not influence internal validity.
- An effort was made to only manipulate the structural properties of interest (coupling or cohesion), while keeping the other structural properties (controlled variables) as constant as possible in order to prevent their influence on the experimental results (refer to Section 6.2.5).
- A pilot study was conducted, using two experienced software engineers, to ensure that the experimental tasks were comparable in terms of their conceptual complexity. The software engineers subjectively confirmed that the tasks were conceptually and structurally comparable (refer to Section 6.2.8.1).

iv) Anticipation effects.
- To ensure that the participants’ expectations about which condition would occur next do not influence their responses, the participants were not told about the expected study outcomes or the structural specifics (individual levels of coupling or cohesion) of experimental software services.
CHAPTER 6. EMPIRICAL EVALUATION OF METRICS

6.5.1.3 External Validity

External validity refers to the degree to which the results of the research can be generalised to the wider population, and associated environment and settings (i.e. applied software engineering practice).

i) Experimental materials and tasks.

   - This particular threat was not addressed, and thus, constitutes the greatest limitation of this study. This is because, the experimental systems may not be representative of industrial software products in terms of their size. Also, the complexity of the systems and associated experimental tasks was low. As such, unforeseen circumstances may arise in more complex systems, or when more complex tasks are performed. Nevertheless, we believe that the findings are valuable and can provide a basis for future empirical investigation of the structural properties of SO software.

ii) Participants.

   - Experienced software engineers were recruited to participate in the study. That is, all participants had prior development skills and experience. Therefore, our sample is likely to be representative of the overall population of software practitioners. Note however, that the participants were experienced with OO development, but had more limited exposure to service-oriented development, and as such, different results could be obtained if participants had more substantial experience with SOC.

iii) Environment.

   - A widely used IDE (Eclipse 3.3), modelling language (UML), and programming language (Java) were employed in the study. These technologies are used frequently in the software development industry, and as such, can be considered as representative of an industrial environment.
Chapter 7. Summary and Conclusion

This concluding chapter consists of three sections. Section 7.1 summarises the findings of this thesis; Section 7.2 discusses the practical applicability of the derived service-oriented coupling and cohesion metrics; and Section 7.3 outlines areas of further research.

Note that the core contribution chapters (Chapters 3-5) have already provided a descriptive summary of the main findings, as well as future work specific to particular chapters. As such, this concluding chapter provides a general summary, whilst referring the reader to specific chapters for a more detailed description.

7.1 Summary

The primary goal of this research was to derive, theoretically validate, and empirically evaluate, a suite of service-oriented coupling and cohesion design-level metrics that can be used early in the Software Development LifeCycle (SDLC) in order to predict the maintainability of SO software products. This goal has been achieved as summarised below:

- A detailed critical review of the existing work in the areas of software metrics, software maintainability, and SOC was conducted in order to gain knowledge and expertise required to derive a suite of valid software metrics (Chapter 2). Additionally, a number of face-to-face and correspondence-based discussions with experts in the area (Appendix A) were held to further improve the required knowledge.

- An initial case-study was conducted in order to: empirically determine whether some of the widely-used Procedural and OO metrics can be used to accurately measure the structure of service-oriented designs; and gain additional knowledge needed to derive the metrics. The study, which is described in Section 2.5.5, demonstrated that the metrics under investigation cannot quantitatively distinguish between service-oriented (SO) designs that were qualitatively different.

- A formal model of service-oriented software design was derived in order to support the definition of metrics using rigorous and precise mathematical notations and techniques (Chapter 3). More specifically, the model provided a mechanism for defining metrics in an unambiguous and formal manner, thereby ensuring that the metrics accurately represent the entities and attributes they purport to quantify. Note that the model itself can be considered as contribution in itself since it is generic and can be used in future research related to the structural properties of SO software designs (Section 7.3.1).
- Metrics for measuring the structural properties of coupling and cohesion in service-oriented designs have been defined in a precise and unambiguous manner using the formalism captured by the model of service-oriented software design. Additionally, coupling and cohesion assumptions were given regarding explicit links between the metrics and the sub-characteristics of maintainability (analysability, changeability, and stability) of service-oriented software.

- Specifically, the following coupling and cohesion metrics were introduced:

  - Sixteen coupling metrics (presented in Chapter 4) intended to measure the static and dynamic aspects of coupling in service-oriented designs. The metrics cover service-oriented specific and common relationships (Section 4.2) between different service-oriented design entities as captured by a model of service-oriented system design. Additionally, a SO-specific type of coupling, service autonomy, was introduced in order to investigate the conformance of a given system design to the fundamental principles of service-orientation.

  - Five cohesion metrics (presented in Chapter 5) intended to measure the cohesion of services in service-oriented designs, based on the conceptual relatedness of operations exposed in a service interface. Such conceptual relatedness is difficult to quantify without examining the semantics of a service. Therefore, to provide a conceptual framework for the derivation of automated and objective cohesion metrics, seven qualitative categories of service cohesion were defined and classified into purely semantic and quantifiable types (Section 5.2). The derived cohesion metrics measure the quantifiable cohesion categories and also estimate the purely semantic Coincidental and Conceptual categories of cohesion according to a number of cohesion suppositions (Section 5.2.1).

  - The metrics were theoretically validated using the property-based software engineering measurement framework of Briand et al. [30] (described in Section 2.5.2). The theoretical validation demonstrated the mathematical soundness of the metrics. More specifically, the derived metrics satisfied the required mathematical characteristics of coupling and cohesion [30], and as such, are considered as theoretically-valid measures of coupling and cohesion.

  - The metrics were evaluated empirically in order to statistically test the correlation between the derived measures of coupling and cohesion, and the maintainability of service-oriented software products (Chapter 6). An empirical evaluation involved a group of ten participants who performed a number of maintenance activities on two controlled SO software systems developed for this study. The majority of the observed factors were in line with our expectations and assumptions, although some of the empirical results were not statistically significant as described in Section 6.5. Consequently, it is understood that further demonstrating the empirical validity of the proposed metrics requires additional larger-scale empirical
CHAPTER 7. SUMMARY AND CONCLUSION

studies. Such large-scale evaluation was considered beyond the scope of this thesis but could be conducted in future work as discussed further in Section 7.3.2.

- The practical applicability of metrics is discussed in the following section, where the derived metrics are shown to fulfil the following pragmatic properties: i) the metrics are technology independent; ii) the metrics can be collected in an automated manner using a dedicated software tool; and iii) the metrics can provide support for the service-oriented analysis and design development phases.

From the above summary, it is evident that the metrics derivation, validation, and evaluation process was conducted in a systematic methodological manner, and thus the goal of this thesis was achieved. Furthermore, the findings presented in this thesis have been introduced to the research community via a number of refereed publications [189-195], which provides additional indication of the soundness and validity of the obtained results.

7.2 Practical Applicability

This section discusses issues related to the practical applicability of the derived coupling and cohesion metrics. The practical applicability is important since it was suggested previously [38, 76] that: i) software metrics are most useful in practice if they can provide support for decision making during the software development process [98]; and ii) software managers and engineers will be reluctant to use metrics if they are difficult (or labour-intensive) to collect, and as such, it is desirable to demonstrate that it is possible to automate the collection of metrics. To this end, the following sub-section describes how the derived metrics can be used within the SO design process, and Section 7.3.3 provides an initial outline of the tool support that could be developed in future work to assist in the automated collection of metrics.

Finally note that another important practical requirement for software metrics is that they should be platform and language independent; otherwise, the metrics will have a limited scope of use, and comparison across products developed using different technologies will be difficult. This requirement is satisfied because the metrics were defined using a generic model of SO software design which does not enforce any particular software implementation (the model was designed to be as generic and technology agnostic as possible as described in Section 3.5).

7.2.1 Metrics Application in the SO Design Process

One of the main reasons for investigating and quantifying the structural properties of software is to evaluate the quality of design structures, thereby supporting the development of

---

Note that developing a metric collection tool was considered to be outside of the research scope and is part of future work.
software products that can be easily maintained [55]. To this end, the proposed metrics can provide the following general benefits: i) facilitate design decisions leading to the specification of quality SO designs; ii) identify problems in existing service-oriented design structures; and iii) allow for more effective control of maintainability in the earlier stages of SDLC. The specific benefits of the proposed coupling and cohesion metrics are described below:

**Coupling Metrics.**
- The metrics can be used to detect high levels of intra- and extra-service coupling, thereby suggesting a need to restructure a system;
- Specific design guidelines can be formulated in terms of concrete metric values. For instance, minimum and maximum recommended degrees of coupling for the most important services in the system can be suggested based on empirical/historical data (to be collected as part of future work);
- The metrics can be used to evaluate the conformance of a given system design to the core principles of SOC in terms of the level of system partitioning and system purity (SPARF and SPURF metrics described in Section 4.3). More specifically, by quantifying the number of implementation elements shared between system services, and also measuring the direct extra-service coupling, the metrics can provide support for determining the degree of service-autonomy (if possible, all services should be independent from one another).

**Cohesion Metrics.**
The derived cohesion metrics can provide support for the systematic identification of service interfaces, which is a primary activity in the SO analysis and design phases [12, 67, 181, 222]. At present, service granularity is the only structural aspect of service-interfaces that is investigated in sufficient detail in the research and industry literature, where services are typically categorised into fine- and coarse-grained types based on the amount of functionality exposed in the service interfaces [174, 184]. We believe that service cohesiveness (reflected by the relatedness of the operations exposed in the service interface) should also be considered as an important structural aspect given its influence on the analysability of a service as shown in Chapter 6. More specifically, it is not sufficient to evaluate the granularity of a service in isolation, the cohesiveness of the provided functionality should be taken into consideration when analysing and designing SO solutions. For example it is expected that a Coincidental coarse-grained service will have a negative impact on the quality of software systems compared to a Conceptual coarse-grained service.

---

36 Although, defining service interfaces is still considered to be a conceptually difficult task and there are no agreed criteria for determining the right granularity of services [68]
CHAPTER 7. SUMMARY AND CONCLUSION

7.3 Future Work

This section is divided into three subsections. Section 7.3.1 describes the possible extensions to the coupling and cohesion metrics derived in this research and also discusses some opportunities for the derivation of additional metrics. Section 7.3.2 proposes a number of experiments for expanding upon the results of Chapter 6. Finally, Section 7.3.3 presents an outline of the tool support for metrics collection designed to increase the practicality of metrics (as described in the previous section).

7.3.1 Metrics

There are a number of possible future directions related to the service-oriented software metrics as follows:

- Extending the derived coupling metrics to consider additional dimensions of coupling that could potentially influence the maintainability of SO software. For example, the following coupling-related factors could be investigated in future work: i) the frequency of relationships between two elements; ii) the size and semantics of the data passed between design elements; and iii) the type (control or data) of information flow.

- Deriving metrics for measuring the structural properties of size and complexity. Note that given that the design-level complexity can be considered as an amalgamation of coupling and cohesion (refer to Section 2.4.1), the service-level coupling and cohesion metrics derived in this research could be redefined and combined together in order to establish an integrated Service Maintainability Index, which can be used to quantify the overall complexity of service-oriented design constructs.

- Investigating additional quality characteristics as defined in the ISO/IEC TR 9126 standards [112, 113]. For example, based on the results reported for the empirical studies in the OO paradigm, it can be expected that the structural properties of coupling and cohesion also have an effect on the reliability [10, 244] and efficiency (performance) [207, 228] of software products. Investigating other quality characteristics might require updating and refining the coupling and cohesion metrics derived in this research in order to allow for more accurate prediction of the particular quality characteristic under investigation. For example, when designing for the maintainability of SO software, relationships that undermine the rules of service autonomy should be avoided, whereas when estimating the performance quality characteristic, the service interface-related relationships will have greater significance due to the extra processing introduced by XML marshalling when communicating via WSDL-based service interfaces [4].

199

(Feb
March 2009)
7.3.2 Empirical Studies

The experimental study described in this thesis could be replicated and extended in order to perform more comprehensive evaluation of the derived coupling and cohesion metrics. More specifically, a complete family of experiments that captures multiple similar studies pursing the same goal can be defined according to the methodology of Ciolkowski et al. [47] along the following four dimensions:

1. Nature of the study and experimental tasks:
   Additional controlled studies should be conducted in order to replicate the overall experimental design and tasks. Such studies should employ the same experimental material and statistical tests as the study presented in this thesis.

   - Large-scale industrial studies should be conducted in order to increase the external validity of the results (refer to Section 6.5.1.3). Such studies are expected to produce a greater range of values for the independent and dependent variables, thus supporting more complete and statistically sound evaluation of metrics and associated preliminary weights.

   - Different types of maintenance changes could be investigated. More specifically, the study conducted in this research only evaluated functional changes, whereas future work could also investigate the impact of structural changes (such as merging or removing system services) on the maintainability of SO software.

2. Technology:

   - The experimental systems were implemented using Java EE 5 framework with the business logic and rules being encapsulated in stateless Session Beans. It would be advisable to use existing business process orchestration languages (such as WS-BPEL) to encapsulate compositional and business logic in dedicated business process scripts in order to see whether this will have any effect on system maintainability.

   - Different implementation paradigms/languages (e.g. scripting languages such as PHP, and procedural languages such as C) could be used to implement the experimental services. This will allow evaluating the proposed coupling weights given that it was suggested in Section 4.2 that communication between elements belonging to different development paradigms could require extra development and maintenance efforts.

3. Target systems:

   - Experimental software systems should include implementation elements with a greater variance of coupling in order to support more comprehensive evaluation of the coupling metrics. That is, for each investigated type of coupling, the number of relationships
could be varied across a number of linearly increasing levels (as opposed to the two arbitrarily chosen [low/high] levels used in the present study). This in turn, will allow: i) re-evaluation of the coupling-related findings presented in this thesis, and establishing preliminary weights for the different types of coupling relationships; ii) evaluating the predictive capability of the coupling metrics using linear regression analysis; and iii) evaluating coupling metrics not investigated in the present study (refer to Section 6.2.4).

- Experimental software systems should include services with a greater variance of cohesion in order to allow the investigation of the individual analysability impact of the constituent metrics of Total Interface Cohesion of a Service (TICS). That is, each constituent metric of TICS [SIDC, SIUC, SIIC, and SISC] could be manipulated in isolation across a number of different levels of cohesion, while keeping the other three constituent metrics at constant levels. This will allow investigating the relative strength of the particular cohesion type and establishing preliminary weights for the different types of service-oriented cohesion in order to derive a more accurate (weighted) metric of total interface cohesion of a service (TICS).

- Large-scale, operational software product/s should be used as the experimental material when conducting industrial studies, with the data required to calculate the dependent variables (ISO/IEC metrics) being extracted from the associated maintenance–related documentation collected over a longer period of time (for example, one year). This in turn, will increase the power of statistical analysis applied as part of the metrics evaluation process.

4. Participants:

- A larger sample size (number of participants) should be used in future experiments in order to increase the internal validity of the experimental results (refer to Section 6.5.1.2).

- Participants possessing substantial skills and experience with SOC should be employed in future experiments. The participants used in the study presented in this thesis were experienced with OO development, but had more limited exposure to service-oriented development.

- Participants with varying industrial software development and maintenance experience profiles should be employed in future experiments. Specifically, the development and maintenance experience of participants used in the study presented in this thesis ranged from six months to five years. It would be desirable to conduct experimental studies targeting participants with more specific and constrained experience profiles (for example, novice software engineers with experience ranging from 1 to 12 months; and expert software engineers with experience ranging from 10 to 20 years).
CHAPTER 7. SUMMARY AND CONCLUSION

Finally note that different dependent variables (maintainability metrics) could be used in future work. For example, ISO/IEC had recently proposed a new set of international standards for software product quality requirements and evaluation (SQuaRE) [115]. The goal of SQuaRE is to provide a complete and comprehensive framework for covering two major quality processes: software quality requirements specification and software quality evaluation, supported by a well-defined software quality measurement process. Specifically, SQuaRE incorporates five quality divisions: i) quality management; ii) quality model; iii) quality measurement; iv) quality requirements; and v) quality evaluation. The divisions of quality model and measurement are currently based on the ISO/IEC9126:1-4 [111-114] suite of standards covered in this thesis, and as such, the metrics used in the empirical study presented in Chapter 6 can still be considered relevant and up-to-date. Note however that the description of SQuaRE provided by ISO/IEC acknowledges some inconsistencies in ISO/IEC9126 models [115] and it is possible that new quality metrics will be incorporated into SQuaRE in a near future. To this end, these updated metrics could be suitable for use in the future experiments described in this section.

7.3.3 Tool Support for Metrics Collection

Technological support is important for the effective use of software measures [75]. To support the collection of metric values in large industrial software products, there is a need to automate the metric collection process. Lavazza [136] defined three major advantages of automated metrics collection: i) reducing the effort required to collect the metrics; ii) minimizing errors in calculating metric values; and iii) focusing on the analysis of the results, rather than on data collection.

Although developing tool support for the derived coupling and cohesion metrics is beyond the scope of this research, the following recommendations should be considered when developing software tool support in future work:

- The metric collection functionality should be incorporated into a CASE tool supporting SOC. To this end, the formal model shown in Chapter 3 can provide support for the development of such a CASE tool given that the constructs and formal notation captured by the model can be easily represented using graphical notation (the specification of which is part of future work) developed specifically for SO software designs. Such notation can be based on existing notations, for example UML, with extensions for SOC-specific constructs. Additionally, the model can provide support for automated design consistency checks and metrics collection.

- The metrics collection functionality should provide support for:
i) Transforming the input (for example SO design diagrams) into the internal representation of the design abstractions captured by the formal model;

ii) Computing the metric values from the internal representation of design abstractions;

iii) Producing prediction estimates for the maintainability sub-characteristics using the collected metric values. Note that producing such estimates is possible given the predictive power of some of the derived metrics\(^\text{37}\).

iv) Producing output of the results in different formats that can be easily understood by software engineers.

\(^{37}\) In this thesis only the predictive power of the cohesion metrics was evaluated, future work should also evaluate the predictive power of the coupling metrics.
Software Design Metrics for Predicting Maintainability of Service-Oriented Software

By Mikhail Perepletchikov

APPENDICES
Appendix A. Literature Review Resources

The literature search/review process was conducted in a systematic manner following the guidelines of Kitchenham et al. [127]. The scope of the literature search was constrained by the research topics discussed in Section 2.1, where a given source was included in the review only if it provided some direct evidence or knowledge related to the Research Questions defined in Chapter 1. The overall search strategy was based on two main search activities: i) identification of key research and industry resources including: journals, conferences, standards organisations, and technical reports repositories, for each major research area of interest; and ii) identification and selection of the seminal papers, books and industry reports based on consultations with internal and external [60, 99, 141, 182, 206, 248] experts. Table A1 lists major source types and associated resources covered in the review.

<table>
<thead>
<tr>
<th>SOURCE TYPE</th>
<th>SPECIFIC RESOURCES</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Electronic databases / Digital libraries</strong></td>
<td>IEEEExplore (ieeexplore.ieee.org); ACM digital library (portal.acm.org); Springer Link (springerlink.com); Science Direct (sciencedirect.com); DBLP Computer Science Bibliography (dblp.uni-trier.de); Google Scholar (scholar.google.com); CiteSeer (citeseer.ist.psu.edu).</td>
</tr>
<tr>
<td><strong>Journals</strong></td>
<td>IEEE Transactions on Software Engineering; IEEE Computer; IEEE Internet Computing; IEEE Software; Software Quality Journal (Springer); Empirical Software Engineering (Springer); Communications of the ACM; Journal of Information and Software Technology (Elsevier); Journal of Systems and Software (Elsevier).</td>
</tr>
<tr>
<td><strong>Conference proceedings</strong></td>
<td>International Conference on Software Engineering (ICSE); International Symposium on the Foundations of Software Engineering (FSE); International Conference on Service Oriented Computing (ICSOC); International Conference on Software Maintenance (ICSM); International Symposium on Software Metrics (METRICS); International Symposium on Empirical Software Engineering and Measurement (ESEM).</td>
</tr>
</tbody>
</table>
### Other resources

- IBM Redbooks (www.redbooks.ibm.com/);
- Microsoft MSDN Magazine (msdn.microsoft.com/magazine/);
- Object Management Group (www.omg.org/);
- ISO/IEC standards (www.standardsinfo.net/);

### Personal contacts

<table>
<thead>
<tr>
<th>Name</th>
<th>Organisation/Position</th>
<th>Contact Details</th>
<th>Year of Contact</th>
<th>Area/Purpose of Contact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mike P. Papazoglou</td>
<td>Tilburg University, The Netherlands</td>
<td><a href="mailto:mikep@uvt.nl">mikep@uvt.nl</a></td>
<td>2005-2007</td>
<td>SOC</td>
</tr>
<tr>
<td>Olaf Zimmermann</td>
<td>IBM Zurich Research Lab, Switzerland</td>
<td><a href="mailto:olz@zurich.ibm.com">olz@zurich.ibm.com</a></td>
<td>2005/2006</td>
<td>SOC</td>
</tr>
<tr>
<td>Grace A. Lewis</td>
<td>Carnegie Mellon University, USA</td>
<td><a href="mailto:glewis@sei.cmu.edu">glewis@sei.cmu.edu</a></td>
<td>2005</td>
<td>Maintainability, SOC</td>
</tr>
<tr>
<td>Michael Rosemann</td>
<td>Queensland University of Technology, Australia</td>
<td><a href="mailto:m.rosemann@qut.edu.au">m.rosemann@qut.edu.au</a></td>
<td>2005</td>
<td>SOC, BPM</td>
</tr>
<tr>
<td>Brian Henderson-Sellers</td>
<td>University of Technology Sydney, Australia</td>
<td><a href="mailto:brian@it.uts.edu.au">brian@it.uts.edu.au</a></td>
<td>2005</td>
<td>Software metrics</td>
</tr>
</tbody>
</table>

Table A1. Literature review – main resources
Appendix B. Measurement Theory Concepts and Application

- **Empirical relational system** (E) is a tuple \( E = (A, R_1, ..., R_n, o_1, ..., o_n) \), where \( A \) is a set of entities (or empirical objects); \( R_1, ..., R_n \) is a set of empirical relations defined on \( A \); and \( o_1, ..., o_n \) are binary operations defined on \( A \).

The empirical relational system is a model of the part of the problem domain representing the perspective on common knowledge about the phenomenon to be measured. The model should ensure agreement about the empirical relations with respect to an attribute in question. For example, in order to use the empirical relationship (R) “service S1 is more cohesive than service S2”, there is a need to clearly define the concept of a service, and describe the intuitive understanding of service cohesion. An empirical relation system does not contain any reference to measures, and as such, it needs to be mapped to a formal relational system.

- **Formal relational system** (F) is a tuple \( F = (B, R_1', ..., R_n', o_1', ..., o_n') \) where \( B \) is a set of formal objects (numbers); \( R_1', ..., R_n' \) is a set of formal (numerical) relations defined on \( B \); and \( o_1', ..., o_n' \) are closed binary operations defined on \( B \) (such as the addition).

The formal relational system formalises our intuitive understanding of the relationships between attributes in a precise mathematical way. For instance, the empirical relation (R) “service S1 is more cohesive than service S2” can be redefined as a formal relationship (R’) “cohesion value (S1) > cohesion value (S2)”. To establish a complete and comprehensive formal relationship system, it is advisable (but not strictly necessary) to formalise an intuitive model of the problem domain, such as service-oriented design, in order to establish an objective and formal framework for reasoning about attributes.

- **Measure** (\( \mu \)) represents a mapping from the empirical to formal systems and can be defined as \( \mu : E \rightarrow F \), which yields for every empirical object \( a \in A \), a formal object (or measurement value) \( b \in B \). That is, \( b = \mu(a) \).

A theoretically valid measure (or metric) requires equivalence between the empirical and formal systems. That is, the formal relations must preserve the meaning of the empirical statements. For example, given an empirical relation “more cohesive than”, a formal relation “>”, and a cohesion metric \( \mu \), the assertion that “S1 is more cohesive than service S2” will be true if and only if \( \mu(S1) > \mu(S2) \). This mapping from one relational system to another that preserves all relations and operations is referred to as a homomorphism. The homomorphism
APPENDIX B. MEASUREMENT THEORY CONCEPTS AND APPLICATION

covers all possible admissible transformations on \( E \) and \( F \), which in turn enables a classification of measurement scales.

Table B1 illustrates the example application of measurement theory to the measurement of the height of humans (physical entities). In this example, the empirical relation \( R_1 \) (human \( a \) is taller than human \( a' \)) is mapped to the formal relation \( R_1' \) (real number \( b \) > real number \( b' \)) via a metric \( \mu \), thereby supporting the explicit understanding of the height of humans, and suggesting that bigger numbers or categories must be assigned by \( \mu \) to the taller humans so as to be consistent with the empirical model.

Based on the intuition captured in \( R_1 \), the valid definition of \( \mu \) will be constrained to the admissible transformations defined on an ordinal scale, for example, humans can be classified into three categories based on their height: “short”, “average”, and “tall”. Given that our intuitive understanding of a height allows establishing more complex relations (such as relative difference captured in \( R_2 \) and \( R_2' \)), we can define a more detailed and useful ratio-scale metric \( \mu \), for example a centimetre. Upon measuring the height of two humans using centimetres, it can be established that the obtained values fully correspond to the empirical knowledge, indicating that: i) homomorphism was preserved; ii) a height can be measured on a ratio scale; and iii) a centimetre is a valid metric in terms of measurement theory.

<table>
<thead>
<tr>
<th>ENTITIES/OBJECTS</th>
<th>EMPIRICAL RELATION SYSTEM</th>
<th>FORMAL RELATIONAL SYSTEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Humans</td>
<td>( R_1 ) taller than: ( T ) human ( a ) is taller than human ( a' ) : ( a \ T \ a' )</td>
<td>( R_1' ) greater than: &gt; ( \mu(a) &gt; \mu(a') : b &gt; b' )</td>
</tr>
<tr>
<td></td>
<td>( R_2 ) N times taller than: ( T_1 ) human ( a ) is N times taller than human ( a' ) : ( a \ T_1 \ a' )</td>
<td>( R_2' ) N times greater than: ( b / b' = N ) ( \mu(a) / \mu(a') = N : b / b' = N )</td>
</tr>
</tbody>
</table>

Table B1. Example mapping from the empirical to formal relational systems

(February, 2009)
Appendix C. Model Notation

Set Theory
A = \{a, b, \ldots, z\} - set;
a \in A - set membership;
|A| - set cardinality;
A \subseteq B - A is a subset of B;
A \subset B - A is a proper subset of B (A not equal to B);
A = \emptyset - empty set;
A \cup B - union;
A \cap B - intersection;
A \cap B = \emptyset - disjoint set
A \times B - Cartesian product;
A - B - relative complement.

Propositional Calculus
\neg - logical NOT;
\land - AND (conjunction);
\lor - OR (inclusive disjunction);
\oplus - exclusive OR (exclusive disjunction).

Predicate Calculus (first-order)
\forall - (universal quantification – “for all”), e.g. if P(n) is the predicate \(1^n > 1+n\) and N is the set of natural numbers, then \(\forall n \in N \ P(n)\) will be false;
\exists - (existential quantification – “there exists”), e.g. if P(n) is the predicate \(n \times 5 < n + 5\) and N is the set of natural numbers, then \(\exists n \in N \ P(n)\) can be true (n=1).

Miscellaneous
<...> angular brackets have been used to represent elements of a set in a case when these elements are sets themselves. e.g. SOS = <SI; \ldots, R>;
\{...\} curly braces have been used to represent atomic elements of a set. e.g. SI = \{si_1, \ldots, si_n\};
(...) parentheses have been used to indicate an ordered pair of elements (representing an edge on the design graph, i.e. a relationship between two implementation elements). e.g. R = \{(a,b), \ldots, (y, z)\}.
Appendix D. User Profile Questionnaire and Pre-test Task Description

<table>
<thead>
<tr>
<th>AGE</th>
<th>18-24</th>
<th>25-29</th>
<th>30-34</th>
<th>35-39</th>
<th>40-49</th>
<th>50+</th>
</tr>
</thead>
</table>

**EDUCATION** (Please tick the boxes that apply and fill in the details. Specify current and completed levels if applicable):

<table>
<thead>
<tr>
<th>Level</th>
<th>Area of Study</th>
<th>Educational Institution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(e.g. Comp. Sci.)</td>
<td>(e.g. RMIT University, Melbourne)</td>
</tr>
</tbody>
</table>

- [ ] Undergraduate
- [ ] Honours
- [ ] Masters
- [ ] PhD
- [ ] Other (e.g. Grad Dip.):

**RELATED WORK EXPERIENCE** (Please check all boxes that are applicable, i.e. multiple job duties and project types are possible). **Note:** If you have worked for more than three employers please state the three most recent.

**Employer 1 (optional):**

<table>
<thead>
<tr>
<th>Job Description(s):</th>
<th>Project/Team Leader</th>
<th>Software Engineer</th>
<th>Analyst/Consultant</th>
<th>Programmer</th>
<th>Research or/and Academic position</th>
<th>Other (describe below)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Position Type:</th>
<th>Full-Time</th>
<th>Part-Time</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Duration:</th>
<th>0-12 months</th>
<th>1-2 years</th>
<th>2-3 years</th>
<th>3-5 years</th>
<th>5+ years</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Project(s):</th>
<th>Object-Oriented</th>
<th>SOA/Web Services</th>
<th>Web Development</th>
<th>Other (please describe)</th>
</tr>
</thead>
</table>

**Employer 2 (optional):**

<table>
<thead>
<tr>
<th>Job Description(s):</th>
<th>Project/Team Leader</th>
<th>Software Engineer</th>
<th>Analyst/Consultant</th>
<th>Programmer</th>
<th>Research or/and Academic position</th>
<th>Other (describe below)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Position Type:</th>
<th>Full-Time</th>
<th>Part-Time</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Duration:</th>
<th>0-12 months</th>
<th>1-2 years</th>
<th>2-3 years</th>
<th>3-5 years</th>
<th>5+ years</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Project(s):</th>
<th>Object-Oriented</th>
<th>SOA/Web Services</th>
<th>Web Development</th>
<th>Other (please describe)</th>
</tr>
</thead>
</table>
APPENDIX D. USER PROFILE QUESTIONNAIRE AND PRE-TEST TASK DESCRIPTION

<table>
<thead>
<tr>
<th>Employer 3 (optional):</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Job Description(s):</td>
<td></td>
</tr>
<tr>
<td>Project/Team Leader</td>
<td></td>
</tr>
<tr>
<td>Programmer</td>
<td></td>
</tr>
<tr>
<td>Research or/and Academic position</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Position Type:</td>
<td></td>
</tr>
<tr>
<td>Full-Time</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Duration:</td>
<td></td>
</tr>
<tr>
<td>0-12 months</td>
<td></td>
</tr>
<tr>
<td>2-3 years</td>
<td></td>
</tr>
<tr>
<td>5+ years</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Project(s):</td>
<td></td>
</tr>
<tr>
<td>Object-Oriented</td>
<td></td>
</tr>
<tr>
<td>Web Development</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Rate your experience with the following software development paradigms:**

**Object-Oriented Development**
- Never Used
- Novice
- Intermediate
- Expert

**Component-Based Development (e.g. EJB/Corba)**
- Never Used
- Novice
- Intermediate
- Expert

**Service-Oriented Computing**
- Never Used
- Novice
- Intermediate
- Expert

**Rate your experience with the Unified Modelling Language (UML 1.1 or 2.0):**
- Never Used
- Novice
- Intermediate
- Expert

**Rate your experience with the Java programming language (J2SE and/or J2EE):**
- Never Used
- Novice
- Intermediate
- Expert

**Rate your (practical) experience with maintaining software products**
- Never Used
- Novice
- Intermediate
- Expert

**Rate your experience with the following Web Service related technologies:**

<table>
<thead>
<tr>
<th></th>
<th>Never Used</th>
<th>Novice</th>
<th>Intermediate</th>
<th>Expert</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOAP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WSDL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BPEL4WS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Rate your understanding of the principles of Service-Oriented Computing (SOC)**
- 1 Low
- 2
- 3
- 4
- 5 High

**Rate your understanding of the maintainability quality characteristic of software**
- 1 Low
- 2
- 3
- 4
- 5 High

**Rate your understanding of the coupling and cohesion structural properties of software**

(February, 2009)
APPENDIX D. USER PROFILE QUESTIONNAIRE AND PRE-TEST TASK DESCRIPTION

1 Low  2  3  4  5 High

Figure F1: Pre-test Questionnaire

PRETEST TASK

Please perform the pretest task described below and then fill-in the required data including a time spent on completing the task and its perceived degree of difficulty. Do not rate the quality of your solution – this will be done by independent assessors).

<table>
<thead>
<tr>
<th>Time spent on task (minutes)</th>
<th>Degree of task difficulty (1 easy to 5 hard)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task (Student system)</td>
<td></td>
</tr>
</tbody>
</table>

Quality of the solution (1-5):

TASK DESCRIPTION:

Draw a UML class diagram (on the following page) and then implement a small Java system that manages student data and has the following requirements:

- there are 4 types of students: Local, International, PostGraduate, UnderGraduate
- all students have the following attributes: student id (int), name (String), address (String), and a collection of academic results with each result represented by course code (String) and grade (int).
  - International students also have a visa attribute represented by visa number (int).
  - PostGraduate students have an additional attribute - program cluster (String)
- all the above attributes are set during the construction time
- you should implement all required getter/setter methods for accessing/mutating student data (no other operations need to be implemented).

Notes: i) you need to record the actual time taken to complete the task – i.e. do not include the reading time; ii) do not compile/run your systems - we are only interested in the actual coding time; and iii) you can base your code on any version of J2SE (1.4- 1.6).

Figure F2: Pre-test Task
Appendix E. Experimental Software Systems - Documentation

1. AMS - Software Requirements Specification (SRS)

This SRS document lists the functional requirements of the Academic Management System (AMS). The requirements are organised ‘by feature’, according to the IEEE 1998 Std 830-1998 standard (Recommended Practice for Software Requirements Specifications) [108].

NOTE: The requirements related to the experimental systems are highlighted in bold. Each requirement includes details related to the associated business rules and logic (if applicable).

Student Management (STM)

Actors: Admin, Student, the Department of Immigration Australia (DIMA) - external, Library - external. Priority: High.

- **STM1. The system shall allow a Student to enrol into the University.**
  Rules: i) There are six different types of students: Local/International, Undergraduate (UG)/Postgraduate (PG), and Full-time/Part-time; and ii) Visa clearance must be obtained from the Department of Immigration Australia (DIMA) for all international students.

- **STM2. The system shall allow Admin to create and add a student to the Student Database**

- **STM3. The system shall allow Admin to remove a student from the Student Database**

- **STM4. The system shall allow Admin to update the details of a course in the Courses Database**

- **STM5. The system shall allow Student to enrol into program**
  Rules: i) Visa clearance must be obtained from the Department of Immigration Australia (DIMA) for all international students; and ii) There are two distinct types of programs: UG and PG, and students can only be assigned to the type corresponding to their status.

- **STM6. The system shall allow Student to enrol into semester**
  - **STM6.1 The system must ensure that the student does not have an outstanding balance with the University Library.**

- **STM7. The system shall allow Student to enrol into course**
  - **STM7.1 The system must enforce the student load integrity**
    - **Rules:**
      - Full Time load range: 36-48 credit points\(^38\); Part Time load: <36;

\(^38\) The credit points assigned to different course types are described in the requirement CM1.
Maximum allowable load per semester: Local [UG/PG] 60 credit points; International [UG/PG] 48 credit points

- STM7.2 The system shall ensure that international students cannot repeat a course if this course had been completed (passed) previously
- STM7.3 The system shall check that the student completed all required prerequisite courses

- STM8. The system shall allow Student to withdraw from course
  - STM8.1 The system must enforce the student load integrity
    ○ Rules:
      - Full Time load range: 36-48 credit points; Part Time load: \(\leq 36\)
      - Minimum allowable load per semester: Local [UG/PG] 18 credit points; International [UG/PG] 36 credit points

- STM9. The system shall allow Student to view his/her academic history and other personal information.

- STM10. The system shall allow Student to validate his/hers eligibility to enrol into a given course (all prerequisite courses must be completed).
- STM11. The system shall allow Student to view the available (eligible) electives for a given semester.

Task Management (TM)

**Actors:** Task Manager, Academic. **Priority:** High.

- TM1. The system shall allow Task Manager to create and add a new academic task to the Task Database.

**Details:** Task data consists of: unique task id (numeric), name (alphabetic), type (three types in total [LECTURING, COURSE_COORDINATING, RESEARCH_SUPERVISING], status [READY, COMPLETED, ACTIVE], and end date. Task types have varying number of work units (points) allocated to them (currently: LECTURING = 10 points, COURSE_COORDINATING = 7 points, and RESEARCH_SUPERVISING = 5 points).
- TM2. The system shall allow Task Manager to remove a task.
- TM3. The system shall allow Task Manager to update the details of a task in the Task Database.
- TM4. The system shall allow Task Manager to specify the duration and number of work units for a task.
- TM5. The system shall allow Task Manager to assign an Academic staff member to a task.
  - TM5.1. The system shall ensure that the workload of an Academic Staff member does not exceed the preset limit of work units (which is currently set to 35 points).
- **TM6.** The system shall calculate the workload (total number of task points) for each Academic Staff member.
- **TM7.** The system shall allow Task Manager to find Academic Staff members whose current workload is below a specified threshold.

**Staff Management (SM)**

**Actors:** Admin, Human Resource Manager. **Priority:** High.

- **SM1. The system shall allow Administrator to add a new Academic Staff member to the Staff Database**
  
  **Rules:** Each academic has an associated academic category (level) There are five academic levels that will influence: i) the academic workload calculation procedure (requirement **TM6**) as follows: level A – load equals current task points x 0.6; level B – task points x 0.7; level C – task points x 0.8; level D – task points x 0.9; level E - task points x 1; and ii) task assignment process (requirement **TM5**) as follows: only academics with level B or higher can undertake the COURSE_COORDINATING tasks, and only academics with level C or higher can undertake the RESEARCH_SUPERVISING task.

  --- note that the above rules are only implemented in the SYS-COUP experimental system.

- **SM2. The system shall allow Administrator to remove an Academic Staff member from the Staff Database**

- **SM3. The system shall allow Administrator to update the details of an Academic Staff member in the Staff Database**

- **SM4. The system shall allow Human Resources Manager to validate the academic promotion requests by checking the performance of the associated academic.** The current simplified business rule is that if the staff load is 100% (35 points worth of tasks), the promotion is granted.

**Program Management (PM)**

**Actors:** Admin, Academic Staff. **Priority:** High.

- **PM1. The system shall allow Admin to create and add a program to the Programs Database.**

- **PM2. The system shall allow Admin to remove a program from the Programs Database.**

- **PM3. The system shall allow Admin to update the details of a program in the Programs Database.**

- **PM4. The system shall allow Admin to assign Academic Staff member as program manager.**
Course Management (CM)

- **CM1. The system shall allow Admin to create and add a course to the Courses Database.** Rule: The courses are categorised into two distinct types: Elective courses: 6 credit points; and Core courses: 12 credit points.
- **CM2. The system shall allow Admin to remove a course from the Courses Database.**
- **CM3. The system shall allow Admin to update the details of a course in the Courses Database.**
- **CM4. The system shall allow Academic Staff member to set the prerequisites for a course.**

Report Generation (RG)
Actors: Admin, Academic Staff. Priority: Medium.

- **RG1. The system shall allow the user to specify a time-span (start/end date) of any report.**
- **RG2. The system shall be able to generate a workplan report for an academic staff member.**
- **RG3. The system shall be able to generate task distribution report (in graph-based, table-based, and chart-based formats).**
- **RG4. The system shall be able to generate enrolment statistics (categorised by student type and status) report covering all enrolled students.**
- **RG5. The system shall be able to generate Academic Staff workload (categorised by task type) reports.**
- **RG6. The system shall be able to generate Grade Point Average (GPA) reports.** Rules:
  i) HD = 4 credit points; DI = 3 points; CR = 2 points; PA = 1 point; NN = 0;
  ii) GPA = total credit points / number of completed courses (results).

Finance Management (FM)

- **FM1. The system shall allow Admin to organise staff salary payments.**
- **FM2. The system shall calculate the academic and non-academic (Student Union fees) for Student on a per semester basis.**
  - Rules:
    - Union (voluntary) fee = fixed rate ($) per credit point × number of semester credit points.
    - The system shall also support fixed minimum and maximum charges. The current values used in AMS are:
      - MIN_LIMIT = $30;
MAX_LIMIT = $150;
VOLUNTARY_CHARGE_PER_POINT = $5;

Academic (compulsory) fee = varied rate ($) per credit point × number of semester credit points. The credit point rate will vary according to the student’s local or international status. There are no minimum and maximum charges for academic fee.

The current values used in AMS are:

LOCAL_STUDENT_CHARGE_PER_POINT = $100;
INTERNATIONAL_STUDENT_CHARGE_PER_POINT = $190;

- FM3. The system shall allow Student to pay the academic and non-academic fees.
- FM4. The system shall allow Admin to send bulk overdue payment reminders to the students (on a per program basis).
2. AMS (SYS-COUP) – UML Class Diagrams

- Overview - high level (service level) system design structure

The complex data types are represented internally as Java Enterprise Edition 5 (JEE 5) entity beans (EJB3). The data access and manipulation is performed using Java Persistence API (JPA) via two dedicated services (ams.data.services.DataAccessService and DataManipulationService). Note that the same data types are used in the cohesion experimental system shown in Section 3 of this appendix.

(February, 2009)
- **ams.services.academic_management service**

  This service was designed to support the evaluation of intra-service coupling metrics. Experimental elements (classes): TaskManager, TaskPointsHandler, LoadCalculator, TaskDataHandler.
- **ams.services.student_management service**

This service was designed to support the evaluation of indirect extra-service coupling metrics. **Experimental elements (classes):** CourseHandler, ElectivesManager, StudentManagementServiceInterface.
This service was designed to support the evaluation of direct extra-service coupling metrics. Experimental elements (classes): VoluntaryPayment, CompulsoryPayment, SeeMasterEnrollment, CourseEnrollment, StudentEnrollment.
3. AMS (SYS-COH) – UML Class Diagrams

- Overview - High level (service level) system design structure
**APPENDIX E. EXPERIMENTAL SOFTWARE SYSTEMS - DOCUMENTATION**

- *ams.services.AMS1 service*

![Diagram of AMS1 service and related classes and interfaces]

- AMS1 Interface
  - `@org.apache.wsdl.WebServiceDesc` Admin
  - `@org.apache.wsdl.WebServiceDesc` TaskManager
  - `@org.apache.wsdl.WebServiceDesc` HRManager
  - `@org.apache.wsdl.WebServiceDesc` Component
  - `@org.apache.wsdl.WebServiceDesc` Student

- AMS1Impl
  - `@type=java.lang.String`
  - `@type=java.lang.String`
  - `@type=java.lang.String`
  - `@type=java.lang.String`

- FinanceChecker
  - `@name=FinanceChecker`
  - `@name=FinanceChecker`
  - `@name=FinanceChecker`
  - `@name=FinanceChecker`

- ResourceManager
  - `@name=ResourceManager`
  - `@name=ResourceManager`
  - `@name=ResourceManager`
  - `@name=ResourceManager`

- WorkloadHandler
  - `@name=WorkloadHandler`
  - `@name=WorkloadHandler`
  - `@name=WorkloadHandler`
  - `@name=WorkloadHandler`

- ElectivesManager
  - `@name=ElectivesManager`
  - `@name=ElectivesManager`
  - `@name=ElectivesManager`
  - `@name=ElectivesManager`

- AMS.data.complexTypes
  - `Course`
  - `Result`
  - `Academic`
  - `Semester`
  - `Student`
  - `Task`

- AMS.data.services
  - `services::DataAccessServiceInterface`
    - `@name=DataAccessServiceInterface`
    - `@name=DataAccessServiceInterface`
    - `@name=DataAccessServiceInterface`
    - `@name=DataAccessServiceInterface`

- AMS1.wssd_interface
  - `@name=AMSI_wssd_interface`
  - `@name=AMSI_wssd_interface`
  - `@name=AMSI_wssd_interface`
  - `@name=AMSI_wssd_interface`

223

*(February, 2009)*
- **ams.services.AMS2 service**
- **ams.services.AMS3 service**
- `ams.services.AMS5 service`
Appendix F. Experimental Tasks

1. Coupling (SYS-COUP) Task

The coupling related tasks are separated into: i) determining the cause of failures; and ii) implementing changes to the system requirements (business logic and rules). Note that the provided system implementation includes source-code comments explaining the business rules and logic related to the system failures and/or required changes shown below.

1.a Determining the Cause of Failures

This task requires you to determine the cause of identified failures shown in the table below. All failures are caused by the incorrect implementation of the associated functionality.

<table>
<thead>
<tr>
<th>Failure id</th>
<th>Requirement id (from SRS)</th>
<th>Failure Synopsis</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ams.services.academic_management</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMF1</td>
<td>TM5</td>
<td>The service fails to correctly perform the required task assignment validation checks. i.e. it incorrectly assigns tasks to academics.</td>
</tr>
<tr>
<td>AMF2</td>
<td>RG5</td>
<td>The service fails to correctly generate workload statistics (number of tasks in each task category). i.e. the returned statistics are incorrect for each task type.</td>
</tr>
<tr>
<td><strong>ams.services.student_management</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SMF1</td>
<td>STM11</td>
<td>The service fails to correctly generate the list of (eligible) available elective courses. i.e. the returned collection of electives includes unsupported electives.</td>
</tr>
<tr>
<td>SMF2</td>
<td>STM8</td>
<td>The service fails to correctly perform the required course withdrawal checks when withdrawing a student from a course. i.e. it allows students to incorrectly withdraw from a course.</td>
</tr>
</tbody>
</table>
### APPENDIX F. EXPERIMENTAL TASKS

**ams.services.enrollment_support**

<table>
<thead>
<tr>
<th>Requirement id</th>
<th>Functionality Change Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ESF1</strong> STM7</td>
<td>The service fails to correctly perform the required enrolment checks when <em>enrolling a student into a course</em>. i.e. it allows students to incorrectly enrol in a course.</td>
</tr>
<tr>
<td><strong>ESF2</strong> STM6</td>
<td>The service fails to correctly perform the required enrolment checks when <em>enrolling a student into a semester</em>. i.e. it allows invalid students to enrol in a semester.</td>
</tr>
</tbody>
</table>

#### 1.b Changes to existing functionality

This task requires you to implement changes to the functional requirements of AMS. **Make sure that you examine the entire system, after performing the required changes, in order to avoid introducing new faults.**

<table>
<thead>
<tr>
<th>Change id</th>
<th>Requirement id (from SRS)</th>
<th>Functionality Change Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AMC1</strong> TM6</td>
<td>The TaskPointsHandler.calculateTaskPoints() operation should now receive a collection of <em>all academics</em> in the system as its parameter. <strong>You need to implement this change and also update any affected system elements.</strong></td>
<td></td>
</tr>
<tr>
<td><strong>AMC2</strong> TM1</td>
<td>Instead of returning the Task complex type, the TaskDataHandler.createNewTask() operation should now return the number of workload points of this task. <strong>You need to implement this change and also update any affected system elements.</strong></td>
<td></td>
</tr>
<tr>
<td><strong>AMC3</strong> TM5</td>
<td>Assigning academic to task procedure should make sure that this academic is not assigned to more than three tasks of the same type (Lecturing, etc.). Also, all level A and B academics can now be overloaded by up to 10 task points.</td>
<td></td>
</tr>
</tbody>
</table>
**APPENDIX F. EXPERIMENTAL TASKS**

<table>
<thead>
<tr>
<th>AMC4</th>
<th>RG5</th>
<th>Generating workload statistics procedure should now create and return an array that includes task statistics for all academics in the system. Also, instead of counting the number of tasks in each task category, the procedure should count the total task points for all the tasks in a particular category.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ams.services.student_management</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SMC1</td>
<td>STM9</td>
<td>Instead of returning an array of academic Results for all students enrolled in a given course, the viewResults service interface operation should now return a collection of Strings representing the grades received by the students. You need to implement this change and also update any affected system elements.</td>
</tr>
<tr>
<td>SMC2</td>
<td>RG6</td>
<td>Instead of calculating and returning the GPA value for a given student, the calculateGPA service interface operation should now return the average GPA for all the students in a given course. You need to implement this change and also update any affected system elements.</td>
</tr>
<tr>
<td>SMC3</td>
<td>STM11</td>
<td>The find available electives procedure is no longer required to check if the course type is the same as the student type for PG International students. Also, the system should ensure that the returned collection of electives does not contain more than three electives managed by the same course leader (academic).</td>
</tr>
<tr>
<td>SMC4</td>
<td>STM8</td>
<td>The minimum allowable load (used when validating the course withdrawal requests) for the international students is now the same as the one for local students. Also, the system should ensure that all students are enrolled in at least two courses (the withdrawal requests should be rejected otherwise).</td>
</tr>
</tbody>
</table>

(February, 2009)
### Experimental Tasks

**ams.services.enrollment_support**

<table>
<thead>
<tr>
<th>Task</th>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ESC1</strong></td>
<td>FM2</td>
<td>Instead of calculating and returning the compulsory academic fee, the corresponding service operation <code>calculateAcademicFees()</code> should now return the predefined credit point rates for both international and local students. You need to implement this change and also update any affected system elements.</td>
</tr>
<tr>
<td><strong>ESC2</strong></td>
<td>FM2</td>
<td>The <code>calculateStudentUnionFees()</code> operation does not need to enforce the minimum and maximum student union charges, only return the calculated fee. You need to implement this change and also update any affected system elements.</td>
</tr>
<tr>
<td><strong>ESC3</strong></td>
<td>STM7</td>
<td>When validating prerequisites upon course enrolment of International students, any (one) completed course is now sufficient to satisfy the prerequisite constraint. Also, the student load status for all PG part-time students should be upgraded to full-time as needed (i.e. if this enrolment results in a change of load status).</td>
</tr>
<tr>
<td><strong>ESC4</strong></td>
<td>STM6</td>
<td>Upon enrolling Local students into a semester, the system shall obtain library clearance from the external partner (LIBRARY). Also, the visa check for International students should be performed for UG International students only.</td>
</tr>
</tbody>
</table>

### 2. Cohesion (SYS-COH) Task

This task requires you to determine the cause of failures (see below) in the provided experimental systems. More specifically, the behaviour of all operations included in the interfaces of the AMS1-AMS5 services is faulty due to the incorrect implementation of the provided functionality in their constituent implementation elements (classes).
## APPENDIX F. EXPERIMENTAL TASKS

<table>
<thead>
<tr>
<th>Failure Number</th>
<th>Requirement Id (from SRS)</th>
<th>Source of Failure</th>
<th>Synopsis</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Service AMS1</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>F1-AMS1</strong></td>
<td>SM4</td>
<td>validatePromotionRequest</td>
<td>Promotion requests are granted incorrectly to academics that do not maintain full academic load.</td>
</tr>
<tr>
<td><strong>F2-AMS1</strong></td>
<td>STM10</td>
<td>findAvailableElectives</td>
<td>The collection of returned electives contains unsupported (invalid) electives.</td>
</tr>
<tr>
<td><strong>F3-AMS1</strong></td>
<td>TM7</td>
<td>evaluateWorkloadDistribution</td>
<td>The collection of returned academics is incorrect (e.g. it contains academics with a load above the defined cut-off limit).</td>
</tr>
<tr>
<td><strong>F4-AMS1</strong></td>
<td>FM4</td>
<td>sendPaymentReminders</td>
<td>The system sends multiple duplicate reminders to the students who have already payed their semester fees.</td>
</tr>
<tr>
<td><strong>Service AMS2</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>F1-AMS2</strong></td>
<td>STM9</td>
<td>viewAcademicResults</td>
<td>The returned results are incomplete. i.e. there are missing courses.</td>
</tr>
<tr>
<td><strong>F2-AMS2</strong></td>
<td>RG6</td>
<td>calculateGPA</td>
<td>The GPA values calculated by this operation differ from their expected values.</td>
</tr>
<tr>
<td><strong>F3-AMS2</strong></td>
<td>RG5</td>
<td>generateWorkloadStatistics</td>
<td>The statistics returned by the operation are incorrect for all task types.</td>
</tr>
<tr>
<td><strong>F4-AMS2</strong></td>
<td>STM8</td>
<td>withdrawFromCourse</td>
<td>The operation does not allow international and local students to withdraw from the course.</td>
</tr>
<tr>
<td><strong>Service AMS3</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>F1-AMS3</strong></td>
<td>FM2-3</td>
<td>paySemesterFee</td>
<td>International students are undercharged in some situations.</td>
</tr>
<tr>
<td><strong>F2-AMS3</strong></td>
<td>FM2-3</td>
<td>payStudentUnionFee</td>
<td>Students are overcharged in some situations.</td>
</tr>
<tr>
<td><strong>F3-AMS3</strong></td>
<td>TM1</td>
<td>createTask</td>
<td>The workload points are assigned incorrectly to tasks.</td>
</tr>
<tr>
<td><strong>F4-AMS3</strong></td>
<td>TM5</td>
<td>assignAcademicToTask</td>
<td>The maximum load limit is not maintained in some cases (the academic gets overloaded).</td>
</tr>
</tbody>
</table>
### Service AMS4

<table>
<thead>
<tr>
<th>Task</th>
<th>Service</th>
<th>Operation/Function</th>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1-AMS4</td>
<td>STM1</td>
<td>enrollIntoUniversity</td>
<td>The visa details obtained from DIMA cannot be retrieved in later system interactions.</td>
</tr>
<tr>
<td>F2-AMS4</td>
<td>STM5</td>
<td>programEnrollment</td>
<td>The program enrolment fails when enrolling local students.</td>
</tr>
<tr>
<td>F3-AMS4</td>
<td>STM6</td>
<td>semesterEnrollment</td>
<td>The operation fails when enrolling valid students.</td>
</tr>
<tr>
<td>F4-AMS4</td>
<td>RG4</td>
<td>viewEnrollmentStatistics</td>
<td>The statistics returned by the operation are incorrect for all student types.</td>
</tr>
</tbody>
</table>

### Service AMS5

<table>
<thead>
<tr>
<th>Task</th>
<th>Service</th>
<th>Operation/Function</th>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1-AMS5</td>
<td>STM7.1</td>
<td>validateCourseEligibility</td>
<td>The course eligibility check fails when validating international students.</td>
</tr>
<tr>
<td>F2-AMS5</td>
<td>STM7.2</td>
<td>checkPrerequisites</td>
<td>The prerequisite check fails on all inputs.</td>
</tr>
<tr>
<td>F3-AMS5</td>
<td>STM7.3</td>
<td>validateStudyLoadLimit</td>
<td>The operation incorrectly detects overload for local students; and also fails to detect the overload for international students.</td>
</tr>
<tr>
<td>F4-AMS5</td>
<td>STM7</td>
<td>enrollIntoCourse</td>
<td>The operation creates a new enrolment, as expected, but the enrolment data cannot be retrieved in later system interactions.</td>
</tr>
</tbody>
</table>
References


(February, 2009)


[60] V. Doubrovski, IBM Global Services Australia and New Zealand (Senior IT Architect) *purpose of contact (Service-Oriented Computing)*, 2006.


[99] B. Henderson-Sellers, University of Technology Sydney, Australia (Professor of Information Systems) brian@it.uts.edu.au, purpose of contact (software metrics), 2005.


[141] G. Lewis, Carnegie Mellon University, USA (Senior member of the technical staff at SEI) glewis@sei.cmu.edu, *purpose of contact (Service-Oriented Computing, software maintainability)*, 2005.


[182] M. Papazoglou, Tilburg University, The Netherlands (Professor in Computer Science) mikep@uvt.nl, purpose of contact (Service-Oriented Computing), 2005.


243

(February, 2009)


M. Rosemann, Queensland University of Technology, Australia (Professor of Information Systems) m.rosemann@qut.edu.au, purpose of contact (Service-Oriented Computing, Business Process Modelling), 2005.


(February, 2009)
[248] O. Zimmermann, IBM Zurich Research Lab, Switzerland (Research Staff Member/Senior Architect) olz@zurich.ibm.com, *purpose of contact (Service-Oriented Computing)*, 2005.